

Diseño y evaluación de arquitecturas de computadoras

Marta Beltrán Pardo
Antonio Guzmán Sacristán

PEARSON

Diseño y evaluación de arquitecturas de computadoras

Marta Beltrán Pardo

Profesora Titular de Universidad
Área de Arquitectura y Tecnología de Computadores
Universidad Rey Juan Carlos de Madrid

Antonio Guzmán Sacristán

Profesor Contratado Doctor
Área de Arquitectura y Tecnología de Computadores
Universidad Rey Juan Carlos de Madrid

Prentice Hall
es un sello editorial de



Harlow, England • London • New York • Boston • San Francisco • Toronto • Sydney • Singapore • Hong Kong
Tokyo • Seoul • Taipei • New Delhi • Cape Town • Madrid • Mexico City • Amsterdam • Munich • Paris • Milan

Datos de catalogación bibliográfica

Marta Beltrán Pardo, Antonio Guzmán Sacristán

Diseño y evaluación de arquitecturas de computadoras
PEARSON EDUCACIÓN, S.A., Madrid, 2010

ISBN: 978-84-8322-650-6

Materia: 004, Computadores

Formato: 195 × 250 mm

Páginas: 344

Todos los derechos reservados.

Cualquier forma de reproducción, distribución, comunicación pública o transformación de esta obra sólo puede ser realizada con la autorización de sus titulares, salvo excepción prevista por la ley. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. Código penal*).

Diríjase a CEDRO (Centro Español de Derechos Reprográficos: www.cedro.org), si necesita fotocopiar o escanear algún fragmento de esta obra.

DERECHOS RESERVADOS
© 2010, PEARSON EDUCACIÓN, S.A.
C/ Ribera del Loira, 28
28042 Madrid (España)
www.pearsoneducacion.com

PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN

ISBN: 978-84-8322-650-6
Depósito Legal: M-

Equipo editorial:

Editor: Miguel Martín-Romo
Técnico editorial: Esther Martín

Equipo de producción:

Director: José Antonio Clares
Técnico: Isabel Muñoz

Diseño de cubierta: Equipo de Diseño de PEARSON EDUCACIÓN, S. A.

Composición: JOSUR TRATAMIENTO DE TEXTOS, S.L.

Impreso por:

Nota sobre enlaces a páginas web ajena: Este libro puede incluir enlaces a sitios web gestionados por terceros y ajenos a PEARSON EDUCACIÓN S.A. que se incluyen sólo con finalidad informativa.

PEARSON EDUCACIÓN S.A. no asume ningún tipo de responsabilidad por los daños y perjuicios derivados del uso de los datos personales que pueda hacer un tercero encargado del mantenimiento de las páginas web ajena a PEARSON EDUCACIÓN S.A. y del funcionamiento, accesibilidad o mantenimiento de los sitios web no gestionados por PEARSON EDUCACIÓN S.A. Las referencias se proporcionan en el estado en que se encuentran en el momento de publicación sin garantías, expresas o implícitas, sobre la información que se proporcione en ellas.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos.

Contenido

Introducción

1.	Definición de arquitectura de computadoras	XI
2.	Objetivos de este libro	XII
3.	Cómo usar este libro	XIV
4.	Agradecimientos	XV

Capítulo 1: Conceptos básicos de procesadores

1.1.	Diseño de un repertorio de instrucciones	2
1.1.1.	Decisiones acerca del tipo de almacenamiento de operandos	3
1.1.2.	Decisiones acerca de la interpretación de las direcciones de memoria y de los modos de direccionamiento soportados.....	7
1.1.3.	Otras decisiones	11
1.1.4.	Codificación del repertorio de instrucciones.....	12
1.1.5.	Mejoras y optimizaciones del repertorio de instrucciones	14
1.2.	Mecanismo completo de ejecución de una instrucción	15
1.3.	Evaluación de prestaciones de un procesador	18
1.4.	Diseño de procesadores secuenciales.....	19
1.4.1.	Procesador monociclo	20
1.4.2.	Procesador multiciclo	25
1.4.3.	Tratamiento de excepciones en procesadores secuenciales	33
1.5.	Diseño de procesadores segmentados.....	35
1.5.1.	Conceptos básicos de segmentación	36
1.5.2.	Resolución de riesgos en procesadores segmentados	40
1.5.3.	Procesadores segmentados multifuncionales	58
1.5.4.	Tratamiento de excepciones.....	63
	RESUMEN DE DECISIONES DE DISEÑO DEL REPERTORIO DE INSTRUCCIONES	66
	RESUMEN DE DECISIONES DE DISEÑO DEL PROCESADOR	67
	Procesadores secuenciales	67
	Procesadores segmentados	67
	BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS	68
	PROBLEMAS	69
	AUTOEVALUACIÓN	73

Capítulo 2: Conceptos básicos de memoria y E/S

2.1.	Diseño de una jerarquía de memoria básica	76
2.2.	Mecanismo completo de acceso a memoria	79

2.3.	Evaluación de prestaciones de la jerarquía de memoria	81
2.4.	Niveles de la jerarquía de memoria	82
2.4.1.	Diseño de la memoria caché.....	82
2.4.2.	Diseño de la memoria principal	100
2.4.3.	Diseño de la memoria virtual	104
2.5.	Diseño de un sistema de E/S básico	108
2.6.	Mecanismo completo de una operación de E/S.....	110
2.7.	Evaluación de prestaciones del sistema de E/S	111
2.8.	Buses de E/S.....	112
2.8.1.	Diseño de buses de E/S.....	113
2.8.2.	Diseño de jerarquías de buses.....	117
2.9.	Gestión del sistema de E/S	120
2.9.1.	E/S programada con espera de respuesta.....	120
2.9.2.	E/S con interrupciones	120
2.9.3.	E/S con acceso directo a memoria	126
RESUMEN DE DECISIONES DE DISEÑO DE LA JERARQUÍA DE MEMORIA		130
	Memoria caché	130
	Memoria principal	130
	Memoria virtual.....	131
	Buses de E/S.....	131
	Gestión E/S.....	132
BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS		132
PROBLEMAS		133
AUTOEVALUACIÓN.....		136

Capítulo 3: Técnicas de aumento de prestaciones para procesadores

3.1.	Planificación dinámica de instrucciones	138
3.1.1.	Planificación de instrucciones de acceso a memoria	139
3.1.2.	Planificación dinámica de instrucciones centralizada	141
3.1.3.	Planificación dinámica de instrucciones distribuida	145
3.2.	Predicción dinámica de saltos	150
3.2.1.	Estructuras hardware para la predicción de salto	151
3.2.2.	Predictores de salto	161
3.3.	Emisión múltiple de instrucciones.....	167
3.3.1.	VLIW.....	167
3.3.2.	Superescalar estática	169
3.3.3.	Superescalar dinámica.....	172
3.4.	Especulación	173
3.5.	Multithreading.....	181
RESUMEN DE DECISIONES DE DISEÑO DE TÉCNICAS DE AUMENTO DE PRESTACIONES PARA PROCESADORES		184
	Planificación dinámica de instrucciones	184
	Predicción dinámica de saltos	184

Emisión múltiple de instrucciones y especulación	185
Paralelismo a nivel de thread	185
BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS	185
PROBLEMAS	186
AUTOEVALUACIÓN.....	190

Capítulo 4: Técnicas de aumento de prestaciones para memoria y E/S

4.1. Técnicas de optimización para la memoria caché.....	192
4.1.1. Reducción de la penalización por fallo.....	192
4.1.2. Reducción de la tasa de fallos	196
4.1.3. Reducción del tiempo de acceso	200
4.2. Técnicas de optimización para la memoria principal.....	206
4.2.1. Incremento del ancho de banda	207
4.3. Técnicas de optimización conjunta para todos los niveles de la jerarquía: visión global de la jerarquía de memoria	211
4.3.1. Soporte a las técnicas de optimización del procesador	211
4.3.2. Almacenamiento local vs memoria caché.....	212
4.4. Técnicas de optimización para los buses de E/S	212
4.4.1. Ancho de datos y frecuencia de operación	212
4.4.2. Número de transferencias por ciclo	214
4.4.3. Utilización de protocolos de comunicaciones de alto rendimiento	215
4.5. Técnicas de optimización para los dispositivos de E/S.....	217
4.5.1. Optimización para los dispositivos de almacenamiento: RAID	217
4.6. Técnicas de optimización para la gestión de E/S	223
4.6.1. Procesador para el sistema gráfico: GPU	223
RESUMEN DE DECISIONES DE DISEÑO DE TÉCNICAS DE AUMENTO DE PRESTACIONES PARA MEMORIA Y E/S.....	228
Aumento de prestaciones de la memoria caché	228
Aumento de prestaciones de la memoria principal	228
Aumento de prestaciones de buses de E/S	228
Aumento de prestaciones de dispositivos de E/S: almacenamiento.....	229
Aumento de prestaciones en la gestión de E/S.....	229
BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS	229
PROBLEMAS	230
AUTOEVALUACIÓN.....	233

Capítulo 5: Sistemas multiprocesador y multicamputador

5.1. Clasificación de arquitecturas con varios procesadores	236
5.1.1. Arquitecturas de memoria compartida.....	237
5.1.2. Arquitecturas de memoria compartida-distribuida	238
5.1.3. Arquitecturas de memoria distribuida.....	239

5.1.4. Arquitecturas on-chip de memoria compartida.....	240
5.1.5. Arquitecturas on-chip de memoria distribuida.....	242
5.2. Redes dentro de arquitecturas de computadoras.....	242
5.2.1. Topología	245
5.2.2. Técnicas de conmutación	249
5.2.3. Técnicas de encaminamiento	252
5.2.3. Técnicas de control de flujo	253
5.3. Diseño de arquitecturas de memoria compartida.....	254
5.3.1. Soluciones para el problema de la coherencia.....	257
5.3.2. Soluciones para el problema de la consistencia	269
5.3.3. Soluciones para el problema de la sincronización.....	272
5.4. Diseño de arquitecturas de memoria compartida-distribuida.....	274
5.4.1. Soluciones para el problema de la coherencia.....	275
5.5. Diseño de arquitecturas de memoria distribuida	283
5.5.1. Clusters	283
5.5.2. Grids	286
RESUMEN DE DECISIONES DE DISEÑO DE SISTEMAS MULTIPROCESADOR Y MULTICOMPUTADOR	288
Diseño de redes dentro de arquitecturas.....	288
Diseño de arquitecturas de memoria compartida.....	288
Diseño de arquitecturas de memoria compartida-distribuida.....	288
Diseño de arquitecturas de memoria distribuida	289
BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS	289
PROBLEMAS	290
AUTOEVALUACIÓN	296

Capítulo 6: Evaluación de prestaciones

6.1. Definición de métricas de rendimiento	298
6.2. Métricas de rendimiento sencillas	299
6.3. Evaluación y comparación de rendimiento	300
6.3.1. Ley de Amdhal y Ley de Gustafson	303
6.4. Técnicas de medida y benchmarks	307
6.4.1. Técnicas de medida	308
6.4.2. Tipos de benchmark	310
6.5. Métricas de rendimiento complejas.....	312
6.5.1. Speedup en sistemas paralelos y eficiencia.....	313
6.5.2. Escalabilidad.....	315
BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS	317
PROBLEMAS	318
AUTOEVALUACIÓN	320
Índice analítico.....	321

Introducción

Contenidos

1. Definición de arquitectura de computadoras
2. Objetivos de este libro
3. Cómo usar este libro
4. Agradecimientos

La idea de este libro surge tras impartir las asignaturas de Arquitectura de Computadores de 3.^º de Ingeniería Informática y Arquitectura e Ingeniería de Computadores de 4.^º de Ingeniería Informática en la Universidad Rey Juan Carlos de Madrid durante los seis últimos cursos.

El objetivo principal que nos planteamos fue el de proporcionar a los docentes y a los estudiantes un libro de texto en castellano que cubriera los contenidos típicos de las asignaturas de Arquitectura de Computadores de los últimos años de carrera con una visión completa y equilibrada del tema, y un enfoque puramente pedagógico.

Este objetivo ha supuesto un gran reto, principalmente por tres motivos. El primero, la gran variedad de sistemas que hoy en día se agrupan en la denominación “computadora” (variedad en aplicación, prestaciones, complejidad, coste, etc.). El segundo, el ritmo vertiginoso al que cambian todos estos sistemas, tanto en lo que se refiere a la tecnología como al diseño, y la complejidad de las técnicas empleadas en la actualidad. Y el tercero, y no por ello menos importante, la incertidumbre generada por la adaptación de los estudios al Espacio Europeo de Educación Superior (EEES).

A pesar de todo, este libro intenta presentar los conceptos fundamentales relacionados con el diseño y evaluación de arquitecturas de computadoras, teniendo en cuenta las técnicas generales que pueden aplicarse a cualquier tipo de arquitectura, sentando las bases para que el estudiante pueda comprender las tecnologías y diseños que aparezcan en el futuro por complejas que éstas sean, y proporcionando una herramienta de trabajo muy valiosa para la adquisición por parte de los estudiantes de los conocimientos, capacidades y destrezas correspondientes, tanto en sus clases lectivas, teóricas o prácticas, como en sus horas de estudio individual.

1 Definición de arquitectura de computadoras

La definición de la Arquitectura de Computadoras ha generado multitud de discusiones desde la aparición de la que puede considerarse la primera computadora en 1943, la ENIAC.

Desde este momento diferentes autores han propuesto diferentes descomposiciones de una computadora según el nivel de detalle y abstracción con que se estudie. En todos los casos cada uno de los niveles de descripción está gobernado por sus propios lenguajes de representación, por sus reglas de diseño y por sus leyes de funcionamiento. Cada nivel toma como elementos fundamentales o primitivas los constituidos en los niveles inferiores. Además de existir esta jerarquía externa, dentro de cada nivel suele existir una jerarquía interna de sistemas y subsistemas que facilita el estudio de sus elementos y de las relaciones entre ellos.

La idea principal de todas las propuestas realizadas en este sentido es la misma: definir una descomposición de la computadora en niveles de estudio que permita su descripción jerárquica y que facilite su comprensión y diseño.

¿Qué relación tienen estos niveles de estudio con la Arquitectura de Computadoras? La primera vez que se definió este término fue en un artículo que presentaba la familia 360 de IBM y se definía como “La estructura de una computadora que debe conocer un programador en lenguaje máquina para escribir programas correctos para esa máquina”. Una definición similar propusieron Amdhal, Blaaw y Brooks en 1964 “Los atributos de una computadora tal y como los ve un programador en lenguaje ensamblador, comprende la estructura conceptual y el modelo funcional (modelo de programación)”.

Pero hoy en día las computadoras son sistemas demasiado sofisticados para simplificar tanto la definición de su arquitectura. Para completar un poco estas aproximaciones basadas en niveles de descripción, comencemos por algunas definiciones que se encuentran en la literatura del área y que nos permiten acercarnos a esta cuestión:

- Materia que comprende el diseño, la fabricación, la configuración y la explotación eficiente de las computadoras.

- Campo de la informática que se encarga del estudio de todos los elementos, software y hardware necesarios para el diseño de un sistema computacional completo.
- Arte de diseñar una computadora que cumpla las necesidades del usuario teniendo en cuenta las limitaciones tecnológicas y económicas existentes.
- Materia que se encarga del estudio de las relaciones entre los componentes que forman una computadora.
- Estudio de la estructura y organización de los componentes hardware y software que forman una computadora.

De estas definiciones y de muchas otras similares que pueden encontrarse en la bibliografía, se pueden extraer algunas conclusiones. La primera, se puede considerar que la Arquitectura de Computadoras se encarga del estudio de los elementos de la computadora que están por debajo de la visión ofrecida a los usuarios y programadores finales y a sus aplicaciones. Este estudio debe comprender cuatro tipos de aspectos: los relacionados con el repertorio de instrucciones de la computadora, los relacionados con su organización, los relacionados con su implementación concreta, y por último, los relacionados con la evaluación de su rendimiento. La segunda, se trata de una materia estrechamente relacionada con el estudio de la estructura y organización de la computadora y con las relaciones entre sus componentes. Pero no se trata sólo de un estudio teórico, sino que también es necesario saber cómo diseñar la computadora, según esta organización y las relaciones entre sus elementos, para que cumpla unos requisitos específicos.

Por último, cabe destacar dos aspectos comúnmente olvidados:

- La Arquitectura de Computadoras no sólo se encarga de los elementos hardware sino también de los elementos software. Esto se debe a que en muchos casos ambos aspectos se hallan estrechamente relacionados y no se pueden estudiar por separado. El ejemplo más claro es el de los sistemas operativos, estudiados muy frecuentemente dentro del área de Arquitectura de Computadoras por su estrecha relación con el hardware de la computadora. Por ejemplo, en el estudio de la memoria virtual, que no puede comprenderse de manera exhaustiva atendiendo sólo a aspectos hardware.
- El término Computadoras debe ser completamente general y no restrictivo. Es decir, no se deben estudiar sólo computadoras de propósito general sino también sistemas específicos diseñados para aplicaciones o tareas concretas de diferentes tamaños, costes y prestaciones.

Una vez definida la Arquitectura de Computadoras, queda clara la importancia que el estudio de esta materia adquiere en las diferentes titulaciones de ingeniería relacionadas con las Tecnologías de la Información y las Comunicaciones (TIC).

Desde los años 70, organismos internacionales como ACM o IEEE han emitido periódicamente informes que realizan recomendaciones para el currículum de las titulaciones relacionadas con la informática. Finalmente ambas instituciones han unificado sus esfuerzos y desde 1980 realizan una serie de recomendaciones comunes que se han resumido en los *Computing Curricula*. Las recomendaciones de estas dos organizaciones establecen las grandes líneas que deben orientar la educación de los profesionales del mundo de la informática teniendo en cuenta las necesidades y demandas de la sociedad actual, y en la mayor parte de los perfiles profesionales propuestos, la Arquitectura de Computadoras tiene un gran peso en la formación de los egresados y en las competencias que éstos deben adquirir.

2 Objetivos de este libro

Como se señalaba en la introducción de este capítulo, nuestro objetivo principal es proporcionar a los docentes y a los estudiantes de Arquitectura de Computadoras un libro de texto en castellano que

recopile de manera sistemática las técnicas más importantes de diseño y evaluación de arquitecturas de computadoras.

A este objetivo se suma nuestra intención de escribir el libro con un enfoque puramente didáctico, por lo que el texto incorpora un gran número de figuras ilustrativas y ejemplos resueltos, resúmenes con los conceptos más importantes, propuestas de ejercicios y autoevaluaciones, etc. Además, a lo largo de todo este libro se utiliza la arquitectura de un procesador de tipo RISC, el nanoMIPS, como ejemplo didáctico que permita comprender todas las técnicas estudiadas y concretarlas en una misma arquitectura para todas las explicaciones y ejemplos. Este procesador sólo tiene fines pedagógicos, pero su arquitectura se basa en los mismos principios de diseño que otras de tipo RISC como el MIPS, el SPARC o el PowerPC.

Nuestro objetivo es que tras el estudio de este libro el estudiante adquiera las siguientes competencias:

1. Comprender el funcionamiento de las principales unidades funcionales que componen la computadora (procesador, memoria y sistema de E/S).
2. Analizar los aspectos de rendimiento más importantes de cada una de las unidades funcionales y relacionarlos con el rendimiento de las demás.
3. Diseñar y configurar una computadora para que cumpla unos requisitos de altas prestaciones.
4. Conocer las técnicas de aumento de prestaciones que se aplican en diferentes niveles en las computadoras actuales.
5. Evaluar de manera fiable el rendimiento de una computadora y/o de las unidades funcionales que la componen utilizando para ello las métricas de rendimiento adecuadas.
6. Utilizar los resultados obtenidos de esta evaluación para detectar cuellos de botella, comparar diferentes alternativas, optimizar el rendimiento, etc.

Y que el profesor disponga de una herramienta que directamente le permita preparar su asignatura para que el alumno obtenga estas competencias sin necesidad de estar filtrando y fusionando información de distintas fuentes de referencia. Para ello este libro se organiza en tres partes:

- Primera parte: Diseño y evaluación de procesadores, memorias y sistemas de E/S básicos (competencias 1 y 2).
 - **Capítulo 1. Conceptos Básicos de Procesadores.** En este capítulo se comienza por estudiar el diseño de un repertorio de instrucciones para pasar a comprender el mecanismo completo de ejecución de una instrucción. A continuación se proporcionan los fundamentos de evaluación de prestaciones del procesador y se estudia el diseño de procesadores secuenciales y de procesadores segmentados básicos.
 - **Capítulo 2. Conceptos Básicos de Memoria y E/S.** En este capítulo se comienza estudiando el diseño de una jerarquía de memoria básica, el mecanismo completo de acceso a memoria y la evaluación de prestaciones de la jerarquía de memoria para pasar a analizar los diferentes niveles que componen la jerarquía de memoria. A continuación se estudia el diseño de un sistema de E/S básico, el mecanismo completo de una operación de E/S y la evaluación de prestaciones del sistema de E/S. Por último, con todos estos conceptos básicos de E/S, se pasa a profundizar en el diseño de buses de E/S y de mecanismos de gestión de E/S.
- Segunda parte: Diseño y evaluación de procesadores, memorias y sistemas de E/S avanzados (competencias 3, 4, 5 y 6).
 - **Capítulo 3. Técnicas de aumento de prestaciones para procesadores.** En este capítulo se estudian las técnicas de aumento de prestaciones más utilizadas en la actualidad en el diseño de procesadores: planificación dinámica de instrucciones, predicción dinámica de saltos, emisión múltiple de instrucciones, especulación y multithreading.

- **Capítulo 4. Técnicas de aumento de prestaciones para memoria y E/S.** Este capítulo se centra en el estudio de las técnicas que permiten aumentar el rendimiento de la jerarquía de memoria (estudiando técnicas para la memoria caché y para la memoria principal) y del sistema de E/S (aumento de prestaciones de los buses, de los dispositivos y de los mecanismos de gestión).
- **Capítulo 5. Sistemas multiprocesador y multicamputador.** En el último capítulo de esta segunda parte se comienza por realizar una clasificación de arquitecturas con varios procesadores para pasar a continuación a estudiar los conceptos básicos de redes dentro de arquitecturas de computadoras. A continuación se estudia el diseño de arquitecturas de memoria compartida, de memoria compartida-distribuida y de memoria distribuida.
- Tercera parte: Profundización en la evaluación de rendimiento, selección de métricas, técnicas y herramientas de medida, comparación de rendimiento, etc. (competencias 5 y 6).
- **Capítulo 6. Evaluación de prestaciones.** En este capítulo se discute acerca de la definición de métricas de rendimiento adecuadas, se estudian métricas de rendimiento sencillas y cómo utilizarlas para realizar evaluación y comparación de rendimiento. A continuación se estudian las técnicas de medida y benchmarks necesarios para obtener los valores de estas métricas y por último, se estudian métricas de rendimiento complejas necesarias para evaluar las prestaciones de arquitecturas con más de un procesador.

3 Cómo usar este libro

Para comenzar a estudiar este libro es necesario haber estudiado previamente alguna asignatura de Informática Básica que haya proporcionado al estudiante una visión panorámica de la informática y las primeras nociones básicas acerca de la representación de información dentro de la computadora y de la arquitectura Von Neumann. Por otro lado, también sería deseable que el estudiante conociera los conceptos básicos de Fundamentos de Computadoras (con toda la base en electrónica analógica y digital) y de Estructura y Tecnología de Computadoras, para tener clara una descripción básica de las diferentes unidades funcionales de la computadora (procesador, memoria y periféricos) y nociones de lenguaje ensamblador.

Los capítulos de este libro están ordenados para que su estudio secuencial permita al estudiante obtener las competencias antes mencionadas. Sin embargo, todos ellos son autocontenidos y existen diferentes itinerarios que se pueden seguir para estudiar con este libro si no se está siguiendo el programa de ninguna asignatura concreta o si se emplea la técnica del estudio autónomo. A continuación se sugieren los más habituales:

Itinerario 1: Arquitectura de Computadoras	Capítulo 1 Capítulo 2
Itinerario 2: Arquitectura e Ingeniería de Computadoras o Ampliación de Arquitectura de Computadoras	Capítulo 3 Capítulo 4 Capítulo 5 Capítulo 6
Itinerario 3: Arquitectura de Procesadores	Capítulo 1 Capítulo 3 Capítulo 5

En cualquier caso se sugiere al estudiante que estude con atención los ejemplos, que intente resolver los problemas propuestos (que están siempre en orden expositivo, según se han presentado los conceptos

en cada capítulo) y que utilice las autoevaluaciones para cuantificar el grado de obtención de las competencias asociadas a cada capítulo. Y que profundice en sus temas de interés siguiendo para ello las referencias detalladas al final de cada capítulo.

En cuanto a los docentes, se proporcionan los siguientes datos para que les sirvan de referencia. El primer itinerario corresponde casi por completo con la asignatura de Arquitectura de Computadores de 3.^º de Ingeniería Informática (planes antiguos). Esta asignatura se ha impartido en estos planes de estudios en 60 horas durante un cuatrimestre, 40 horas de clases teórico/prácticas y 20 horas de prácticas. El segundo itinerario corresponde, igualmente, con la asignatura de Arquitectura e Ingeniería de Computadores de 4.^º de Ingeniería Informática (planes antiguos). Esta asignatura se ha impartido en 120 horas a lo largo de todo el curso, 75 horas de clases teórico/prácticas y 45 horas de prácticas.

Además del libro de texto, hemos preparado una serie de recursos web disponibles para el apoyo tanto de estudiantes como de profesores. La finalidad es proporcionar documentos, información, herramientas y enlaces útiles para todos los lectores.

El sitio se actualizará según las necesidades y sugerencias de los lectores, aunque en principio contendrá una fe de erratas del libro, las figuras, soluciones a problemas escogidos, ejemplos de prácticas y exámenes y transparencias para las clases.

4 Agradecimientos

No queremos finalizar esta introducción sin manifestar nuestro agradecimiento a los autores de los textos y libros que nos han servido como referencia en nuestra labor docente y en la redacción de este libro: Patterson&Hennessy, Hennessy&Patterson, Stallings, Shen&Lipasti, Ortega&Anguita&Prieto, Parhami, Hwang, Culler&Singh&Gupta, etc.

También a Pearson Educación por la confianza depositada en nosotros. Y por supuesto, a nuestros alumnos de todos estos cursos por la magnífica experiencia que ha supuesto compartir horas de clase y tutorías con ellos.

Los autores
Febrero 2010

1

Conceptos básicos de procesadores

Contenidos

- 1.1. Diseño de un repertorio de instrucciones
- 1.2. Mecanismo completo de ejecución de una instrucción
- 1.3. Evaluación de prestaciones de un procesador
- 1.4. Diseño de procesadores secuenciales
- 1.5. Diseño de procesadores segmentados

En este capítulo se comienza por estudiar el diseño del repertorio de instrucciones de un procesador, ya que desde el punto de vista del diseñador, este repertorio determina muchos de los aspectos de diseño del procesador, que debe ser capaz de ejecutar todas las instrucciones incluidas en este repertorio.

Este análisis de los aspectos de diseño de un repertorio de instrucciones se centra en los repertos de tipo RISC (Reduced Instruction Set Computer), ya que, como se explica en este capítulo, las tendencias de diseño actuales se basan casi completamente en las técnicas de optimización que se aplican a estos procesadores.

Una vez estudiadas las diferentes alternativas para el diseño de un repertorio de instrucciones RISC, se describe el mecanismo completo de ejecución de una instrucción dentro de un procesador de este tipo y se proporcionan las herramientas básicas que permiten determinar el rendimiento de un procesador y así evaluar con técnicas cuantitativas las técnicas de diseño de procesadores que se estudian en el resto del capítulo.

A continuación se presentan las técnicas básicas de diseño de procesadores RISC secuenciales, es decir, de procesadores en los que sólo se puede ejecutar una instrucción en cada momento. Se analizan los dos tipos de temporización que se pueden utilizar en procesadores secuenciales, monociclo cuando cada instrucción tarda un único ciclo de reloj en ejecutarse o multiciclo cuando puede ejecutarse en más de un ciclo.

El resto del capítulo se centra en las técnicas de segmentación de procesadores, que permiten comenzar a explotar el paralelismo a nivel de instrucción y aumentar así el rendimiento de los procesadores secuenciales sin necesidad de incrementar significativamente la complejidad del hardware.

En concreto se estudian los conceptos básicos relacionados con la segmentación (motivación, rendimiento, limitaciones, etc), las técnicas de diseño de procesadores segmentados de única función y multifuncionales, las alternativas existentes para la resolución de riesgos en este tipo de procesadores, y por último, las técnicas de tratamiento de excepciones.

1.1 Diseño de un repertorio de instrucciones

El funcionamiento y diseño de un procesador está completamente determinado por el repertorio de instrucciones máquina o ensamblador que puede ejecutar, por eso este capítulo comienza con el estudio de los conceptos básicos relacionados con este repertorio. Influye directamente en el número de instrucciones que hace falta ejecutar para realizar una determinada tarea (I), por lo que repercute directamente en el tiempo de CPU de la tarea (como se estudiará un poco más adelante en este mismo capítulo). Además, el repertorio influye decisivamente en el diseño del compilador (en la figura 1.1 se resume el proceso de transformación de un código de alto nivel a lenguaje máquina) y está estrechamente relacionado con el diseño de la ruta de datos y la unidad de control del procesador.

Hasta los años 80 la mayor parte de los repertos de instrucciones eran de tipo CISC (Complex Instruction Set Computer), pero a partir de ese momento la tendencia comenzó a cambiar, imponiéndose los repertos de tipo RISC (Reduced Instruction Set Computer). Por ello es este tipo de repertorio el que se va a estudiar a lo largo de este libro, ya que los procesadores actuales o son completamente RISC o, aunque aparezcan como CISC hacia el exterior, en realidad mantienen en su núcleo una arquitectura que incorpora las técnicas típicas de diseño RISC (como es el caso de las arquitecturas x86).

Las arquitecturas de tipo CISC manejan repertos con un gran número de instrucciones complejas, es decir, con gran variedad de tipos de datos, de modos de direccionamiento y de operaciones. Esto permite implementar instrucciones de alto nivel directamente o con un número pequeño de instrucciones ensamblador.

En el caso de las arquitecturas RISC, el repertorio está compuesto por pocas instrucciones y muy básicas. Se trata de repertos simples y ortogonales, en los que los formatos de instrucción son uniformes y se utilizan pocos tipos de datos y de modos de direccionamiento, siempre los más sencillos. Además son fácilmente extensibles y por tanto, permiten diseñar nuevos repertos como modificaciones y/o extensiones de repertos ya existentes.

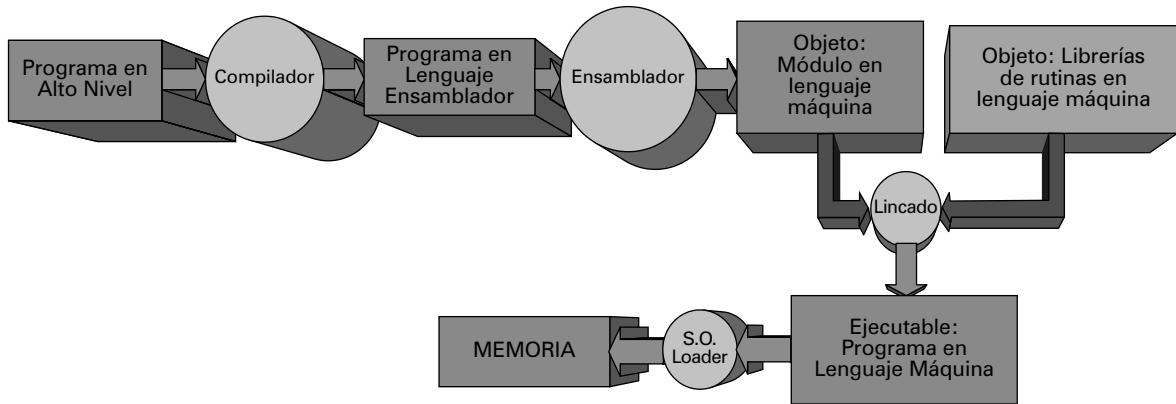


FIGURA 1.1

Proceso de transformación de un código en alto nivel a un código en lenguaje máquina o ensamblador.

También se debe de tener en cuenta que el hardware de este tipo de procesadores es mucho más sencillo de diseñar, más barato y puede trabajar a mayor frecuencia de reloj. Y quizás lo más importante: las técnicas de optimización son mucho más sencillas de implementar, tanto en el propio hardware del procesador como en el compilador. Todo esto compensa la principal desventaja que implican estos repertorios tan sencillos, que siempre necesitan un mayor número de instrucciones que un repertorio CISC para realizar la misma tarea.

Estas son las principales razones por las que se han impuesto las arquitecturas RISC y por ello, es el tipo de repertorio que se estudia en este libro.

1.1.1. Decisiones acerca del tipo de almacenamiento de operandos

Los diferentes repertorios de instrucciones se diferencian principalmente en el tipo de almacenamiento interno que utilizan. Se distinguen distintos tipos (figura 1.2):

- **Pila.** Los operandos son implícitos, siempre en la parte superior de la pila (Top of Stack, TOS). Es decir, no es necesario indicar en las instrucciones dónde se encuentran los operandos.
- **Acumulador.** Uno de los operandos es implícito (el que está en el acumulador) y el otro se debe especificar de manera explícita.
- **Registros de propósito general (GPR o General Purpose Registers).** Los operandos se especifican de manera explícita, pueden ser dos operandos (si uno de los operandos fuente almacena también el resultado de la operación) o tres operandos. Y se puede permitir que alguno o todos los operandos se encuentren en memoria. Por lo tanto se pueden diseñar repertorios:
 - Registro-Registro de 3 operandos, en los que los tres operandos deben estar en registros del procesador antes de operar con ellos. Para llevar los operandos desde la memoria hasta los registros se utilizan instrucciones de carga (load) y para llevar los resultados desde los registros hasta la memoria se utilizan instrucciones de almacenamiento (store).
 - Registro-Memoria de 2 operandos, en los que como mínimo uno de los operandos debe estar en un registro, pero el otro puede estar en memoria.
 - Memoria-Memoria de 2 o 3 operandos, en los que se permite que todos los operandos estén en la memoria.

Casi todas las arquitecturas actuales se basan en registros de propósito general ya que los registros son más rápidos que otros tipos de almacenamiento, sobre todo que la memoria (porque están dentro del

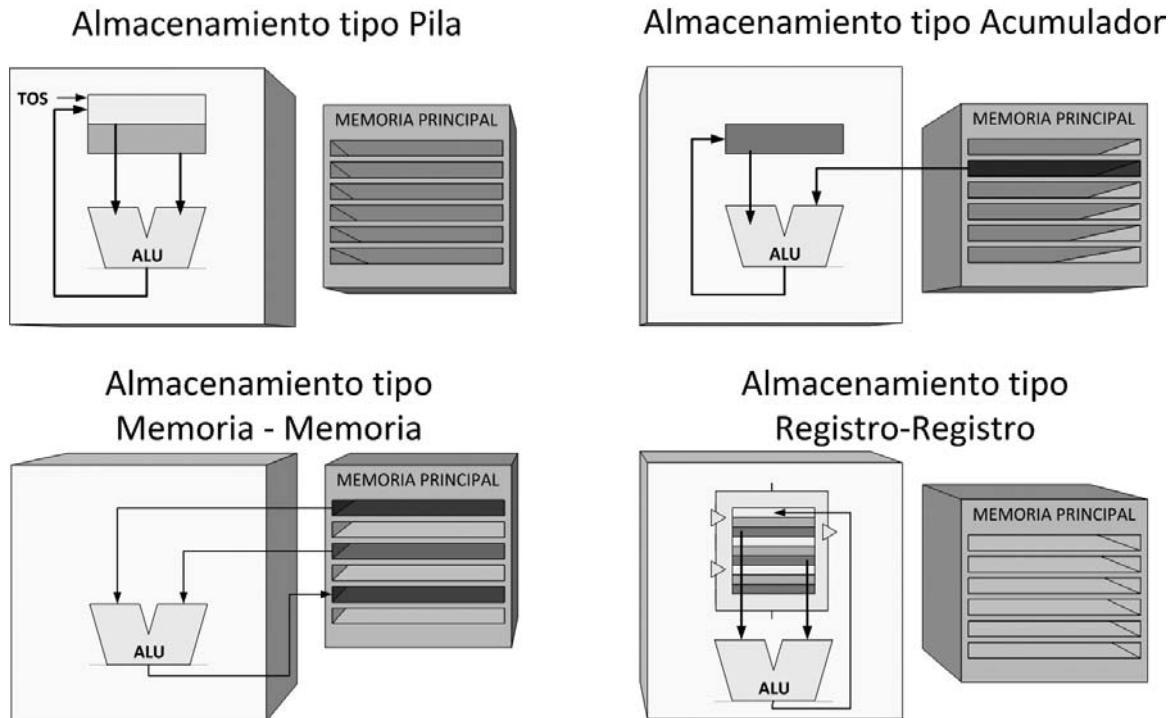


FIGURA 1.2

Alternativas para el almacenamiento de los operandos.

propio procesador), y son utilizados de manera mucho más eficiente por los compiladores. Por ejemplo, se pueden utilizar para almacenar temporalmente variables del programa y reducir así el tráfico con memoria. Por todo esto, es el tipo de almacenamiento que mejor se comporta con repertorios RISC.

De las diferentes alternativas disponibles para diseñar un repertorio GPR, hay que tener en cuenta que en los repertorios registro-registro la codificación de las instrucciones es sencilla puesto que todas tienen la misma longitud y además todas las instrucciones se ejecutan en un número similar de ciclos.

Si los operandos siempre están en registros, para indicar en una instrucción la ubicación de los operandos siempre habrá que especificar el identificador de tres registros y para acceder a ellos, siempre habrá que leer o escribir de registros. Sin embargo hay una desventaja, y es que los programas ocupan más porque se necesitan más instrucciones que en arquitecturas que permitan tener operandos en memoria: las que leen de memoria para traer los operandos a registros dentro del procesador y las que llevan los resultados desde los registros del procesador hasta la memoria. Es decir, se necesitan instrucciones de carga (lectura o load) y de almacenamiento (escritura o store).

En el otro extremo, las arquitecturas memoria-memoria permiten obtener códigos más compactos porque no hacen falta instrucciones de carga y almacenamiento. Además no se malgastan registros en almacenar valores temporales. Sin embargo, la memoria puede llegar a ser un cuello de botella ya que se accede a ella constantemente. Y puede haber grandes diferencias entre la longitud de las instrucciones (se necesitan muchos más bits para especificar la dirección de memoria de un operando que para especificar el identificador de un registro dentro del procesador) y entre su duración (los tiempos de acceso a memoria y a registros son muy diferentes). Estos dos factores complican la codificación del repertorio y hacen que pueda variar mucho el CPI (Ciclos Por Instrucción) del procesador entre unas instrucciones y otras.

Ejemplo 1.1**Alternativas para el almacenamiento de los operandos.**

Vamos a comparar con un ejemplo sencillo las diferencias entre los distintos tipos de almacenamiento de operandos. Supongamos que vamos a evaluar cuatro alternativas:

- a) Pila (implementada de manera que no existe la posibilidad de reutilización de operandos).
- b) Acumulador.
- c) Registro-Registro de 3 operandos con 8 registros de propósito general.
- d) Memoria-Memoria de 3 operandos.

Para realizar la comparación supongamos que se desea realizar una sencilla secuencia de operaciones aritmético-lógicas:

$$\begin{aligned} R &= X \text{ AND } Y \\ Z &= X \text{ OR } Y \\ Y &= R \text{ AND } X \end{aligned}$$

Además necesitamos saber que el código de operación (opcode) de las instrucciones en este repertorio siempre ocupa 1 B, las direcciones de memoria y los operandos ocupan 4 B y que las instrucciones siempre tienen una longitud que es un número entero de bytes para facilitar su almacenamiento y decodificación. Los operandos están inicialmente almacenados en la memoria. Las cuatro secuencias de código para la operación lógica que deseamos realizar serían:

- a) Pila.

	Tráfico con la memoria de instrucciones	Tráfico con la memoria de datos
Apilar X	Opcode+dirección=5B	1 operando=4B
Apilar Y	Opcode+dirección=5B	1 operando=4B
AND	Opcode=1B	—
Desapilar R	Opcode+dirección=5B	1 operando=4B
Apilar X	Opcode+dirección=5B	1 operando=4B
Apilar Y	Opcode+dirección=5B	1 operando=4B
OR	Opcode=1B	—
Desapilar Z	Opcode+dirección=5B	1 operando=4B
Apilar X	Opcode+dirección=5B	1 operando=4B
Apilar R	Opcode+dirección=5B	1 operando=4B
AND	Opcode=1B	—
Desapilar Y	Opcode+dirección=5B	1 operando=4B

Las instrucciones AND y OR no indican cuáles son sus operandos ya que están implícitos: almacenados en la parte superior de la pila.

- b) Acumulador.

	Tráfico con la memoria de instrucciones	Tráfico con la memoria de datos
Load X	Opcode+dirección=5B	1 operando=4B
AND Y	Opcode+dirección=5B	1 operando=4B
Store R	Opcode+dirección=5B	1 operando=4B

	Tráfico con la memoria de instrucciones	Tráfico con la memoria de datos
Load X	Opcode+dirección=5B	1 operando=4B
OR Y	Opcode+dirección=5B	1 operando=4B
Store Z	Opcode+dirección=5B	1 operando=4B
Load R	Opcode+dirección=5B	1 operando=4B
AND X	Opcode+dirección=5B	1 operando=4B
Store Y	Opcode+dirección=5B	1 operando=4B

En este caso las instrucciones de AND y OR tienen que indicar explícitamente uno de sus operandos, el otro está siempre en el acumulador.

c) Registro-Registro de 3 operandos con 8 registros de propósito general.

Si hay 8 registros, se necesitan 3 bits para incluir la etiqueta de un registro en una instrucción.

	Tráfico con la memoria de instrucciones	Tráfico con la memoria de datos
Load R1,X	Opcode+dirección+registro=6B	1 operando=4B
Load R2,Y	Opcode+dirección+registro=6B	1 operando=4B
AND R3,R1,R2	Opcode+3 registros (9 bits que se redondean a 2 B)=3B	—
OR R4,R1,R2	Opcode+3 registros (9 bits que se redondean a 2 B)=3B	—
AND R2,R3,R1	Opcode+3 registros (9 bits que se redondean a 2 B)=3B	—
Store R3,R	Opcode+dirección+registro=6B	1 operando=4B
Store R4,Z	Opcode+dirección+registro=6B	1 operando=4B
Store R2,Y	Opcode+dirección+registro=6B	1 operando=4B

d) Memoria-Memoria de 3 operandos.

	Tráfico con la memoria de instrucciones	Tráfico con la memoria de datos
AND R,X,Y	Opcode+3 direcciones=13B	3 operandos=12B
OR Z,X,Y	Opcode+3 direcciones=13B	3 operandos=12B
AND Y,R,X	Opcode+3 direcciones=13B	3 operandos=12B

Si comparamos el tráfico total con la memoria de instrucciones y con la memoria de datos en cada caso:

	Tráfico total con la memoria de instrucciones	Tráfico total con la memoria de datos
Pila	48B	36B
Acumulador	45B	36B
Registro-Registro	39B	20B
Memoria-Memoria	39B	36B

Podemos observar que el almacenamiento Registro-Registro implica el menor tráfico con la memoria de datos ya que permite la reutilización de los operandos que se cargan en los registros del procesador. Sin embargo, con este tipo de almacenamiento el tráfico con la memoria de instrucciones no se puede reducir excesivamente ya que el código tiene más instrucciones debido a las instrucciones de carga y almacenamiento necesarias para traer los operandos a los registros.

En el otro extremo se encuentra el almacenamiento Memoria-Memoria, que permite conseguir un código muy compacto de sólo 3 instrucciones (aunque de gran tamaño porque incluyen hasta 3 direcciones de memoria), pero no permite ningún tipo de reutilización de operandos, accediendo a memoria para los dos operandos fuente y para el destino en todas las instrucciones. Con un código tan sencillo el efecto de estos continuos accesos a la memoria de datos no es crítico, pero en un código real (con muchas más instrucciones), esto haría que la memoria se convirtiera, casi con toda probabilidad, en el cuello de botella del sistema.

1.1.2. Decisiones acerca de la interpretación de las direcciones de memoria y de los modos de direccionamiento soportados

El diseñador de un repertorio de instrucciones debe determinar cómo se especifican e interpretan las direcciones de memoria.

La mayor parte de los repertorios permiten acceder a 1, 2, 4 u 8 bytes de información. Hay dos posibilidades de ordenación, según dónde se coloque el byte más significativo de información en la palabra de memoria (figura 1.3):

- **Big Endian.** El byte menos significativo de información se coloca en la posición menos significativa de la palabra de memoria (big end).
- **Little Endian.** El byte menos significativo se coloca en la posición más significativa de la palabra de memoria (little end).

En algunos casos se denomina **Middle-Endian** a una arquitectura que es capaz de trabajar con ambas ordenaciones (por ejemplo, el procesador MIPS o el PowerPC, ambos arquitecturas RISC).

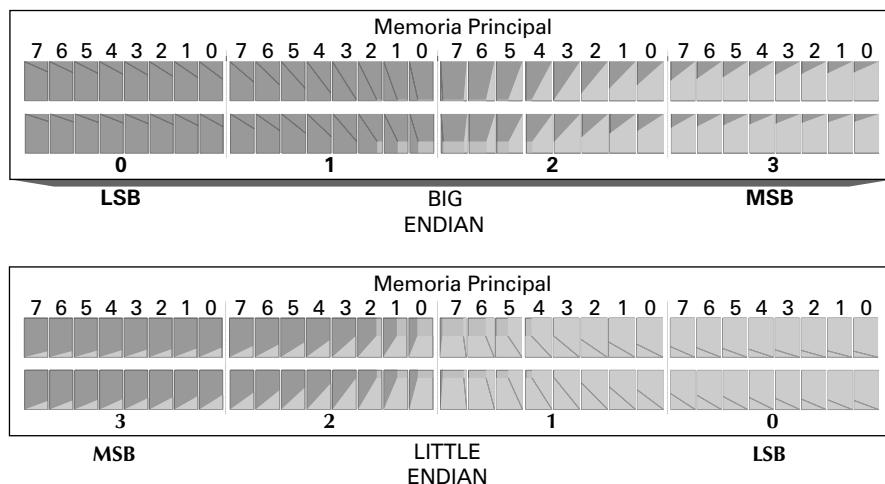
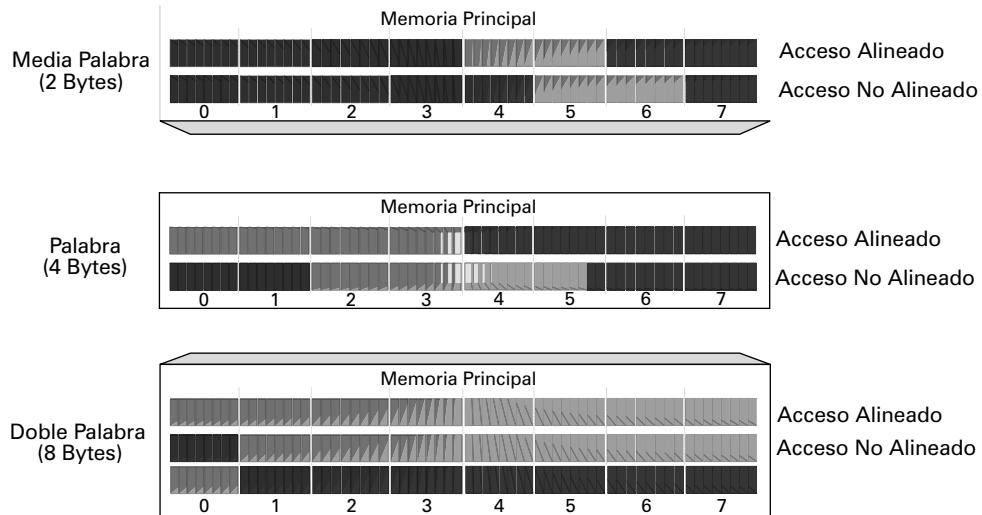


FIGURA 1.3

Diferencia entre Little Endian y Big Endian.

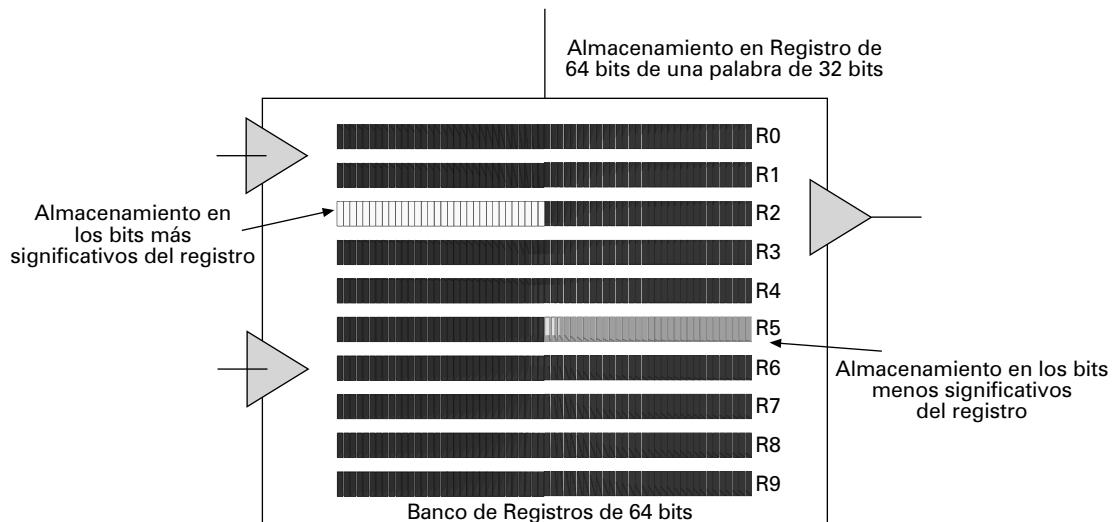
Además hay que especificar cómo deben ser los accesos a información de tamaño mayor que 1 byte. Normalmente estos accesos deben estar alineados (figura 1.4). Un acceso a una información de s bytes en la dirección del byte B está alineado si $B \bmod s = 0$.

**FIGURA 1.4**

Alineación de accesos a memoria.

Estas restricciones de alineamiento se deben a que las memorias, físicamente, están diseñadas para hacer accesos alineados. Si un repertorio de instrucciones permite hacer accesos no alineados, se debe recordar que estos suelen ser muy lentos porque en realidad suponen varios accesos alineados a la memoria.

Por último, también relacionada con el alineamiento, hay una última decisión que se debe tomar al diseñar un repertorio de instrucciones. ¿Si los registros del procesador tienen un determinado tamaño y se lee de memoria una información con un tamaño menor, cómo se alinean los datos leídos dentro del registro? Se debe especificar si en los bits menos significativos o en los más significativos (figura 1.5).

**FIGURA 1.5**

Alineación dentro de los registros del procesador.

En cuanto a los modos de direccionamiento soportados por el repertorio, hay que decidir cómo se puede especificar la localización de un operando dentro de las instrucciones.

Los modos de direccionamiento básicos son los siguientes:

- **Inmediato.** El operando se codifica dentro de la instrucción (figura 1.6).
- **Registro.** Se incluye en la instrucción el identificador del registro del procesador en el que está almacenado el operando (figura 1.7).
- **Directo o absoluto.** Se incluye en la instrucción la dirección de memoria en la que está almacenado el operando (figura 1.8).
- **Indirecto.** Se incluye en la instrucción el identificador del registro del procesador en el que está almacenada la dirección de memoria en la que se encuentra el operando.
- **Indirecto con desplazamiento.** Igual que el anterior, pero se incluye también en la instrucción un operando inmediato denominado desplazamiento que debe sumarse al contenido del registro para obtener la dirección de memoria en la que se encuentra el operando. En muchos casos se considera que el modo de direccionamiento indirecto es un caso particular de éste, en el que el desplazamiento vale 0 (figura 1.9).

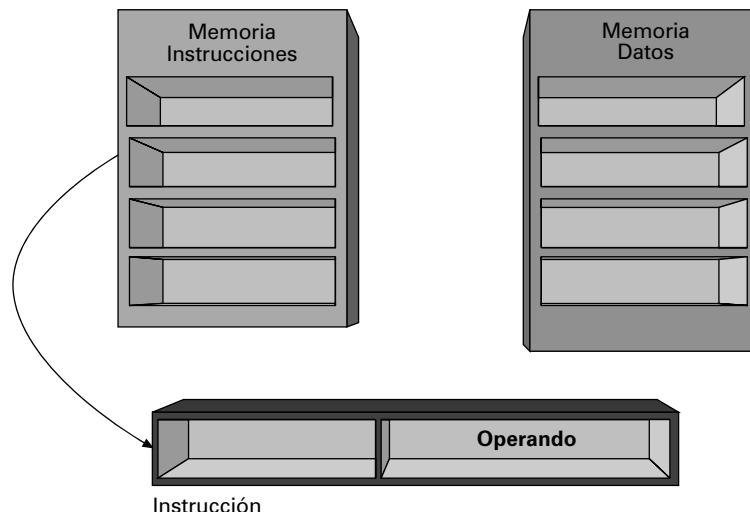


FIGURA 1.6

Modo de direccionamiento inmediato.

El conjunto de modos soportado por el repertorio de instrucciones, si es lo suficientemente rico, puede lograr que se realicen operaciones complejas con pocas instrucciones. Pero los modos muy complejos pueden aumentar la complejidad del hardware e incrementar el CPI de algunas instrucciones.

Prácticamente todos los repertorios RISC incluyen como mínimo los siguientes modos (y no suelen incluir muchos más), que son los más naturales para la mayoría de las instrucciones típicas en estos repertorios:

- Direccionamiento inmediato. Principalmente hay que tomar dos decisiones para diseñar el repertorio. La primera, si todas las instrucciones deben soportar este modo o basta con que lo haga un subconjunto de ellas. La segunda, el rango de valores que puede tomar el operando inmediato, ya que esto influye directamente en la longitud de las instrucciones.
- Direccionamiento indirecto con desplazamiento. Este modo incluye al direccionamiento indirecto, basta con que el desplazamiento tome el valor 0. La decisión más importante consiste, de nuevo, en fijar el rango de valores que puede tomar el desplazamiento.

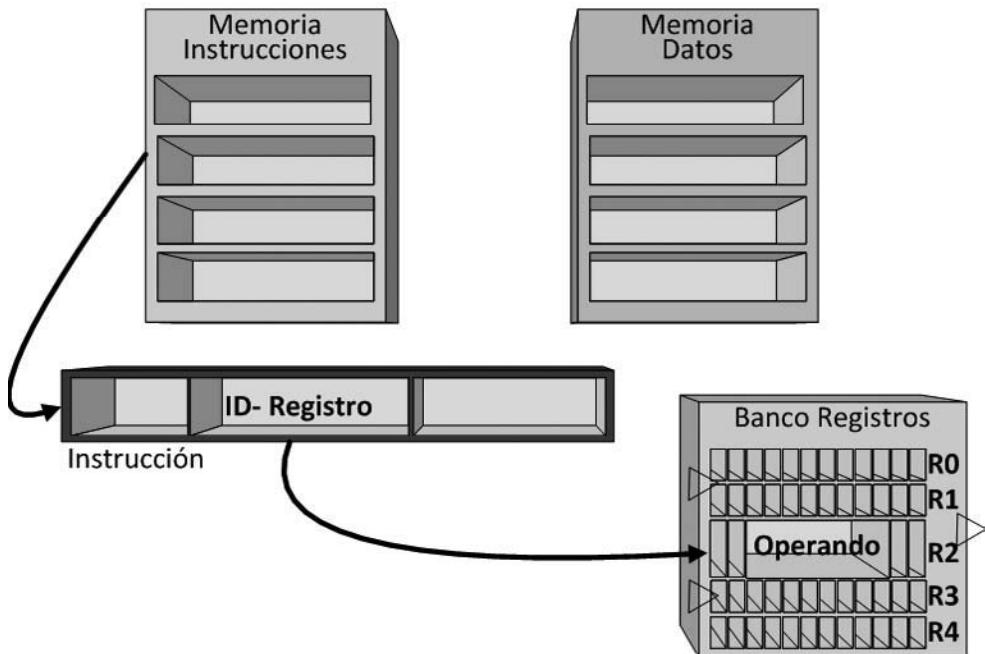


FIGURA 1.7

Modo de direccionamiento de registro.

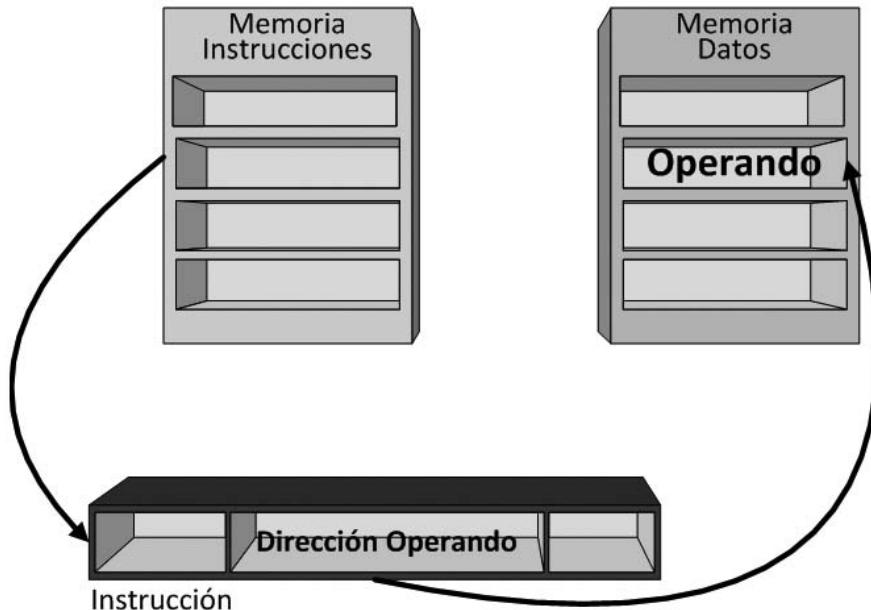


FIGURA 1.8

Modo de direccionamiento directo o absoluto.

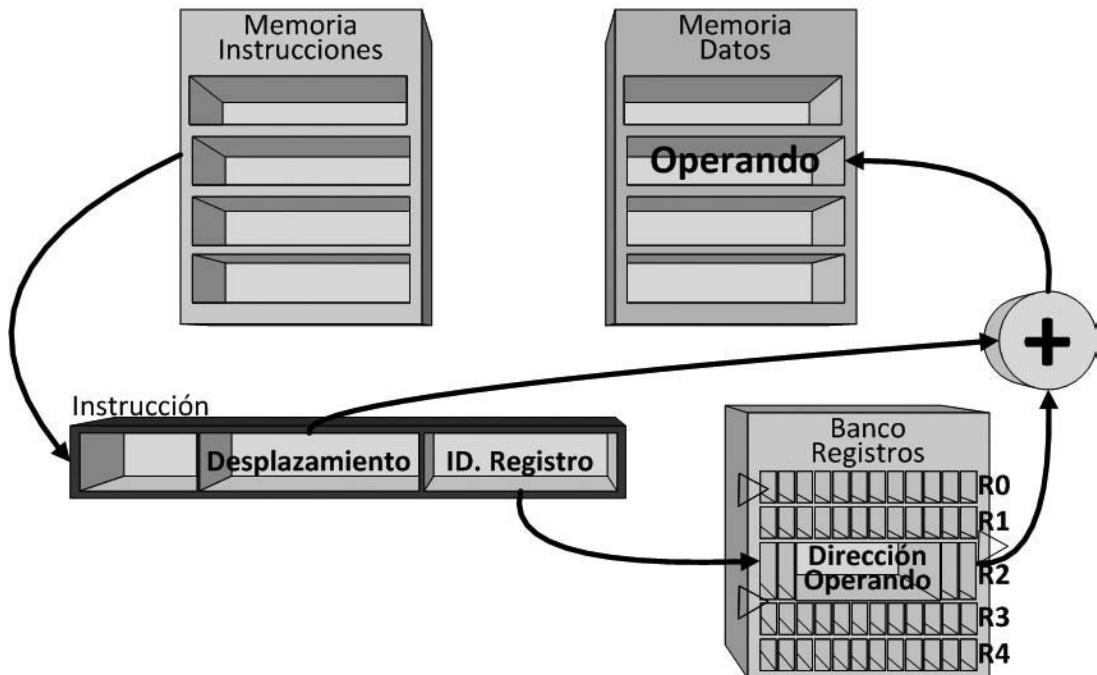


FIGURA 1.9

Modo de direccionamiento indirecto con desplazamiento.

1.1.3. Otras decisiones

Además de definir todo lo relacionado con dónde se encuentran los operandos y cómo especificar dónde se encuentran, existen otras decisiones que también afectan al diseño del repertorio de instrucciones y del procesador:

- **Tipo y tamaño de los operandos.** Se debe decidir qué tipos de datos se soportan (carácter, enteros, coma flotante, etc) y con qué tamaños. Normalmente, al codificar las instrucciones, el código de operación indicará siempre cuáles son los tipos de los operandos implicados en la ejecución de esta instrucción. La otra opción, que sería que cada operando llevara una etiqueta en la instrucción que indicara cuál es su tipo, consumiría demasiados bits y haría que las instrucciones ocuparan demasiado espacio innecesariamente.
- **Conjunto de operaciones soportadas.** También hay que decidir qué tipo de operaciones van a realizar las instrucciones del repertorio. Como se está estudiando el diseño de un repertorio RISC, debe ser un conjunto reducido de operaciones sencillas, normalmente aritmético-lógicas, de acceso a memoria, de control de flujo (saltos) y llamadas al sistema operativo. Además, dependiendo de los tipos de datos soportados, se incluirán instrucciones capaces de operar con caracteres, coma flotante, etc.
- **Tratamiento de las instrucciones de control de flujo.** Las instrucciones de salto suelen implicar decisiones adicionales ya que se trata de instrucciones algo más complejas, que pueden modificar el flujo secuencial de ejecución de un código. Estas decisiones se pueden resumir en las siguientes preguntas si se tienen en cuenta las instrucciones de control de flujo más comunes:
 - Saltos condicionales: ¿Cómo se especifica la condición? ¿Cómo se especifica la dirección destino de salto?
 - Saltos incondicionales: ¿Cómo se especifica la dirección destino de salto?

Existen dos alternativas para la especificación del destino del salto:

- **Direccionamiento relativo al PC.** Cuando se conoce el destino del salto en tiempo de compilación, se puede expresar como un desplazamiento respecto del contador de programa. Como normalmente los destinos de los saltos están cercanos al salto, se utilizan muchos menos bits que si se tuviera que incluir dentro de la instrucción la dirección completa de la instrucción destino. Y además se permite independencia respecto de la posición de memoria en la que se cargue el programa (esto ahorra tiempo en el lincado de los programas). Lo que hay que decidir es cuántos bits se dedican al campo que especifica el desplazamiento dentro de las instrucciones.
- **Direccionamiento indirecto con registro.** Se utiliza en los casos en los que no se conoce el destino del salto en tiempo de compilación o si se conoce, el campo de desplazamiento se queda corto para incluir su valor dentro de la instrucción. Simplemente se incluye en la instrucción el identificador de un registro que contiene la dirección destino de salto.

En cuanto a la condición de los saltos condicionales, siempre se basa en una comparación y existen tres opciones para esta comparación:

- Evaluar los códigos de condición de la ALU.
- Evaluar uno o varios registros.
- Especificar la comparación en la propia instrucción.

Como la primera alternativa es muy inflexible y no siempre se pueden expresar las condiciones recurriendo a los códigos de condición de la ALU, y la última alternativa, a pesar de su flexibilidad suele llevar a instrucciones demasiado complejas para un repertorio RISC, la condición de los saltos suele ir siempre referida a comparaciones que se realizan con uno o varios registros, de manera que en la instrucción sólo se tengan que incluir uno o varios identificadores de registro.

1.1.4. Codificación del repertorio de instrucciones

La codificación binaria de las instrucciones afecta al tamaño del código de los programas y, sobre todo, a la implementación del procesador, que debe decodificar las instrucciones máquina de la manera más eficiente que sea posible.

Para escoger una codificación adecuada se deben conocer las características del repertorio de instrucciones, e influirán todos los factores estudiados hasta el momento: tipo de almacenamiento de operandos, número de registros, modos de direccionamiento soportados, tipos y tamaños de los operandos, etc.

Existen tres opciones para hacer la codificación del repertorio:

- **Longitud variable.** Se soporta cualquier número de operandos y cualquier combinación instrucción/modo de direccionamiento. Los operandos llevan asociados unos especificadores o etiquetas de direccionamiento para indicar el modo de cada uno. Por lo tanto se añaden tantos campos a las instrucciones como sea necesario, añadiendo a estos campos los especificadores y etiquetas que permitan interpretarlos durante la decodificación en el procesador.
- **Longitud fija.** El modo de direccionamiento de los operandos se especifica en el código de operación porque sólo se permiten unas combinaciones determinadas de operaciones y modos. Los campos de la instrucción son siempre los mismos, lo único que puede variar es su interpretación dependiendo del tipo de instrucción.
- **Híbrida.** Se permiten sólo unos determinados formatos de instrucción, que incluyen distinto número de especificadores de modo y de operandos.

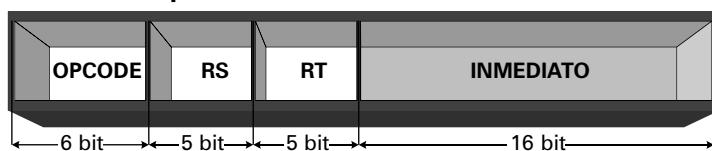
La codificación de longitud variable se suele utilizar cuando se soportan muchos tipos de modos de direccionamiento y operaciones, y distinto número de operandos. Consigue los códigos de menor tamaño

ya que sólo se incluyen los campos de instrucción que se necesitan, pero la decodificación hace que empeore su rendimiento. Por lo tanto es típica en arquitecturas de tipo CISC.

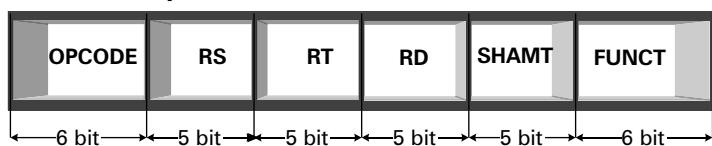
La codificación de longitud fija se utiliza cuando se soportan pocos modos de direccionamiento y operaciones. Como no se particulariza el tamaño de las instrucciones a sus necesidades, a veces se quedan campos en blanco o infroutilizados, y el código tiene un tamaño mayor del que podría tener. Pero se consigue un gran rendimiento en la decodificación, que es mucho más sencilla, por lo que es la típica en arquitecturas RISC. Sin embargo la híbrida suele ser la más utilizada en este tipo de arquitecturas, ya que consigue combinar las ventajas de los dos tipos de codificación.

En cualquiera de los tres casos interesa que la longitud de las instrucciones sea siempre un número entero de bytes, porque así es mucho más sencilla la decodificación en el procesador y el almacenamiento en la memoria.

Instrucción tipo I



Instrucción tipo R



Instrucción tipo J

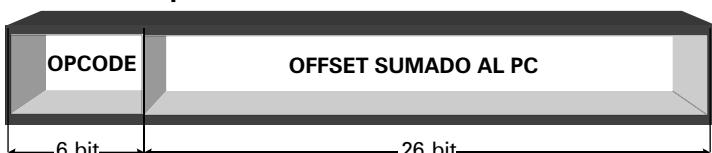


FIGURA 1.10

Ejemplo de codificación híbrida: tipos de instrucción para el MIPS64.

CASO PRÁCTICO 1.1. Repertorio de instrucciones del procesador MIPS64

El MIPS64 es una arquitectura basada en registros de propósito general tipo carga/almacenamiento (registro-registro), con direcciones de memoria de 64 bits.

Este procesador tiene un banco de registros de 32 registros de propósito general de 64 bits (R0,...,R31). El registro R0 siempre almacena un 0 y no se puede variar su contenido así que en realidad es como si sólo hubiera 31 registros. También tiene un banco de 32 registros para coma flotante (F0,...,F31) que pueden almacenar números en precisión simple o doble.

Soporta tipos enteros de 8, 16, 32 y 64 bits y tipos en coma flotante de 32 y 64 bits. También permite trabajar con números en coma flotante de precisión simple empaquetados de dos en dos.

Los modos de direccionamiento que se soportan son:

- Inmediato: Con un campo para el operando de 16 bits dentro de la instrucción.

- Indirecto con desplazamiento: Con un campo para el desplazamiento de 16 bits. Si se pone un cero en este campo tenemos el modo indirecto puro. Y si se escoge el registro R0 como registro base, este modo nos permite además un direccionamiento absoluto.

Así que en la práctica es como si tuviéramos cuatro modos de direccionamiento diferentes.

Como sólo hay dos tipos de direccionamiento (en cuanto al tipo de instrucción), el modo se codifica junto con el opcode. Todas las instrucciones son de 32 bits (longitud fija) con un opcode de 6 bits.

Se escoge una codificación híbrida en la que se incluyen tres formatos diferentes de instrucción (figura 1.10):

- Tipo I (Inmediato).
- Tipo R (Registro).
- Tipo J (Jump).

Las instrucciones de Tipo I codifican las siguientes operaciones:

- Load/store
 - RS (registro fuente): Registro base para el acceso a memoria.
 - RT (registro destino): Registro para los datos.
 - Inmediato: Desplazamiento para el cálculo de la dirección de memoria a la que hay que acceder.
- Operaciones aritmético-lógicas con direccionamiento inmediato
 - RS (registro fuente): Registro que contiene el operando 1.
 - RT (registro destino): Registro destino de la operación.
 - Inmediato: Operando 2 (directamente su valor).
- Saltos condicionales/incondicionales
 - RS (registro fuente): Registro de condición (para la comparación)/Registro que contiene la dirección destino del salto.
 - RT (registro destino): Registro de condición (para la comparación)/No se utiliza.
 - Inmediato: Desplazamiento respecto del PC/0.

En cuanto a las instrucciones de Tipo R, se utilizan para codificar las operaciones aritmético-lógicas registro-registro:

- RS: Registro que contiene el operando 1.
- RT: Registro que contiene el operando 2.
- RD: Registro destino.
- Shamf (shift amount): Indica el desplazamiento en las instrucciones shift.
- Funct: Junto con el opcode indica el tipo de operación que se debe realizar.

Por último, las instrucciones de Tipo J codifican las instrucciones de salto incondicional y de retorno de procedimiento que utilizan direccionamiento con desplazamiento relativo al PC (Offset de 26 bits).

1.1.5. Mejoras y optimizaciones del repertorio de instrucciones

Casi todas las mejoras de los repertorios de instrucciones RISC se aprovechan de la extensibilidad que presentan estos repertorios y que ya se ha mencionado en este capítulo.

Por ello, a lo largo del tiempo se han ido añadiendo instrucciones a estos repertorios que pueden mejorar el rendimiento de los procesadores de propósito general en la ejecución de aplicaciones y códigos concretos: instrucciones que incorporan predicados y ahorran así bifurcaciones (esto se entenderá mejor al estudiar el capítulo 3), instrucciones que combinan dos o más operaciones que suelen aparecer siempre agrupadas, etc.

El ejemplo típico de estas extensiones que mejoran la ejecución de cierto tipo de aplicaciones son las extensiones multimedia. En este tipo de aplicación es habitual realizar operaciones repetitivas sobre dife-

rentes operandos que no llegan a los 32 bits de un típico tipo entero (un píxel son 8 bits, una muestra de audio son 16 bits). Esto hace que se desaprovechen recursos durante su ejecución, por lo que se incorporan al repertorio instrucciones capaces de operar con vectores (instrucciones Single Instruction Multiple Data o SIMD) que agrupen en los registros típicos de 64 bits, 8 operandos de 8 bits, 4 operandos de 16 bits o 2 operandos de 32 bits.

Para cada tipo de vector se suelen incluir en esta extensión instrucciones aritmético-lógicas (suma, resta, multiplicación, comparación, desplazamiento, and, or y xor, típicamente), de acceso a memoria (load y store) y de empaquetamiento, reordenamiento y copia de datos.

Estos repertorios surgieron para trabajar con vectores de números enteros pero luego se han realizado nuevas extensiones para poder agrupar también operandos en coma flotante en los registros de 128 bits.

Algunos ejemplos de extensiones multimedia en repertorios RISC son las MIPS Digital Media eXtension (MDMX) o las Multimedia Acceleration eXtensions (MAX). Pero quizás las más conocidas sean las extensiones de las arquitecturas x86, primero fueron las MMX, y después las diferentes versiones de las 3Dnow! y las SSE.

1.2 Mecanismo completo de ejecución de una instrucción

Este libro se centra en el estudio de diferentes arquitecturas MIPS simplificadas con fines pedagógicos (denominadas nanoMIPS) para comprender el funcionamiento de los procesadores actuales y las técnicas de aumento de prestaciones que pueden ayudar a mejorar su rendimiento. Es decir, se tomará como punto de partida un procesador RISC cuyo diseño sirve de referencia para la mayor parte de las arquitecturas actuales. La comprensión del funcionamiento y de las técnicas de optimización de este procesador permitirá comprender los mismos aspectos para otros procesadores. Algunos ejemplos de procesadores tremadamente parecidos al MIPS son el PowerPC de IBM, el SPARC de Sun Microsystems o el Digital Alpha. Hay que recordar que en el diseño de procesadores RISC prácticamente se ha llegado a un estándar, por lo que las diferencias entre unas arquitecturas y otras suelen ser muy pequeñas.

Ya se ha estudiado en el caso práctico 1.1 que el procesador MIPS64, al igual que la mayor parte de los procesadores actuales, es una arquitectura registro-registro de 3 operandos. Esto significa que el repertorio de instrucciones está diseñado para que los tres operandos que utiliza como máximo cualquier instrucción (los dos fuente y uno destino), estén en registros dentro del procesador, ninguno puede estar ubicado en la memoria.

Sabiendo esto, se puede resumir la ejecución de una instrucción completa en un procesador de este tipo en cinco etapas, cada instrucción pasará por aquéllas que necesite para completar su ejecución:

- **Fetch (F).** En esta primera etapa se busca en la memoria la instrucción que está almacenada en la dirección que indica el contador de programa (Program Counter o PC). Normalmente, ya se deja preparado el contador de programa (sumando o restando una cantidad fija que depende de la arquitectura y del repertorio de instrucciones) para buscar la siguiente instrucción.
- **Decode (D).** A continuación, se decodifica la instrucción separando sus diferentes campos. El código de operación de la instrucción indica qué tipo de instrucción es, y por tanto, qué tipo de operación se debe realizar en la ruta de datos. Si es necesario, se leen 1 o 2 operandos de los registros del procesador.
- **Execution (X).** En esta etapa se ejecuta la operación que indicaba el opcode, normalmente, utilizando para ello algún tipo de ALU o de unidad funcional aritmético-lógica.
- **Memory Access (M).** Si es necesario acceder a memoria para leer o escribir, el acceso se realiza en esta fase.
- **Writeback (W).** Por último, si es necesario volcar algún resultado a un registro, se realiza esta escritura.

CASO PRÁCTICO 1.2.**Mecanismo completo de ejecución de las instrucciones del repertorio del MIPS64**

Veamos a continuación las tareas que los diferentes tipos de instrucción del MIPS64 realizan en cada una de las etapas descritas para la ejecución de instrucción. Esta división de trabajo corresponde con su implementación monociclo.

Instrucciones de Tipo I:

- Load/store

F	Búsqueda de la instrucción en la dirección de la memoria de instrucciones que indica el PC. Preparación del PC para la siguiente instrucción.	MI[[PC]] PC <- PC+4 (porque las instrucciones son de 32 bits, 4 bytes)
D	Decodificación de la instrucción. Lectura del registro RS y del registro RD si la instrucción es un Store. Extensión de signo para Inmediato (que es un entero de 16 bits).	[RS] ext(Inmediato) Y si es un Store: [RD]
X	La ALU realiza la suma del contenido del registro base para el acceso a memoria y el desplazamiento.	[RS]+ext(Inmediato)
M	Acceso a la dirección de memoria que se ha obtenido en la etapa anterior. Si es una instrucción Load, para leer el contenido de esta posición. Si es una instrucción de Store, para escribir el contenido del registro RD en esta posición.	Si es un Load: MD[[RS]+ext(Inmediato)] Si es un Store: MD[[RS]+ext(Inmediato)]<-[RD]
W	En el caso de las instrucciones de Load, se escribe lo que se ha leído de memoria en el registro RD.	Si es un Load: [RD]<- MD[[RS]+ext(Inmediato)]

- Operaciones aritmético-lógicas con direccionamiento inmediato

F	Búsqueda de la instrucción en la dirección de la memoria de instrucciones que indica el PC. Preparación del PC para la siguiente instrucción.	MI[[PC]] PC <- PC+4
D	Decodificación de la instrucción. Lectura del registro RS. Extensión de signo para Inmediato (que es un entero de 16 bits).	[RS] ext(Inmediato)
X	La ALU realiza la operación que indique el opcode de la instrucción.	[RS] OP ext(Inmediato)
M	—	—
W	Se escribe el resultado de la operación en el registro RD.	[RD]<- [RS] OP ext(Inmediato)]

- Saltos condicionales (condición evaluada sobre el contenido de dos registros)

F	Búsqueda de la instrucción en la dirección de la memoria de instrucciones que indica el PC. Preparación del PC para la siguiente instrucción.	MI[[PC]] PC <- PC+4
D	Decodificación de la instrucción. Lectura de los registros RS y RD. Extensión de signo para Inmediato (que es un entero de 16 bits).	[RS] [RD] ext(Inmediato)

I. CONCEPTOS BÁSICOS DE PROCESADORES

X	Se suma al PC el Inmediato para obtener la dirección destino del salto. Además se evalúa la condición del salto sobre el contenido de RS y RD. Si la evaluación de la condición ha tenido un resultado positivo, se carga el PC con el valor obtenido para la dirección destino del salto.	[PC]+ext(Inmediato) cond ([RS],[RD]) Si cond = TRUE PC<-[PC]+ext(Inmediato)
M	—	
W	—	
• Saltos incondicionales (direcciónamiento indirecto con registro)		
F	Búsqueda de la instrucción en la dirección de la memoria de instrucciones que indica el PC. Preparación del PC para la siguiente instrucción.	MI[[PC]] PC <- PC+4
D	Decodificación de la instrucción. Lectura del registro RS.	[RS]
X	Se carga el PC con el valor que se ha leído del registro RS.	PC<-[RS]
M	—	
W	—	

Instrucciones de Tipo R:

F	Búsqueda de la instrucción en la dirección de la memoria de instrucciones que indica el PC. Preparación del PC para la siguiente instrucción.	MI[[PC]] PC <- PC+4
D	Decodificación de la instrucción. Lectura de los registros RS y RT.	[RS] [RT]
X	La ALU realiza la operación indicada por la combinación del opcode de la instrucción y del campo Funct.	[RS] OP [RT]
M	—	
W	Se escribe el resultado de la operación en el registro RD.	[RD]<- [RS] OP [RT]

Instrucciones de Tipo J (saltos incondicionales con direcciónamiento relativo al PC):

F	Búsqueda de la instrucción en la dirección de la memoria de instrucciones que indica el PC. Preparación del PC para la siguiente instrucción.	MI[[PC]] PC <- PC+4
D	Decodificación de la instrucción. Lectura del registro RS. Extensión de signo para Offset (que es un entero de 16 bits).	[RS] ext(Offset)
X	Se suma al PC el Offset para obtener la dirección destino del salto. Se carga el PC con el valor obtenido para la dirección destino del salto.	[PC]+ext(Offset) PC<-[PC]+ext(Offset)
M	—	
W	—	

1.3 Evaluación de prestaciones de un procesador

La ecuación más general que permite cuantificar el rendimiento de un procesador suele denominarse ecuación de prestaciones y utiliza como métrica de rendimiento del procesador el tiempo que un código tarda en ejecutarse en él (tiempo de CPU).

Si T es el periodo de reloj del procesador, CPI son los ciclos que en media tarda una instrucción en ejecutarse e I es el número de instrucciones ensamblador que componen el código, el tiempo que este código tarda en ejecutarse en el procesador es:

$$t_{CPU} = I \cdot CPI \cdot T$$

Observando esta ecuación de prestaciones, queda claro que para mejorar el diseño de un procesador siempre hay que intentar reducir uno de los tres factores de los que depende el tiempo que tarda en ejecutarse un código.

El número de instrucciones que componen el código depende casi exclusivamente del repertorio de instrucciones del procesador y del compilador, ya que en prácticamente todos los casos el desarrollador programará en un lenguaje de alto nivel y será el compilador el que traduzca las instrucciones programadas en C, Visual Basic o Java, por poner algunos ejemplos, al repertorio a bajo nivel del procesador.

Los antiguos repertorios CISC conseguían un menor número de instrucciones que los repertorios RISC actuales, pero por otro lado, al ser las instrucciones mucho más complejas, necesitaban más ciclos para ejecutarse y el CPI era bastante mayor.

Si se fija el número de instrucciones, por ser un factor que depende del repertorio de instrucciones y del compilador, tanto el CPI como el periodo de reloj dependen del diseño de la arquitectura del procesador. Y el periodo de reloj (inversamente relacionado con la frecuencia de funcionamiento) también guarda una relación muy estrecha con la tecnología de fabricación del procesador. A lo largo de este capítulo y del capítulo 3, se estudiarán técnicas que permiten reducir tanto el CPI como el periodo del procesador para mejorar los tiempos de CPU de los códigos. Pero por norma general, cuando se consigue reducir uno de los dos factores, suele ser a costa de incrementar el otro.

Antes de continuar, sólo una última aclaración. El tiempo que un código tarda en ejecutarse en un procesador no suele coincidir hoy en día con lo que solemos denominar tiempo de ejecución o tiempo de respuesta. Este tiempo es el que el usuario percibe que transcurre desde que la aplicación se lanza hasta que devuelve resultados. Teniendo en cuenta que el sistema operativo consume tiempo en realizar sus tareas y que la mayor parte de los sistemas operativos actuales son multitarea, casi siempre se cumple $t_{respuesta} > t_{CPU}$.

Como este capítulo se centra en la evaluación de prestaciones de un procesador, desde el punto de vista de su arquitectura, y sin tener en cuenta al resto del sistema o al sistema operativo, el tiempo de procesador es el más adecuado para cuantificar el rendimiento en nuestro caso.

Ejemplo 1.2

Utilización de la ecuación de prestaciones del procesador.

Un procesador que funciona a una frecuencia de 2.1 GHz ejecuta un código compuesto por 1200 instrucciones ensamblador.

Se ha estudiado el repertorio de instrucciones de este procesador y se sabe que en media el 20% de las instrucciones ejecutadas son saltos, el 50% son instrucciones aritmético-lógicas y el 30% restante son instrucciones de acceso a memoria. Los saltos tardan 3 ciclos en ejecutarse, las instrucciones aritmético-lógicas tardan 4 ciclos y las instrucciones de acceso a memoria tardan 5 ciclos.

Con toda esta información podemos saber cuánto tiempo tarda en ejecutarse este código en el procesador:

$$T = \frac{1}{f} = \frac{1}{2.1 \cdot 10^9} = 0.48 \text{ ns}$$

$$CPI = \sum f_{\text{instrucción}} \cdot CPI_{\text{instrucción}} = 0.2 \cdot 3 + 0.5 \cdot 4 + 0.3 \cdot 5 = 4.1$$

$$t_{\text{CPU}} = I \cdot CPI \cdot T = 1200 \cdot 4.1 \cdot 0.48 = 2361.6 \text{ ns}$$

Otra ecuación muy relacionada con la evaluación de prestaciones del procesador es la que permite calcular el speedup (ganancia) conseguido al realizar una determinada mejora en el diseño del procesador:

$$S = \frac{t_{\text{sin mejora}}}{t_{\text{con mejora}}}$$

Obviamente si la mejora lo es de verdad, el speedup tendrá siempre un valor mayor que 1 (se ha conseguido gracias a la mejora reducir el tiempo de CPU de un determinado código).

Los tiempos con o sin mejora pueden medirse directamente, pero también pueden calcularse con la ecuación de prestaciones, por lo que:

$$S = \frac{(I \cdot CPI \cdot T)_{\text{sin mejora}}}{(I \cdot CPI \cdot T)_{\text{con mejora}}}$$

1.4 Diseño de procesadores secuenciales

Actualmente el diseño de procesadores sigue una metodología prácticamente universal gracias al estándar de facto en el que se han convertido las arquitecturas RISC.

En este diseño siempre se pueden distinguir dos grandes módulos dentro del procesador: la ruta de datos y la unidad de control. No se debe olvidar que el procesador no es más que un circuito digital, por lo que la ruta de datos es la parte combinacional del procesador, encargada de operar con datos y producir resultados. Por otro lado, la unidad de control es la responsable del estado del procesador y de gestionar las operaciones que se realizan en la ruta de datos. Como se verá en las próximas secciones, puede ser un circuito combinacional o secuencial.

Un procesador secuencial es aquel que hasta que no termina de ejecutar una instrucción no comienza a ejecutar la siguiente. Es decir, en el procesador sólo se encontrará una instrucción ejecutándose cada vez.

Existen dos tipos de procesadores secuenciales dependiendo del método de temporización escogido durante su diseño:

- **Procesador monociclo.** Cada instrucción debe completarse en un único ciclo de reloj.
- **Procesador multiciclo.** Cada instrucción puede tardar más de un ciclo en ejecutarse, tantos como sea necesario.

Obviamente, en un procesador monociclo el CPI es siempre igual a 1. Pero el problema está en que la duración del ciclo de reloj del procesador vendrá fijada por la instrucción que más tarde en ejecutarse, ya que le tiene que dar tiempo a finalizar en un único ciclo de reloj. Por el contrario, un procesador multiciclo tendrá siempre CPI mayor 1, pero el periodo de reloj será menor que en el caso del procesador monociclo.

Normalmente, la ejecución de una instrucción está dividida en etapas (ya se ha estudiado una división habitual del trabajo en la sección 1.2) y se fija el periodo de reloj para que cada una de estas etapas se complete en un ciclo. Por lo tanto en el caso del procesador multiciclo es la duración de la etapa más larga la que fija el periodo de reloj.

Supongamos que se diseña un procesador similar al MIPS64, denominado nanoMIPS, con un repertorio de instrucciones muy sencillo en el que sólo están presentes las siguientes instrucciones para números enteros:

- Acceso a memoria: LW y SW (tipo I).
- Operaciones aritmético-lógicas: ADD, SUB, AND, OR y SLT (tipo R).
- Control de flujo: BEQ (tipo I).

Este repertorio tan sencillo sólo permite la ejecución de tareas también muy sencillas (tabla 1.1), pero es suficiente para comprender en profundidad el funcionamiento y diseño de las rutas de datos y unidades de control para la alternativa monociclo y para la multiciclo.

TABLA 1.1

Valores del opcode y del campo Funct para las instrucciones del repertorio del nanoMIPS.

Instrucción	Pseudocódigo	Opcode	Funct
LW	LW RT,inmediato(RS)	100011	—
SW	SW RT,inmediato(RS)	101011	—
ADD	ADD RD,RS,RT	000000	100000
SUB	SUB RD,RS,RT	000000	100010
AND	AND RD,RS,RT	000000	100100
OR	OR RD,RS,RT	000000	100101
SLT	SLT RD,RS,RT	000000	101010
BEQ	BEQ RS,RT,destino	000100	—

En ambos casos se siguen los mismos pasos para realizar el diseño del procesador:

1. Analizar el repertorio de instrucciones que se va a ejecutar en el procesador.
2. Establecer la metodología de temporización (monociclo o multiciclo).
3. Seleccionar el conjunto de módulos que componen la ruta de datos teniendo en cuenta ambos factores: el repertorio de instrucciones y la metodología de temporización.
4. Ensamblar la ruta de datos conectando los módulos escogidos e identificando los puntos de control.
5. Determinar los valores de los puntos de control para cada instrucción del repertorio.
6. Diseñar la unidad de control.
7. Optimizar el diseño obtenido ajustándose a las especificaciones de área y coste, a la funcionalidad del procesador, a las limitaciones de calor y potencia, etc.

Este capítulo se centra de momento en los seis primeros pasos de esta metodología, y en la optimización básica que se utiliza siempre en la actualidad, la segmentación del procesador, dejando el resto de técnicas de optimización para el capítulo 3 de este libro.

1.4.1. Procesador monociclo

Teniendo en cuenta el repertorio de instrucciones que debe ejecutarse y la temporización escogida para el procesador en este caso, la ruta de datos del nanoMIPS monociclo debe incluir los siguientes módulos:

- Contador de programa. Es el registro que almacena la dirección de memoria en la que se debe buscar la siguiente instrucción que se va a ejecutar en el procesador.

- Memorias de instrucciones y de datos. Estas dos memorias tienen que estar separadas porque una instrucción no puede acceder al mismo recurso más de una vez si la implementación es monociclo. De momento se supone que son memorias ideales en las que siempre se encuentra la información que se busca y que su tiempo de acceso es menor que el tiempo de ciclo del procesador. Además, son direccionables por bytes pero capaces de devolver 4 bytes en cada acceso.
- Banco de registros de datos visibles para el programador. Los 32 registros necesarios para almacenar números enteros se agrupan en un banco de registros porque en las instrucciones de tipo R es necesario acceder a dos registros simultáneamente. Este banco tiene por tanto dos salidas de datos de 32 bits, una entrada de datos de 32 bits y tres entradas de 5 bits para la identificación de los registros.
- Dos sumadores. El primero se utiliza para sumar 4 (las instrucciones son de 32 bits, es decir, entre una instrucción y la siguiente siempre hay 4 bytes) al PC y dejarlo preparado para la búsqueda de la siguiente instrucción. El segundo se utiliza para sumar el desplazamiento relativo al PC en las instrucciones de salto y dejar la ALU para la evaluación de la condición.
- Una ALU de enteros capaz de realizar las operaciones incluidas en el repertorio de instrucciones. Para la comparación que se realiza en el salto condicional BEQ, basta con hacer una resta y ver si el resultado obtenido es igual a 0.
- Un extensor de signo para extender el signo de los operandos inmediatos, que son de 16 bits.
- Un desplazador a la izquierda de 2 posiciones. Para recuperar los dos ceros al final del desplazamiento relativo de los saltos. Como las instrucciones siempre son de 32 bits, el desplazamiento siempre es un múltiplo de 4. Esto significa que siempre termina por 00 y que no es necesario malgastar dos bits del campo de instrucción que permite almacenar el desplazamiento de salto en almacenar estos dos ceros. Por eso el desplazamiento se almacena sin ellos en la instrucción (por lo que pueden llegar a almacenarse desplazamientos de hasta $2^{18}-1$ a pesar de tener sólo 16 bits) y luego se recuperan antes de sumarlo con el PC con este desplazador a la izquierda.

Se puede observar en la figura 1.11 que no es necesario incluir un hardware específico para la decodificación de instrucciones ya que al tratarse de un repertorio tan sencillo con una codificación de longitud fija (aunque sea híbrida), basta con separar los diferentes campos de la instrucción y utilizar su contenido directamente.

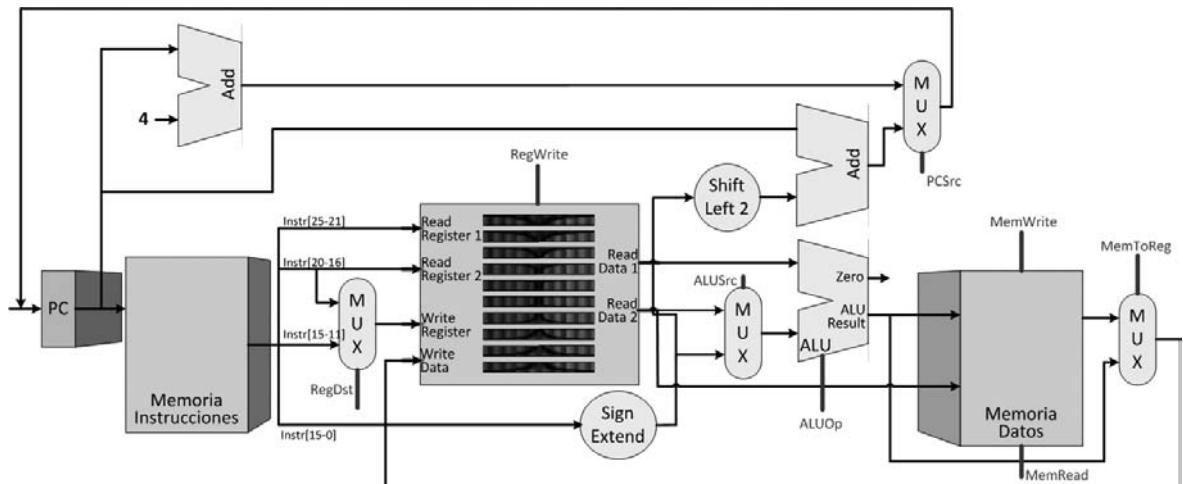


FIGURA 1.11

Ruta de datos del nanoMIPS monociclo.

Ejemplo 1.3**Cálculo de la frecuencia máxima de funcionamiento de un procesador nanoMIPS monociclo.**

Con la ruta de datos del nanoMIPS monociclo diseñada y las siguientes latencias para los módulos hardware que la componen, podemos calcular la frecuencia máxima de funcionamiento del procesador teniendo en cuenta que el periodo de reloj estará limitado por la instrucción que más tarde en ejecutarse:

- Lectura de la memoria de instrucciones: 0.3 ns
- Lectura de la memoria de datos: 0.35 ns
- Escritura en la memoria de datos: 0.5 ns
- Lectura y escritura en el banco de registros (dos puertos de lectura y uno de escritura): 0.05 ns
- Operación aritmético-lógica con la ALU: 0.25 ns
- Suma para preparar el PC siguiente: 0.1 ns
- Suma del PC y el desplazamiento del salto: 0.1 ns

Con estos datos podemos calcular la duración de cada una de las instrucciones, vamos a suponer que no se utilizan módulos de la ruta de datos en paralelo:

- Load: latencia = lectura de la memoria de instrucciones + suma para preparar el PC siguiente + lectura del banco de registros + operación aritmético-lógica con la ALU + lectura de la memoria de datos + escritura en el banco de registros = $0.3+0.1+0.05+0.25+0.35+0.05=1.1 \text{ ns}$
- Store: latencia = lectura de la memoria de instrucciones + suma para preparar el PC siguiente + lectura del banco de registros + operación aritmético-lógica con la ALU + escritura en la memoria de datos = $0.3+0.1+0.05+0.25+0.5=1.2 \text{ ns}$
- Aritmético-lógicas: latencia = lectura de la memoria de instrucciones + suma para preparar el PC siguiente + lectura del banco de registros + operación aritmético-lógica con la ALU + escritura en el banco de registros = $0.3+0.1+0.05+0.25+0.05=0.75 \text{ ns}$
- BEQ: latencia = lectura de la memoria de instrucciones + suma para preparar el PC siguiente + lectura del banco de registros + operación aritmético-lógica con la ALU + suma del PC y el desplazamiento del salto = $0.3+0.1+0.05+0.25+0.1=0.8 \text{ ns}$

Por lo tanto, la instrucción que limita el ciclo de reloj es el Store, y como su latencia es de 1.2 ns, tenemos:

$$T_{\min} = 1.2 \text{ ns} \rightarrow f_{\max} = \frac{1}{1.2 \cdot 10^{-9}} = 0.8 \text{ GHz}$$

En la figura 1.11 se han señalado los puntos de control identificados para los módulos hardware que se han incluido en la ruta de datos. Como el control de la ALU es algo más complejo (dependerá del opcode de la instrucción, pero también del campo Funct en las instrucciones de tipo R), se introduce un hardware específico para realizar este control, un poco más adelante se analizará en detalle.

Una vez diseñada la ruta de datos e identificados los puntos de control, es necesario determinar los valores que deben tomar estos puntos para cada instrucción del repertorio (tablas 1.2 y 1.3).

TABLA 1.2

Valores para los puntos de control del banco de registros y la ALU en la ruta de datos del nanoMIPS monociclo.

Instrucción	RegDst	RegWrite	ALUSrc	ALUOp
Load	0	1	1	00
Store	—	0	1	00
Aritmético-lógicas	1	1	0	10
BEQ	—	0	0	01

Tabla 1.3

Valores para los puntos de control de memoria y del PC en la ruta de datos del nanoMIPS monociclo.

Instrucción	MemRead	MemWrite	PCSrc	MemtoReg
Load	1	0	0	0
Store	0	1	0	—
Aritmético-lógicas	0	0	0	1
BEQ	0	0	0/1 (según evaluación de la condición del salto)	—

Para un diseño de este tipo lo mejor es utilizar dos unidades de control diferentes (figura 1.12). La primera es la unidad de control global, encargada de decodificar el campo de código de operación y de configurar globalmente la ruta de datos. Además se añade un segundo control local para la ALU, que decodifica el campo Funct en las instrucciones de tipo R y selecciona la operación concreta que debe realizar la ALU mediante la señal ALU Control. Así la unidad de control global no tiene que diferenciar entre los distintos tipos de instrucciones (I y R en este caso) y se gana en sencillez y eficiencia.

Este tipo de estructura se denomina en ocasiones de decodificación multinivel ya que en el primer nivel la unidad de control global decodifica la instrucción leyendo su opcode y sólo si se trata

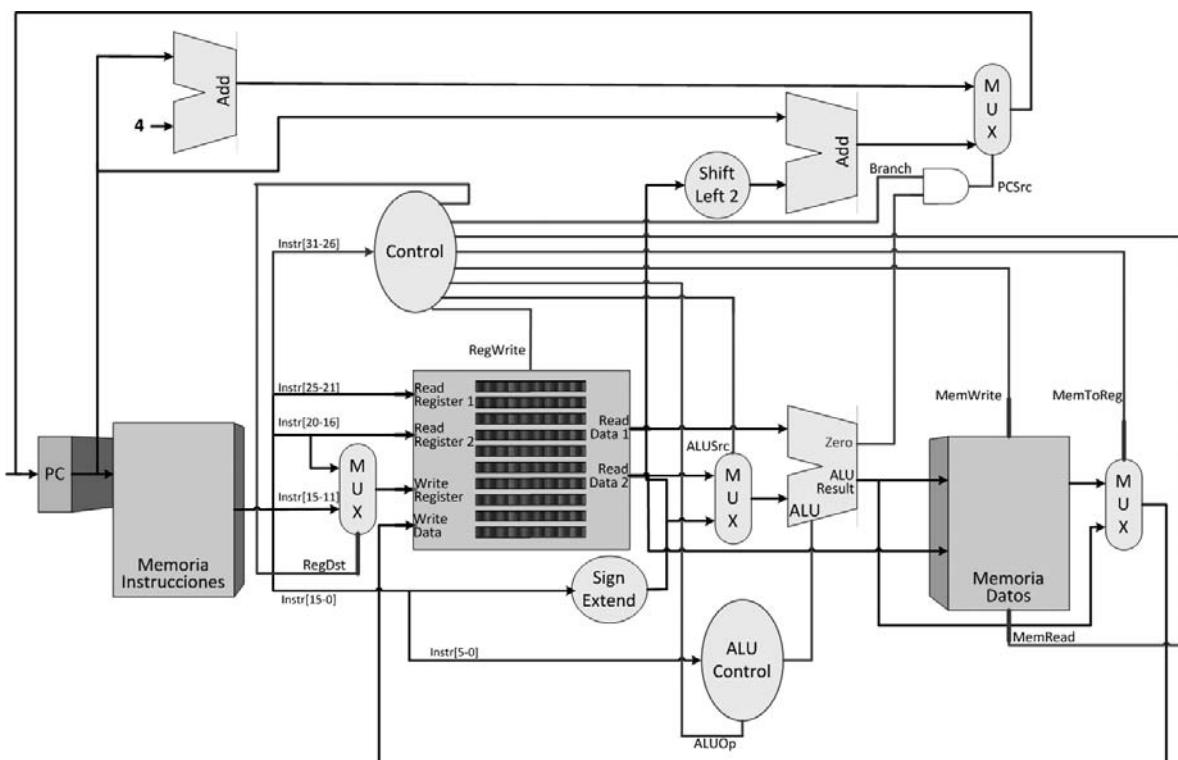


FIGURA 1.12

Ruta de datos y unidad de control del nanoMIPS monociclo.

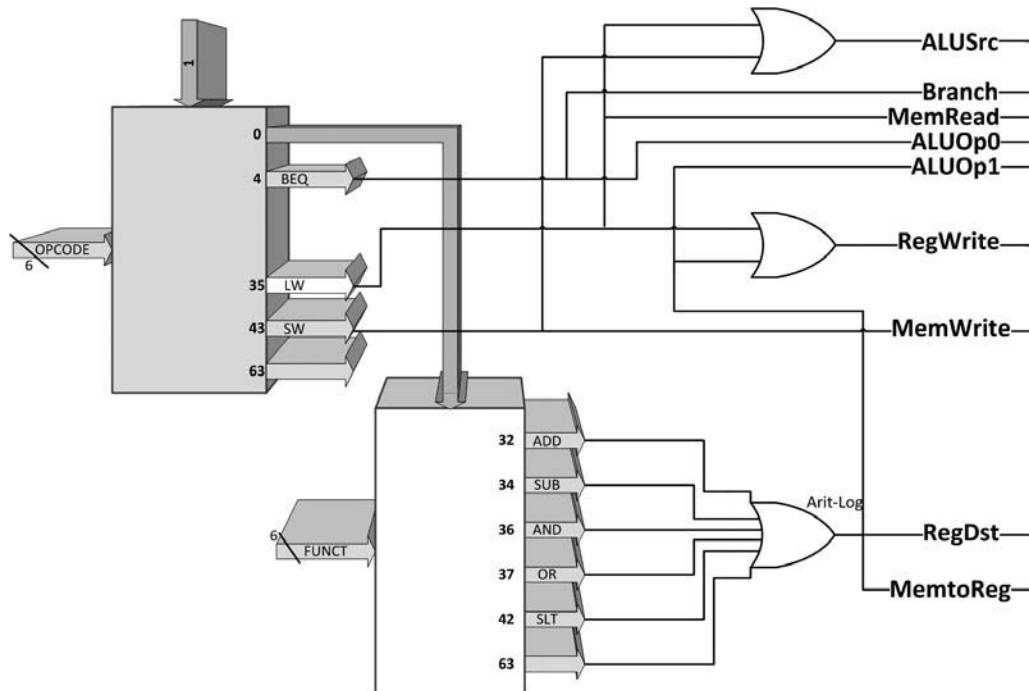


FIGURA 1.13

Diseño de la unidad de control para el nanoMIPS monociclo mediante decodificadores y puertas lógicas.

de una instrucción de tipo R, el control local de la ALU realiza una segunda decodificación leyendo para ello el campo Funct de la instrucción y averiguando la operación aritmético-lógica que debe realizarse.

La unidad de control global recibe como entrada el opcode de la instrucción decodificada (los bits entre el 31 y el 26 de la instrucción que se ha leído de la memoria de instrucciones). A partir de este opcode genera los valores adecuados para las diferentes señales de control.

La única peculiaridad es que el valor de PCSrc no se genera directamente, sino que se genera una señal Branch que vale 1 cuando la instrucción es un salto. Por eso se incluye una puerta AND en la ruta de datos que genera la señal PCSrc en función de esta señal y del resultado de la evaluación de la condición (comparación de igualdad) en la ALU mediante una resta.

Se puede observar que en el caso de la implementación monociclo del procesador, la unidad de control es un circuito combinacional muy sencillo ya que se decodifica la instrucción, a partir del opcode (y del campo Funct en el caso de las instrucciones de tipo R) se generan las señales de control que permanecen activas hasta que finaliza el ciclo y cuando llega una nueva subida del flanco de reloj, al comenzar una nueva instrucción, se vuelve a comenzar el proceso.

Esta unidad de control suele diseñarse mediante decodificadores y puertas OR, ya que si se trata de otro tipo de implementación, añadir una instrucción al repertorio puede obligar a modificar todo el diseño de la unidad de control. Sin embargo con los decodificadores, las modificaciones en el repertorio son casi inmediatas.

Por otro lado la unidad de control local de la ALU recibe como entradas la señal ALUOp y el campo Funct de las instrucciones de tipo R. Y genera la señal ALUControl según la tabla 1.4. La implementación se puede obtener con cualquiera de las metodologías de diseño típicas de circuitos combinacionales.

TABLA 1.4

Valores para el punto de control de la ALU en las instrucciones de tipo R.

Funct	Operación	ALUControl
100000	ADD	0010
100010	SUB	0110
100100	AND	0000
100101	OR	0001
101010	SLT	0111

1.4.2. Procesador multiciclo

Los diseños monociclo no se emplean en la actualidad ya que no son lo suficientemente eficientes, al tener que adaptar la duración del ciclo de reloj para que dé tiempo a que se ejecute la instrucción más larga del repertorio. Esto es impensable si, por ejemplo, se quieren introducir instrucciones de coma flotante en el repertorio, de duración mucho mayor que las de enteros.

Además es casi imposible optimizar la ruta de datos, porque es siempre la instrucción más larga la que limita el rendimiento del procesador. Y la mayor parte de los recursos del procesador están desaprovechados la mayor parte del tiempo (figura 1.14).

Para diseñar la ruta de datos multiciclo se utiliza una división de trabajo como la presentada en la sección 1.2 para ejecutar una instrucción completa (típica de los procesadores con repertorios RISC), de manera que cada una de ellas está relacionada con el hardware de la ruta de datos que se utiliza. Cada una de estas etapas debe completarse en un ciclo de reloj, por lo que se reduce significativamente el periodo del procesador, aunque a cambio, el CPI medio será mayor que 1. Cada instrucción tardará tantos ciclos en ejecutarse como sea necesario.

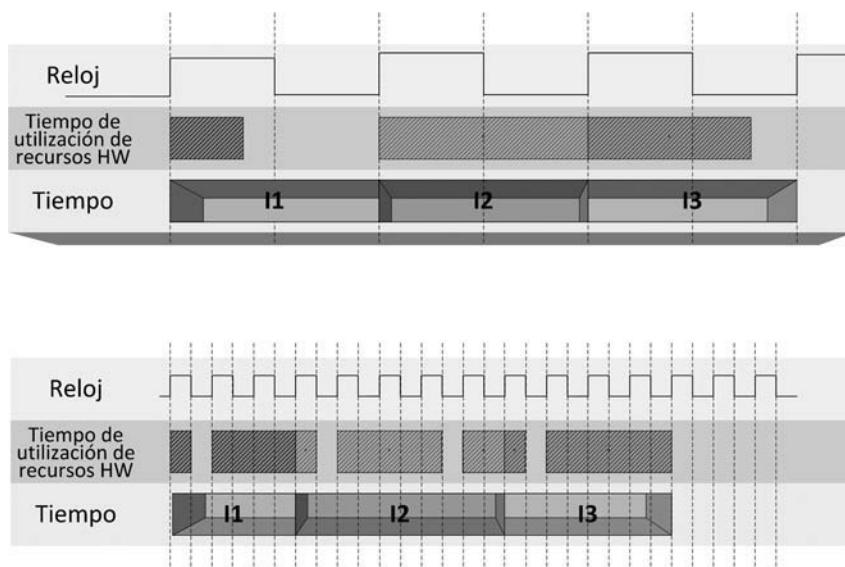


FIGURA 1.14

Comparación de la utilización de recursos en el nanoMIPS monociclo y el multiciclo.

Como una instrucción tarda más de un ciclo en ejecutarse, la ruta de datos es muy similar a la del procesador monociclo, pero es necesario incluir registros intermedios que almacenen los resultados temporales que se van generando según la instrucción avanza en su ejecución (figura 1.15).

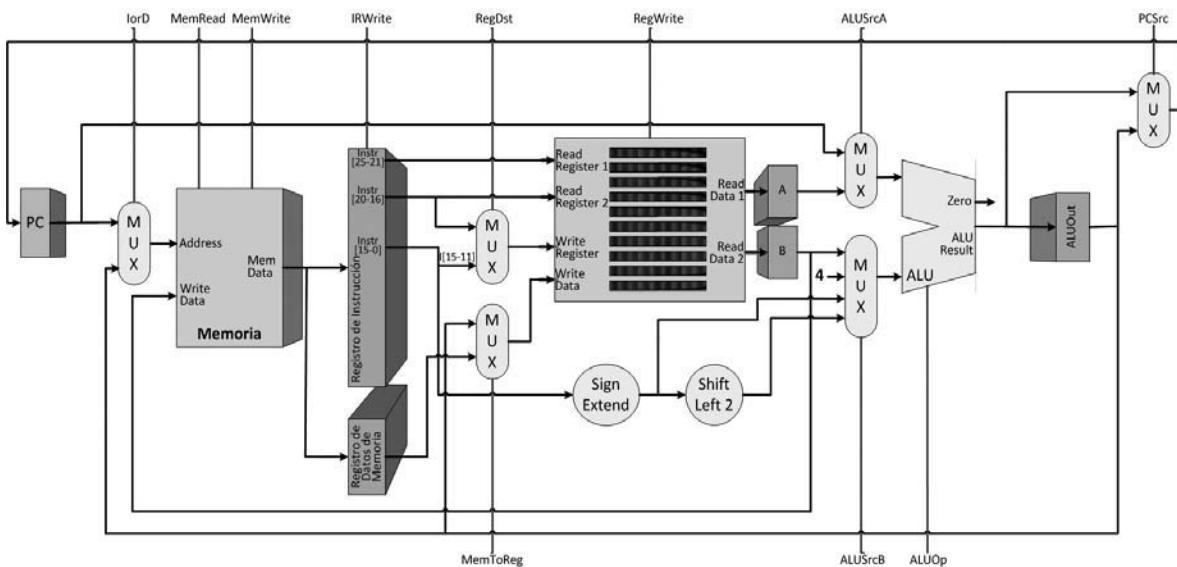


FIGURA 1.15

Ruta de datos del nanoMIPS multiciclo.

Con esta ruta de datos, cuando en el primer ciclo de ejecución se busca la instrucción que se va a ejecutar en la memoria (etapa F), se deja almacenada en el registro de instrucción IR. Así, cuando comienza el segundo ciclo de ejecución, la instrucción se lee de este registro para proceder a su codificación en la etapa D. En esta misma etapa se leen los operandos de la instrucción, y se dejan almacenados en los registros A y B para que en el siguiente ciclo de ejecución, la ALU tenga disponibles en estos registros sus operandos fuente para la etapa X. Y así sucesivamente, sin estos registros intermedios no sería posible realizar la ejecución de una instrucción en más de un ciclo de reloj.

Observando los módulos hardware que son necesarios en este caso para el correcto funcionamiento de la ruta de datos, son prácticamente los mismos que en el caso de la ruta de datos monociclo. Las únicas diferencias están en la aparición de estos registros intermedios (registro de instrucción o IR, registro de datos de memoria o MDR, registros para los operandos de la ALU A y B, y registro ALUOut para los resultados de las operaciones) y en que ahora ya no es necesario ningún sumador o comparador extra, se puede utilizar siempre la ALU, ni tener dos memorias separadas, basta con una. La explicación es muy sencilla, ya que, al tardar la instrucción más de un ciclo en ejecutarse, un mismo recurso (la ALU o la memoria) puede utilizarse en diferentes etapas de la ejecución de la instrucción sin problemas.

Por ejemplo, en el caso de una instrucción de salto BEQ, la ALU se utiliza en tres ocasiones. La primera vez, como en el resto de instrucciones, para sumar 4 al contador de programa y tener preparado su valor por si se tiene que buscar la instrucción siguiente. La segunda vez, para calcular la dirección destino de salto, sumando al contador de programa el desplazamiento incluido como inmediato en la instrucción. Y la tercera vez, para evaluar la condición del salto comparando los dos registros indicados en la instrucción, de manera que se cargará en el PC un valor u otro (el que corresponde a la instrucción siguiente o el que corresponde a la instrucción destino de salto) dependiendo del resultado de la evaluación de la condición.

De igual manera, una instrucción LW accede a la memoria única en dos ocasiones, en la primera como el resto de instrucciones, para leer la instrucción y volcarla al registro IR. En la segunda para leer el dato de la memoria y volcarlo al registro intermedio MDR antes de escribirlo en el registro destino correspondiente.

Ejemplo 1.4

Comparación de una versión monociclo del nanoMIPS con una versión multiciclo.

Recordemos que tenemos un ejemplo de implementación monociclo del nanoMIPS que funciona a una frecuencia máxima de 0.8 GHz (en este caso la limitación a la frecuencia de reloj la imponían las instrucciones de Store, que necesitaban 1.2 ns para completarse).

Si convertimos esta implementación en multiciclo, tenemos las siguientes latencias para los componentes hardware de la ruta de datos:

- Lectura de la memoria: 0.3 ns.
- Escritura en la memoria: 0.45 ns
- Lectura y escritura en el banco de registros: 0.05 ns
- Operación aritmético-lógica con la ALU: 0.25 ns
- Suma para preparar el PC siguiente: 0.1 ns

Esto significa que el periodo de reloj, limitado por la etapa más lenta, vendría determinado por la etapa M. Necesitamos $T=0.45$ ns para que esta etapa tenga tiempo de completarse en un ciclo de reloj. Por lo que en este caso $f_{\max} = 1/0.45 \cdot 10^{-9} = 2.2$ GHz.

Pero para comparar las dos versiones del procesador, monociclo y multiciclo, no podemos tener en cuenta sólo la frecuencia máxima de funcionamiento (o el periodo mínimo), ya que sabemos que los CPI son distintos en ambos tipos de procesador.

Para la implementación monociclo tenemos $CPI=1$, mientras que para la multiciclo, con estas frecuencias de aparición de instrucciones podemos calcular el CPI:

- Load: 20%
- Store: 10%
- Aritmético-lógicas: 35%
- BEQ: 35%

Cada uno de estos tipos de instrucciones atraviesa las siguientes etapas, cada una de ellas con duración de 1 ciclo de reloj:

- Load: $F+D+X+M+W = 5$ ciclos
- Store: $F+D+X+M = 4$ ciclos
- Aritmético-lógicas: $F+D+X+W = 4$ ciclos
- BEQ: $F+D+X = 3$ ciclos

Por lo tanto el CPI en el caso del procesador multiciclo es:

$$CPI = 0.20 \cdot 5 + 0.10 \cdot 4 + 0.35 \cdot 4 + 0.35 \cdot 3 = 3.85$$

Por lo tanto, comparando las implementaciones del nanoMIPS monociclo y multiciclo con el mismo repertorio de instrucciones:

$$S = \frac{t_{monociclo}}{t_{multiciclo}} = \frac{(I \cdot CPI \cdot T)_{monociclo}}{(I \cdot CPI \cdot T)_{multiciclo}} = \frac{1 \cdot 1.2}{3.85 \cdot 0.45} = 0.69$$

Observamos que el rendimiento del procesador monociclo es mejor que el del procesador multiciclo, porque con un procesador tan sencillo pesa más el CPI=1 que el hecho de funcionar con un periodo de reloj tan lento. Lo ideal sería que consiguiéramos aplicar alguna técnica de optimización al procesador multiciclo que permitiera mantener el periodo en 0.45 ns pero rebajando el CPI y acercándolo más a 1.

Reflexionando acerca de lo que implica que una instrucción pueda pasar más de un ciclo ejecutándose en el procesador, la principal diferencia con el procesador monociclo es que ahora los valores de los

puntos de control ya no se pueden dar como una tabla de verdad, ya que no se mantienen constantes a lo largo de toda la ejecución de la instrucción.

Como se trata de una implementación multiciclo, los valores de estas señales se van modificando en los diferentes ciclos de reloj y dependerán de la etapa en la que se encuentre la instrucción.

El conjunto de señales que hay que generar está compuesto por IorD, MemRead, MemWrite, MemToReg, RegDst, RegWrite, ALUSrcA, ALUSrcB, ALUOp, PCSrc, PCWrite, PCWriteCond e IRWrite.

Las señales de control para esta ruta de datos son muy similares a las de la ruta de datos de la versión monociclo. Las principales diferencias con la versión monociclo son que en este caso hace falta una señal ALUSrcA y que ALUSrcB es una señal de control de 2 bits porque hay que escoger entre cuatro alternativas en el multiplexor que selecciona la segunda entrada de la ALU. Además es necesaria la señal IRWrite para controlar la escritura del resultado de las lecturas de memoria en los registros IR o MDR (la señal equivalente MDRWrite no es más que IRWrite negada). El resto de señales de carga de los registros intermedios no se han incluido en el diseño de la unidad de control porque son mucho más sencillas de gestionar, ya que no vuelve a ocurrir que a la salida de un módulo hardware de la ruta de datos se pueda volcar el resultado en dos registros diferentes.

De nuevo la ruta de datos del nanoMIPS incorpora dos unidades de control diferentes, la unidad de control global (ahora algo más compleja que la del procesador monociclo) y el control local de la ALU (figura 1.16).

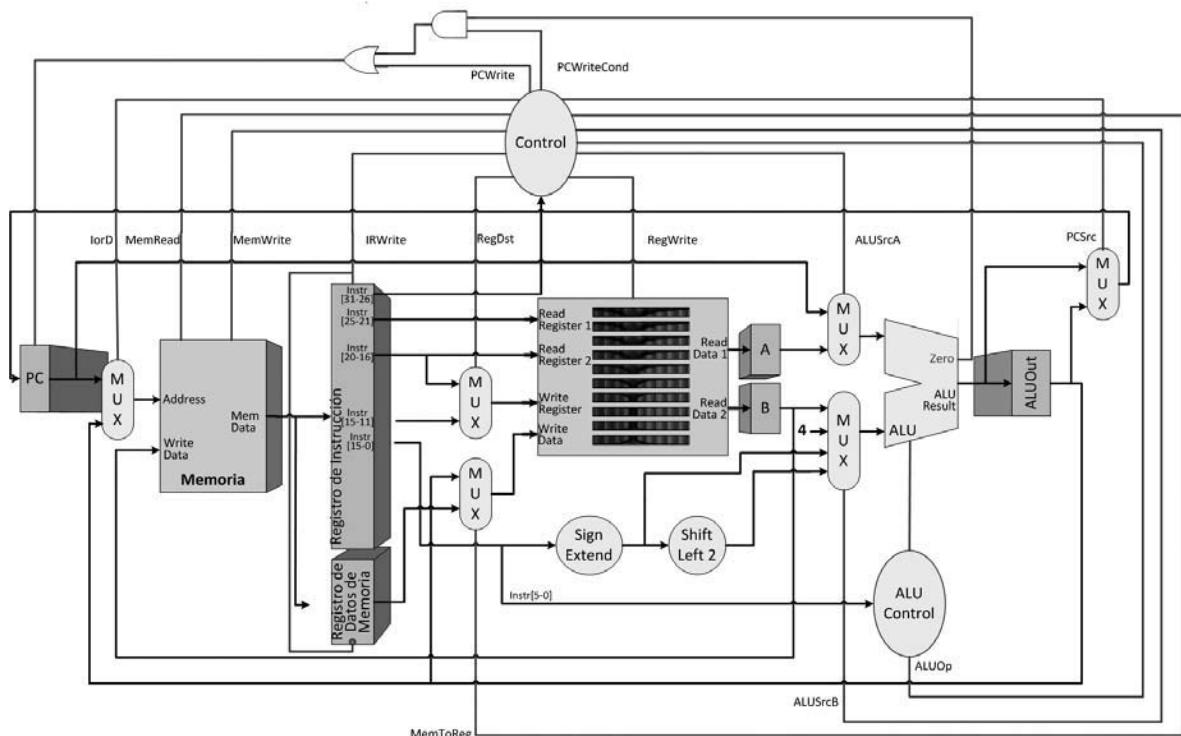


FIGURA 1.16

Ruta de datos y unidad de control del nanoMIPS multiciclo.

La unidad de control global se trata en este caso de un circuito secuencial, ya que hay que tener en cuenta la sincronización de todas las etapas de cada instrucción. Sigue recibiendo como entrada el opcode de la instrucción y se puede diseñar como una máquina de estados o mediante microprogramación.

I. CONCEPTOS BÁSICOS DE PROCESADORES

La unidad de control local de la ALU no se modifica respecto del procesador monociclo, ya que sólo se encarga de controlar el funcionamiento de la ALU en el ciclo en el que ésta debe de realizar la operación asociada a la instrucción.

Para el diseño de la unidad de control global del nanoMIPS como máquina de estados, se tienen en cuenta ocho estados que se ejecutan en un máximo de cinco ciclos o etapas. Los dos primeros ciclos, F y D, se corresponden con las dos primeras etapas de la ejecución de instrucciones y son estados de la unidad de control que se ejecutan para todas las instrucciones por igual. A partir de este punto, y con un repertorio tan sencillo como el del nanoMIPS, cada tipo de instrucción evoluciona por unos estados diferentes. Es decir, referencias a memoria (load y store), aritmético-lógicas (tipo R) y saltos (BEQ), siguen caminos diferentes (figura 1.17).

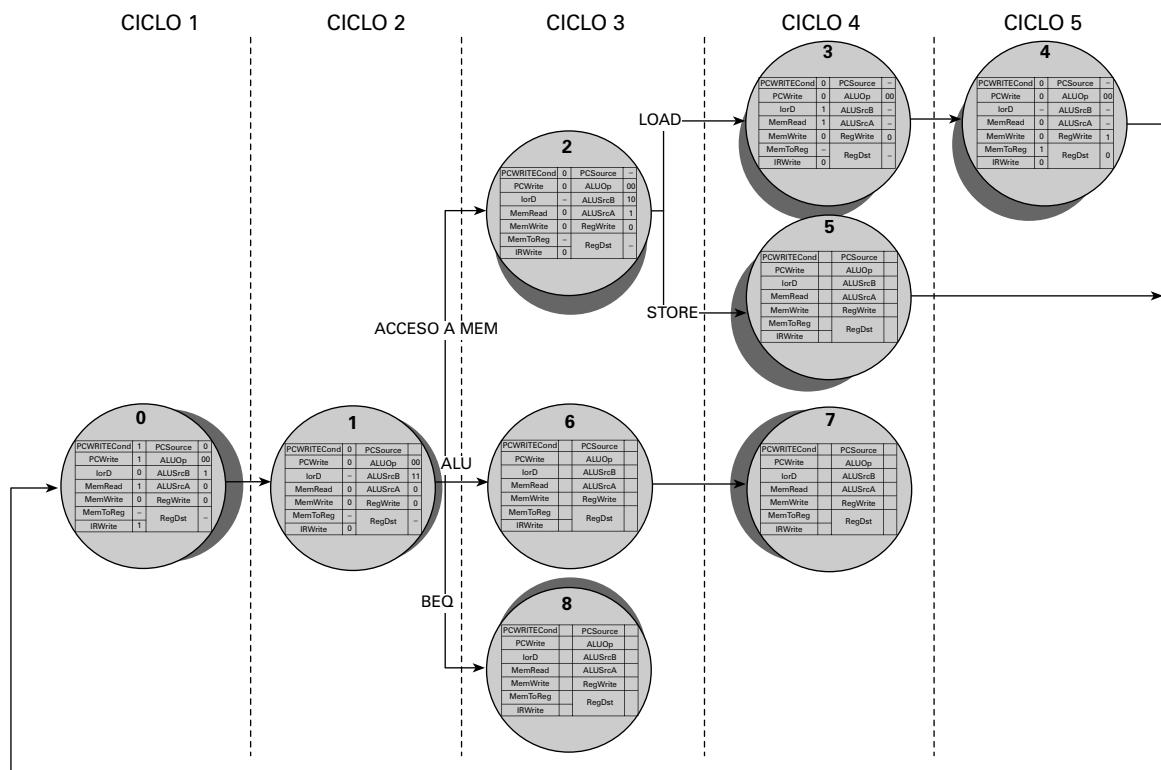


FIGURA 1.17

Diseño de la unidad de control para el nanoMIPS multiciclo como máquina de estados.

La unidad de control como máquina de estados (también llamada en algunos casos cableada) presenta serios problemas cuando los repertorios de instrucciones son más complejos que el del nanoMIPS. Además, cuando se modifica una única instrucción es necesario rediseñar toda la unidad de control.

Para solucionarlo se propone utilizar el control microprogramado, que no es más que una memoria, normalmente una PLA o una ROM, que almacena vectores con el valor de las señales de control que se deben generar en cada momento (figura 1.18). Para ejecutar una instrucción del repertorio adecuadamente basta con leer las sucesivas palabras de esta memoria en un orden determinado. A cada palabra de la memoria de control se le llama microinstrucción, y el conjunto ordenado de microinstrucciones que permiten ejecutar una instrucción del repertorio se llama microprograma.

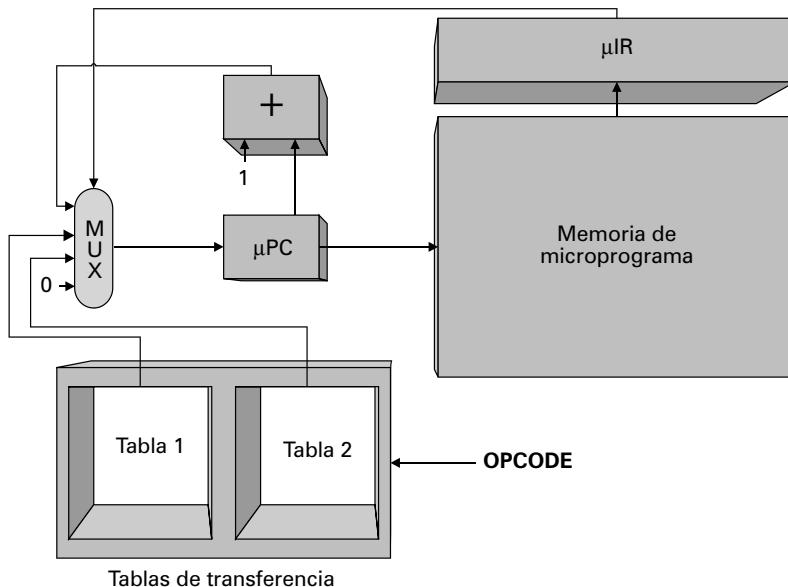


FIGURA 1.18

Diseño de la unidad de control microprogramada para el nanoMIPS multiciclo.

Este tipo de unidad de control es más flexible que la basada en máquina de estados y ocupa mucha menos área, pero presenta el inconveniente de su velocidad, ya que por norma general se trata de unidades de control bastante más lentas que las cableadas. Hay que tener en cuenta que para ejecutar una instrucción del repertorio del nanoMIPS es necesario recuperar 3 o 4 microinstrucciones de la memoria de micropograma, almacenarlas en el registro de microinstrucción (μ IR), dejar tiempo para que se establezcan las señales de control generadas, etc.

En el caso más sencillo, la unidad de control microprogramada del nanoMIPS utiliza codificación horizontal para las microinstrucciones, es decir, cada microinstrucción incorpora directamente los valores que deben tomar las señales de control.

En el caso de arquitecturas MIPS más complejas, se utilizan microinstrucciones altamente codificadas para evitar que éstas sean demasiado largas. Esto obliga a realizar un paso de decodificación previo a la activación de las señales de control, pero evita que la unidad de control ocupe demasiada área.

Si se utiliza esta técnica de diseño en el nanoMIPS se llega a una codificación vertical con 6 campos, cada uno de ellos codifica un grupo de señales de control relacionadas entre sí: control de la ALU (ALU-Control), selección de los operandos 1 (SRC1) y 2 (SRC2), control del banco de registros (RegisterControl), control de la memoria (Memory) y control del contador de programa (PCWriteControl). A estos campos se les añade la etiqueta de la microinstrucción y la información de secuencia. El significado de estos campos y los posibles valores de cada uno de ellos según los valores de las señales de control, se muestran en la tabla 1.5.

Como se puede observar en la figura 1.19, con cualquiera de los dos tipos de codificación dentro de las microinstrucciones no sólo se encuentra información acerca de las señales de control que deben activarse, sino también acerca del secuenciamiento de las microinstrucciones (último campo de 2 bits). En el caso de la unidad de control del nanoMIPS es un secuenciamiento explícito, por lo que en cada microinstrucción se almacena información acerca de la siguiente microinstrucción que se debe ejecutar.

Para comenzar a ejecutar una nueva instrucción en la ruta de datos, siempre se comienza por la dirección 0 del microprograma.

TABLA 1.5

Campos de las microinstrucciones del nanoMIPS (codificación de microinstrucción vertical).

Campo	Señales codificadas	Valores del campo	Valores de las señales	Explicación
ALU Control (2 bits)	ALUOp	Add Sub Funct	ALUOp=00 ALUOp=01 ALUOp=10	La ALU realiza una suma La ALU realiza una resta La ALU realiza la operación especificada por el campo Funct
SRC1 (1 bit)	ALUSrcA	PC A	ALUSrcA=0 ALUSrcA=1	Se escoge entre los dos valores posibles para el operando A de la ALU
SRC2 (2 bits)	ALUSrcB	B 4 Extend ExtendShift	ALUSrcB=00 ALUSrcB=01 ALUSrcB=10 ALUSrcB=11	Se escoge entre los dos valores posibles para el operando B de la ALU
Register Control (2 bits)	RegDst RegWrite MemToReg	Read WriteALU WriteMDR	RegWrite=0, RegWrite=1 y RegDst=1 MemtoReg=0 y RegWrite=1 RegDst=0 y MemtoReg=1	Se leen los dos registros y los resultados se guardan en los registros A y B Se escribe en un registro el resultado obtenido por la ALU Se escribe en un registro el valor que sale de la memoria accedida para datos
Memory (2 bits)	IorD MemRead MemWrite IRWrite	ReadPC ReadALU WriteALU	MemRead=1, IorD=0 y IRWrite=1 MemRead=1 y IorD=1 MemWrite=1 y IorD=1	Lectura de instrucción en memoria Lectura de datos en memoria Escritura de datos en memoria
PCWrite Control (1 bit)	PCWrite PCWriteCond PCSrc	ALU ALUOutCond	PCSrc=0 y PCWrite=1 PCSrc=1 y PCWriteCond=1	Carga en el PC de PC+4 Carga en el PC de la dirección de instrucción de salto si la condición de salto se cumple

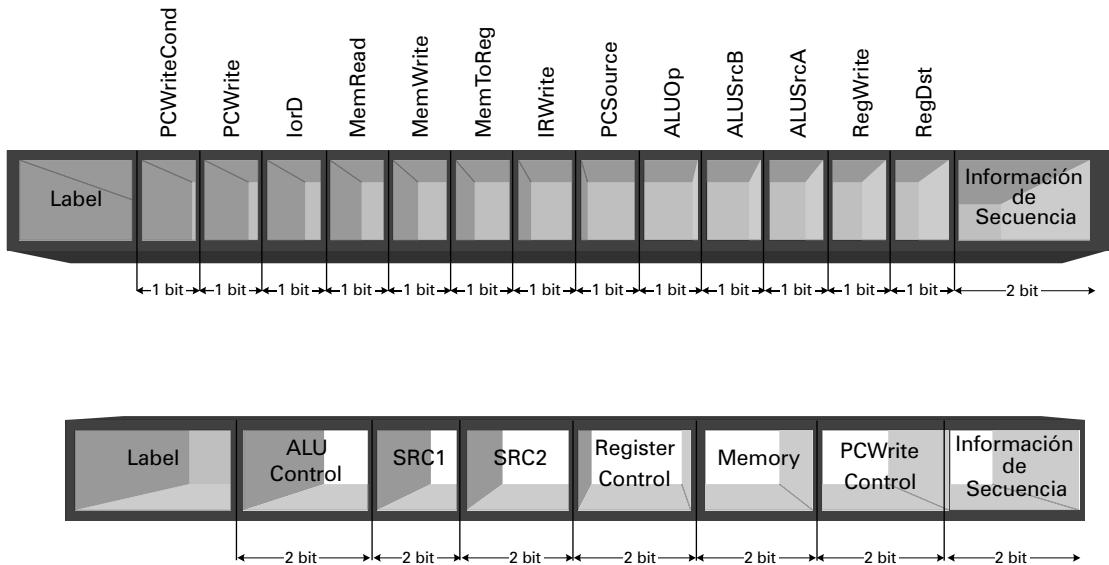


FIGURA 1.19

Microinstrucción para la unidad de control microprogramada del nanoMIPS: diferencias entre codificación de microinstrucción horizontal y vertical.

Existen cuatro opciones para cargar el PC del microprograma (μ PC), es decir, para escoger cuál es la siguiente microinstrucción que debe ejecutarse (por eso es necesario el multiplexor de la figura 1.18):

- La opción 0 es ejecutar la siguiente instrucción del microprograma (sumando 1 al μ PC). En este caso la información de secuencia de la microinstrucción vale 00.
- La opción 1 traduce el opcode de la instrucción máquina para la que se están generando las señales de control a una dirección de microprograma que permita escoger la bifurcación correcta según el tipo de instrucción del que se trate: instrucción de tipo ALU (tipo R), BEQ o acceso a memoria. Esta traducción se realiza en una tabla de transferencia que almacena estas bifurcaciones (la tabla 1 de la figura 1.18). La información de secuencia vale 01.
- La opción 2, de manera similar, traduce el opcode de la instrucción máquina para la que se están generando las señales de control a una dirección de microprograma que permita escoger la bifurcación correcta en el caso de las instrucciones de acceso a memoria: load o store (de nuevo se realiza la traducción en una tabla de transferencia, en este caso la tabla 2 de la figura 1.18). En este caso la información de secuencia de la microinstrucción vale 10.
- Y la opción 3 es la que permite ir a la microinstrucción 0. En este último caso la información de secuencia vale 11.

Con toda esta información, se puede programar el microcódigo completo del nanoMIPS (tabla 1.6) y el contenido de las tablas de transferencia, que normalmente también se implementarán sobre memorias PLA (tablas 1.7 y 1.8). Se puede observar que cada microinstrucción de la tabla 1.6 tiene una relación directa con los estados de la máquina de estados de la figura 1.17, utilizada para implementar la unidad de control cableada.

TABLA 1.6

Microcódigo completo para el nanoMIPS multiciclo (contenido de la memoria de microprograma cuando se utiliza codificación vertical de microinstrucciones).

	Label	ALU Control	SRC1	SRC2	Register Control	Memory	PCWrite Control	Información de Secuencia
0	Fetch	Add	PC	4		ReadPC	ALU	00
1		Add	PC	Extend Shift	Read			01
2	Mem	Add	A	Extend				10
3	LW					ReadALU		00
4					Write MDR			11
5	SW					WriteALU		11
6	InstR	Funct	A	B				00
7					WriteALU			11
8	BEQ	Sub	A	B			ALUOut Cond	11

TABLA 1.7

Tabla de transferencia 1 para el nanoMIPS multiciclo.

OPCODE de la instrucción	Tipo de instrucción	Valor simbólico (etiqueta de la siguiente microinstrucción)
000000	Instrucción ALU	InstR
000100	Instrucción BEQ	BEQ
100011	Instrucción LW	Mem
101011	Instrucción SW	Mem

TABLA 1.8

Tabla de transferencia 2 para el nanoMIPS multiciclo.

OPCODE de la instrucción	Tipo de instrucción	Valor simbólico (etiqueta de la siguiente microinstrucción)
100011	Instrucción LW	LW
101011	Instrucción SW	SW

1.4.3. Tratamiento de excepciones en procesadores secuenciales

Hasta ahora se han descrito los diseños del nanoMIPS monociclo y multiciclo suponiendo un funcionamiento correcto del procesador en todo momento. Pero existen excepciones a este funcionamiento «normal» que deben tratarse adecuadamente.

El tratamiento de excepciones implica, en casi todos los casos, transferir el control a otro programa que salve el estado del procesador cuando se produzca la excepción, corrija la causa de la excepción, restaure el estado del procesador y repita la ejecución de la instrucción causante para continuar con la ejecución por el punto en el que estaba. Este programa suele denominarse Rutina de Tratamiento de Excepción o RTE.

Existen varios tipos de excepciones:

- Interrupciones de E/S (se estudiará la gestión de estas interrupciones en el capítulo 2).
- Llamadas al Sistema Operativo.
- Puntos de ruptura.
- Códigos de operación inválidos.
- Overflow o desbordamiento en la ALU.
- Fallos de página.
- Accesos a memoria no alineados.
- Violación de zonas de memoria protegidas.
- Fallos del hardware.
- Fallos de alimentación.

De momento se supondrá que en el nanoMIPS sólo pueden producirse dos tipos de excepción: códigos de operación inválidos y overflow o desbordamiento de la ALU. Para gestionar estas excepciones sólo es necesario añadir algunos componentes hardware en la ruta de datos, las nuevas señales de control necesarias para gobernar su funcionamiento y modificar ligeramente el diseño de las unidades de control para generar los valores adecuados para estas señales.

En la ruta de datos es necesario incluir un registro Exception que almacene el código del tipo de excepción que se ha producido para que la RTE pueda leerlo y saber cuál ha sido la causa. En el caso del nanoMIPS, la decodificación de un opcode inválido tiene el código 0 y el desbordamiento de la ALU el código 1.

Además es necesario almacenar el valor del contador de programa cuando se produce la excepción menos 4 (porque siempre se deja preparado para ejecutar la instrucción siguiente sumándole 4) para que al volver de la RTE se pueda volver a ejecutar la instrucción que ha provocado la excepción. Este valor se almacena en el contador de programa de excepción o EPC. Y es necesario un restador para poder realizar la resta de 4 al contador de programa.

Por último, se debe cargar en el contador de programa el valor de la dirección de memoria donde comienza la RTE.

Por lo tanto, las señales de control relacionadas con el tratamiento de las dos posibles excepciones son:

- ALU_overflow e Illegal_opcode, que se activan cuando se produce cualquiera de las dos excepciones posibles en el nanoMIPS.
- Exception, para escribir el código de las excepciones que se produzcan en el registro Exception.
- ExceptionWrite y EPCWrite para controlar la escritura en los dos nuevos registros.

Y si es necesario, PCWrite para controlar la carga del contador de programa (esta señal sólo se utiliza en la implementación multiciclo del nanoMIPS).

La implementación de la nueva unidad de control en el caso del nanoMIPS monociclo es casi inmediata, ya que sólo hay que añadir estas nuevas señales al circuito combinacional diseñado con decodificadores y puertas OR que suele utilizarse.

Pero en el caso de la unidad de control para el nanoMIPS multiciclo, es necesario modificar la máquina de estados con la que se diseña la unidad de control, o añadir nuevos microprogramas a la unidad de control microprogramada.

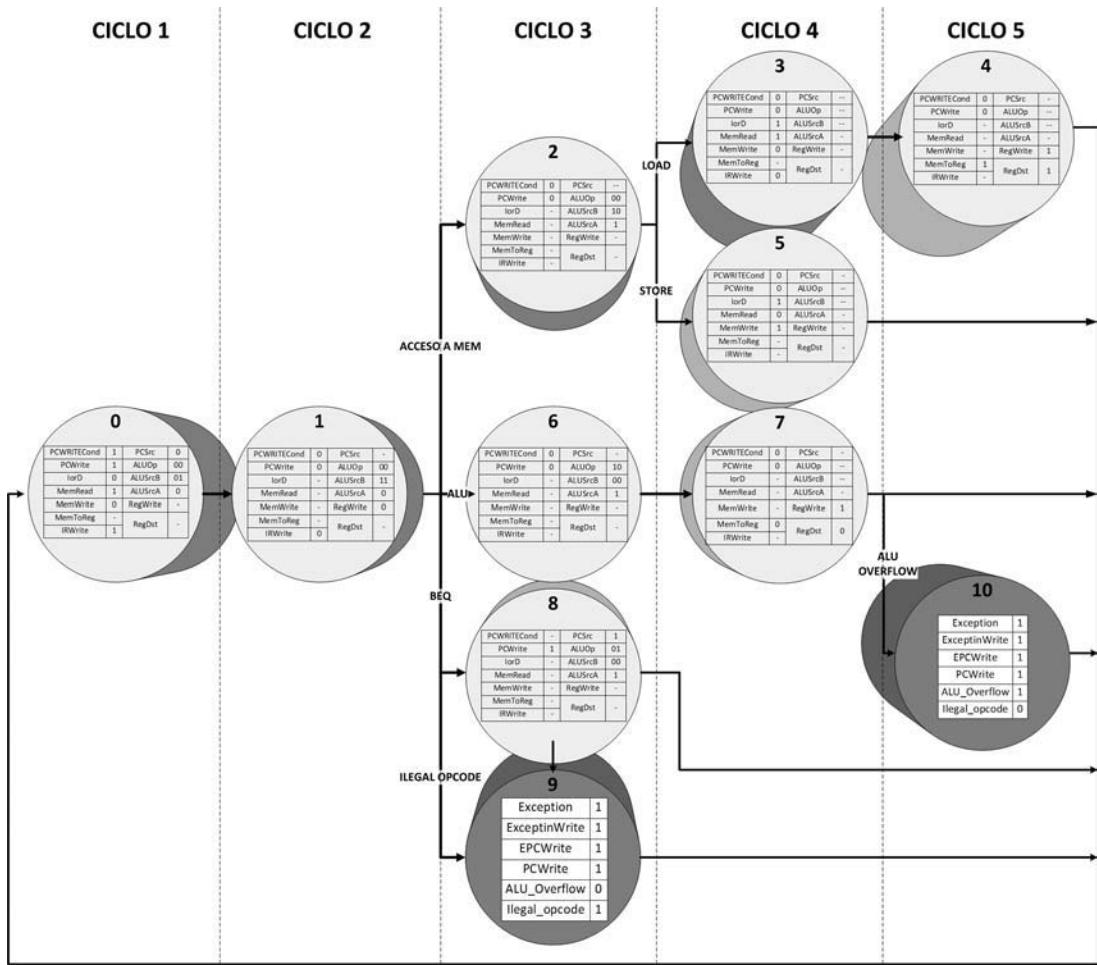


FIGURA 1.20

Diseño de la unidad de control para el nanoMIPS multiciclo como máquina de estados con los dos nuevos estados para el tratamiento de excepciones.

Para comprender el caso de la unidad de control multicycle, en la figura 1.20 se muestra la máquina de estados con los dos nuevos estados necesarios para realizar el tratamiento de excepciones. Al estado 9 se llega cuando al decodificar la instrucción se activa la señal `Ilegal_opcode` porque se detecta un código de operación que no es válido. Al estado 10 se llega cuando, tras realizar una operación en la ALU, se comprueba que se ha producido un desbordamiento (normalmente, comprobando los registros de control de la ALU) y se activa la señal `ALU_overflow`.

Para incluir los nuevos microprogramas en la unidad de control microprogramada bastaría con incluir las nuevas bifurcaciones hacia estos estados en las tablas de transferencia y las microinstrucciones que activen las nuevas señales de control en la memoria de microprograma.

1.5 Diseño de procesadores segmentados

Como se ha observado en el ejemplo 1.4, sería deseable encontrar la manera de tener un procesador con un CPI=1 pero que permitiera que el periodo de reloj no estuviera limitado por la instrucción más

lenta del repertorio, sino por la etapa más lenta de la ejecución de instrucciones. Es decir, tener una técnica de diseño que permita combinar las mejores características de los diseños monociclo y multiciclo.

Esta técnica es la segmentación y se incorpora en prácticamente todos los procesadores actuales, por eso se ha incluido en este capítulo y no en el capítulo 3, porque aunque surgió como una técnica de aumento de prestaciones para el procesador, hoy en día es indispensable en todas las arquitecturas.

1.5.1. Conceptos básicos de segmentación

La segmentación es una técnica de implementación de procesadores que permite solapar en el tiempo la ejecución de varias instrucciones. Es la primera técnica que se estudia en este libro que permite aprovechar el paralelismo a nivel de instrucción (ILP) dentro del procesador.

La idea en la que se basa esta técnica es la de una cadena de montaje, ya que si el trabajo que supone ejecutar una instrucción completa está dividido en diferentes etapas y a cada una de estas etapas le corresponden unos recursos hardware diferentes dentro de la ruta de datos del procesador, con este concepto se consigue aprovechar mucho mejor todo el hardware disponible en el procesador (figura 1.21). Ejecutando una única instrucción cada vez, como ocurre en todos los procesadores secuenciales sean monociclo o multiciclo, el hardware de la ruta de datos está desaprovechado la mayor parte del tiempo.

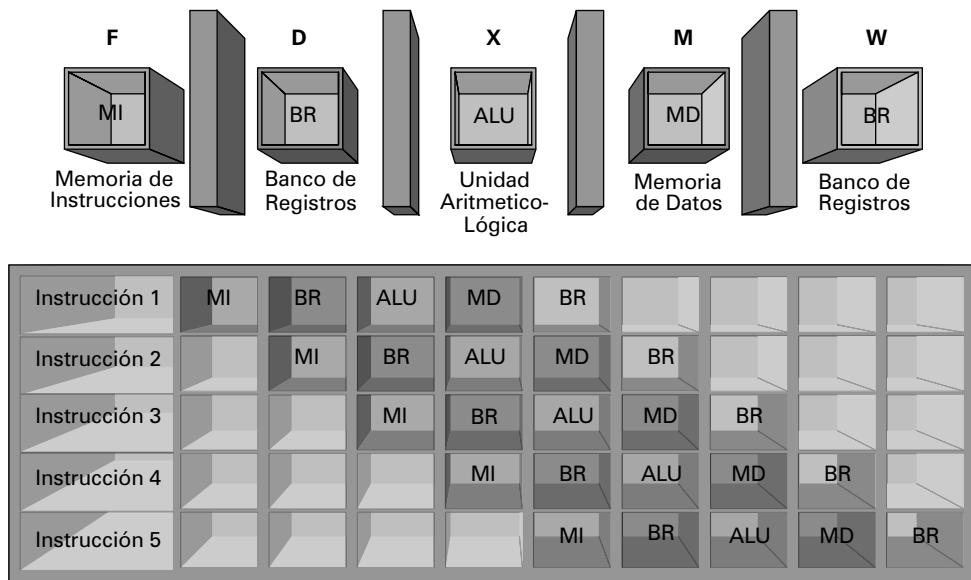


FIGURA 1.21

Concepto de segmentación.

Siguiendo con la idea de una cadena de montaje, el reto del diseñador es equilibrar las etapas en las que se divide una instrucción. De esta manera todas las etapas están listas para empezar al mismo tiempo y no tienen que esperar unas por otras.

Si se parte del nanoMIPS multiciclo, para segmentarlo basta con comenzar la ejecución de una nueva instrucción cada ciclo. Hay que asegurarse de que en cada etapa las diferentes instrucciones están utilizando diferentes recursos para que no haya ningún conflicto:

- Se realizan accesos a memoria en las etapas F (búsqueda de instrucción) y M (lectura o escritura de datos en las instrucciones de carga y almacenamiento).

- Se realizan accesos al banco de registros en las etapas D (lectura de operandos) y W (escritura de resultados).
- El PC cambia en F, cuando se deja preparado para buscar la siguiente instrucción, pero las instrucciones de salto lo modifican en M, después de evaluar la condición y comprobar que es verdadera.

Para solucionar estos problemas se separan de nuevo las memorias de instrucciones y datos (como se hizo con la ruta de datos monociclo). Además, se realizan las escrituras en los registros en la primera mitad del ciclo y las lecturas en la segunda mitad. Esta posibilidad de hacer turnos existe con los registros porque el acceso a ellos es muy rápido, con la memoria sin embargo no sería posible realizar dos accesos en un mismo ciclo y por eso es necesario separar la de instrucciones de la de datos. Por último se utiliza un multiplexor en la etapa F que se encarga de controlar todas las modificaciones del PC y se añade un restador (bloque Zero?) en la etapa de X para poder realizar al mismo tiempo la suma del desplazamiento al PC en la ALU y la comparación de los registros en el caso de las instrucciones BEQ (evaluación de la condición del salto).

Además de evitar conflictos en la utilización de recursos, hay que asegurar que unas etapas no interfieren con otras. Para ello se utilizan los registros de segmentación, en los que se almacenan los resultados de cada etapa al final del ciclo de reloj y se utilizan como entrada de la etapa siguiente al comenzar el siguiente ciclo de reloj (figura 1.22).

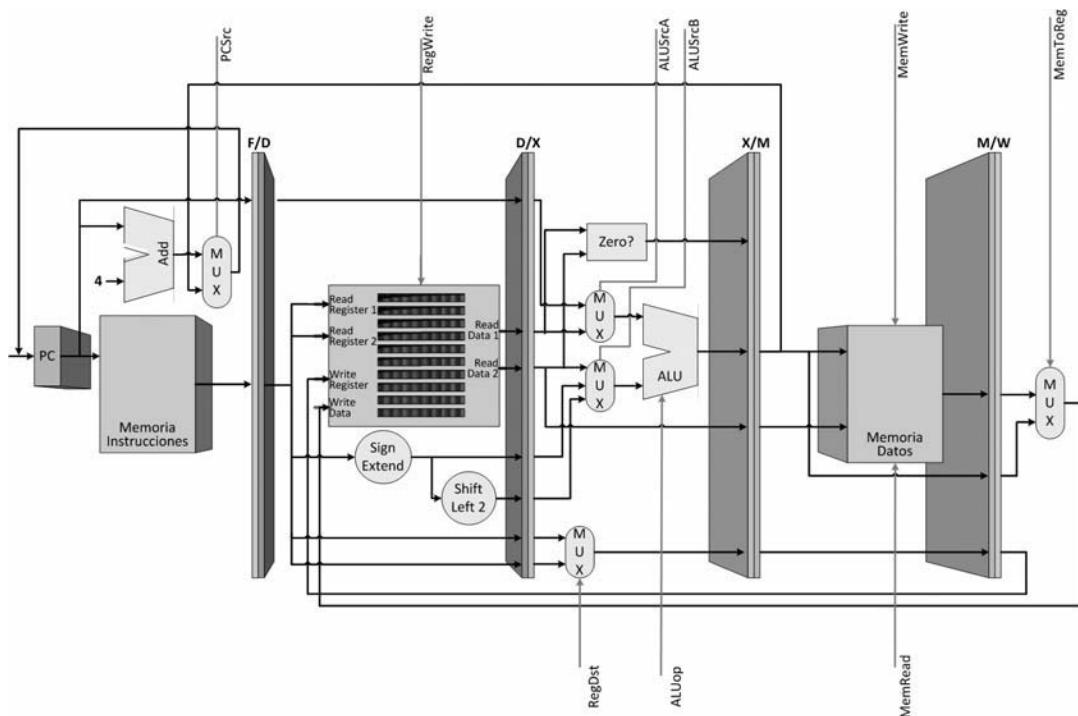


FIGURA 1.22

Ruta de datos del nanoMIPS segmentado.

Estos registros constituyen, fundamentalmente, todo el hardware necesario para evitar conflictos de recursos entre las diferentes etapas y sustituyen a los registros intermedios que aparecían en la ruta de datos del procesador multiciclo. Cada uno tendrá una longitud diferente, dependiendo de la información que tenga que almacenar para que las instrucciones puedan avanzar por la ruta de datos (la propia instrucción, operandos y resultados, señales de control, etc.).

En cuanto a la unidad de control del procesador segmentado, al decodificar la instrucción en la etapa D, el opcode de la instrucción (y el campo Funct para las instrucciones R) permite generar todas las señales de control necesarias para la ejecución de la instrucción. Algunas de estas señales se utilizan en la propia etapa D, pero otras serán necesarias para la correcta ejecución de la instrucción en las etapas X, M y W. Normalmente la unidad de control se basa en una tabla de reservas que tiene tantas filas como etapas tiene el procesador, y una columna para cada ciclo de ejecución, de manera que al comenzar una nueva instrucción se pueda marcar en esta tabla qué etapas van a estar ocupadas en cada ciclo.

Las señales de control generadas se propagan junto con el resto de valores por los registros de segmentación hasta que lleguen a la etapa adecuada (figura 1.23). Así que de nuevo la unidad de control es un circuito combinacional que genera las señales de control necesarias para cada tipo de instrucción a partir del opcode. La diferencia con el procesador monociclo es que en este caso hay que dimensionar adecuadamente los registros de segmentación, de manera que tengan espacio para almacenar las señales de control en cada etapa. Para la etapa X son necesarias las señales ALUSrcA, ALUSrcB (de dos bits), ALUOp (de nuevo de 2 bits) y RegDst. Para la etapa M, son necesarias las señales Branch, MemRead y MemWrite. Y por último, para la etapa W se necesitan los valores de las señales MemToReg y RegWrite. En total son necesarias nueve señales para controlar esta ruta de datos segmentada, sólo una más que en el caso de la ruta de datos monociclo.

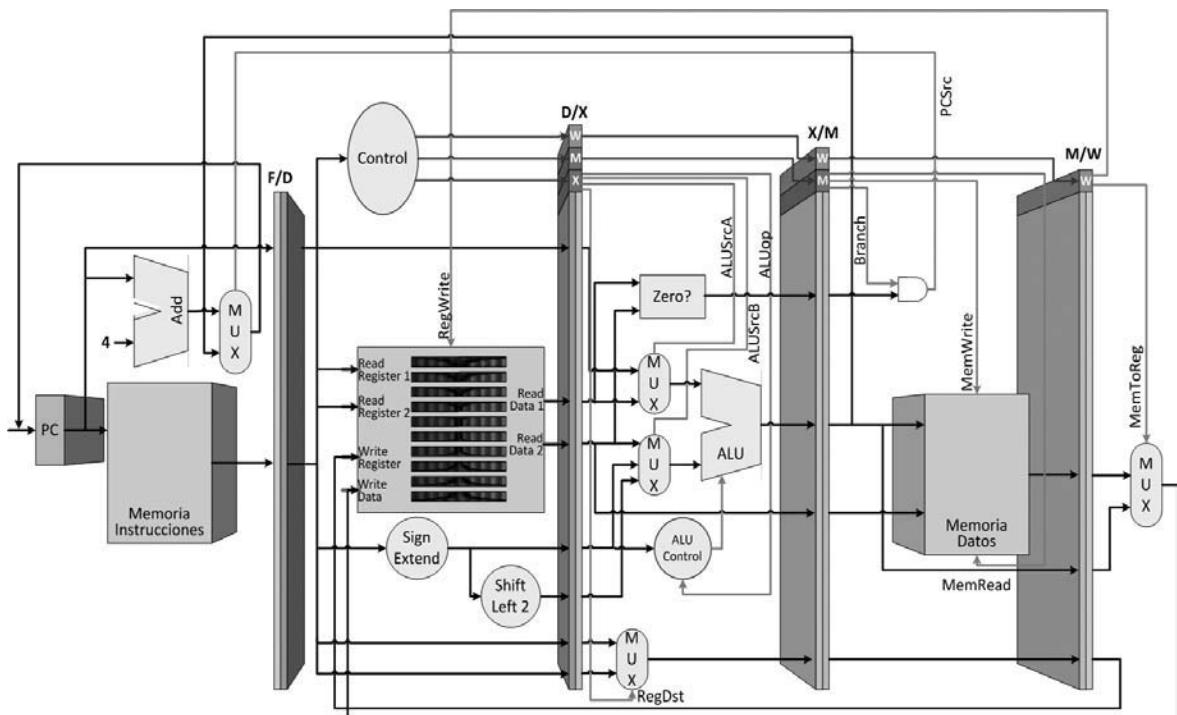


FIGURA 1.23

Ruta de datos y unidad de control del nanoMIPS segmentado.

¿Cómo se modifica el rendimiento del procesador multiciclo cuando se convierte en segmentado? En primer lugar, su productividad (número de instrucciones que se terminan por segundo) aumenta, ya que idealmente se termina una nueva instrucción por ciclo. Sin embargo el tiempo que una única instrucción tarda en ejecutarse (denominado latencia de instrucción) aumenta ligeramente debido al hardware extra que se introduce en la ruta de datos para segmentarla.

Teniendo en cuenta los aspectos que influyen directamente en el tiempo de CPU según la ecuación de prestaciones:

- El CPI ideal de un procesador segmentado debe ser 1, de manera que se finaliza una nueva instrucción cada ciclo y en media, es como si cada instrucción sólo necesitara 1 ciclo para ejecutarse. Es decir, se consigue el CPI de un procesador secuencial monociclo.
- El periodo de reloj viene limitado por la etapa más lenta, a la que tiene que dar tiempo a terminar en un ciclo. Es decir, se consigue el periodo de un procesador secuencial multiciclo. En realidad ligeramente peor, debido al nuevo hardware que aparece en la ruta de datos para que ésta funcione de manera segmentada.

En el caso del nanoMIPS, al calcular el speedup obtenido gracias a la segmentación del procesador multiciclo, obtenemos:

$$S = \frac{t_{multiciclo}}{t_{segmentado}} = \frac{(I \cdot CPI \cdot T)_{multiciclo}}{(I \cdot CPI \cdot T)_{segmentado}} = \frac{5}{1} = 5 = \text{número de etapas}$$

El número de instrucciones es el mismo en ambos casos ya que el repertorio de instrucciones no se modifica. Por lo demás, se trata de una aproximación, ya que se está suponiendo que el CPI en el multiciclo es prácticamente 5 (puede ser algo menor dependiendo de la proporción de cada tipo de instrucción, pero 5 es la cota segura) y que el periodo de reloj empeora imperceptiblemente al introducir los registros de segmentación en la ruta de datos. También se está despreciando el tiempo que la ruta de datos tarda en estar trabajando al 100%, es decir, en tener ocupadas todas sus etapas (lo que en muchas ocasiones se denomina tiempo o ciclos de llenado). Pero generalizando, el speedup máximo que puede conseguirse al segmentar un procesador multiciclo es igual al número de etapas en el que se segmenta la ruta de datos.

En principio, se podría plantear entonces aumentar en todo lo posible el número de etapas para conseguir así la máxima ganancia respecto del procesador multiciclo. De hecho existe un estilo de diseño denominado supersegmentación o hipersegmentación que consiste en aumentar el número de etapas por encima de 20, de manera que se pueda reducir lo más posible la duración del periodo de reloj, realizando muy poco trabajo en cada una de ellas (algunas pueden ser incluso etapas de transferencia de información entre la etapa anterior y la siguiente que no realizan ningún otro tipo de trabajo útil).

Pero existe un límite en el número de etapas de segmentación que viene impuesto por la sobrecarga que introducen los registros de segmentación y por la dificultad de encontrar etapas equilibradas en cuanto a la cantidad de trabajo que realizan.

Ejemplo 1.5

Comparación de una versión multiciclo del nanoMIPS con una versión segmentada.

Recordemos que para la implementación multiciclo, teníamos las siguientes latencias para los componentes hardware de la ruta de datos (ejemplo 1.4):

- Lectura de la memoria: 0.3 ns
- Escritura en la memoria: 0.45 ns
- Lectura y escritura en el banco de registros: 0.05 ns
- Operación aritmético-lógica con la ALU: 0.25 ns
- Suma para preparar el PC siguiente: 0.1 ns

Lo que implicaba $T=0.45$ ns y con estas frecuencias de instrucción un CPI de 4.55.

Supongamos ahora que para segmentar este procesador utilizamos registros de segmentación con un retraso de 0.01 ns cada uno. Esto implica $T = 0.45$ ns + $2 \cdot 0.01$ ns = 0.47 ns, ya que en un ciclo de reloj debe dar

tiempo a leer los contenidos del registro de segmentación de entrada, a realizar la etapa más lenta de la ruta de datos, y a escribir los resultados de la etapa en el registro de segmentación de salida.

Por lo tanto, el speedup que se obtiene al segmentar el nanoMIPS multicitclo es:

$$S = \frac{t_{multicitculo}}{t_{segmentado}} = \frac{(I \cdot CPI \cdot T)_{multicitculo}}{(I \cdot CPI \cdot T)_{segmentado}} = \frac{4.55 \cdot 0.45}{1 \cdot 0.47} = 4.36$$

Es decir, un valor muy próximo al speedup teórico máximo de 5 que se debería obtener al segmentar el procesador de 5 etapas. No se llega a este valor máximo porque se ha contado con datos suficientes para calcular el CPI medio del nanoMIPS multicitculo y es algo inferior a 5, y porque se ha calculado el periodo de la versión segmentada de manera realista, teniendo en cuenta el retardo introducido por las lecturas y escrituras en los registros de segmentación.

Hasta ahora se ha supuesto que segmentar el procesador será suficiente para conseguir que finalice una nueva instrucción en cada ciclo de reloj y, gracias a ello, obtener un CPI=1 en todos los casos en los que el procesador está segmentado. Pero en realidad éste sería sólo el mejor de los casos, porque existen ciertas situaciones denominadas riesgos, que impiden por diferentes circunstancias que se finalice una nueva instrucción cada ciclo, y por lo tanto incrementan el CPI por encima de este valor ideal de 1.

1.5.2. Resolución de riesgos en procesadores segmentados

Para poder solucionar los riesgos en un procesador segmentado primero hay que comprender los diferentes tipos de riesgos que pueden aparecer. Se identifican tres tipos diferentes:

- Riesgos estructurales.** Se producen cuando dos o más instrucciones necesitan utilizar el mismo recurso hardware al mismo tiempo. En la figura 1.24 se observa un caso típico del riesgo estructural que se produciría en el nanoMIPS si no se separaran las memorias de instrucciones y datos, ya que diferentes instrucciones estarían intentando acceder al mismo recurso hardware al mismo tiempo (todas las instrucciones necesitan leer la instrucción de memoria en la fase F y las de load y store necesitan leer o escribir un dato en memoria en la fase M).

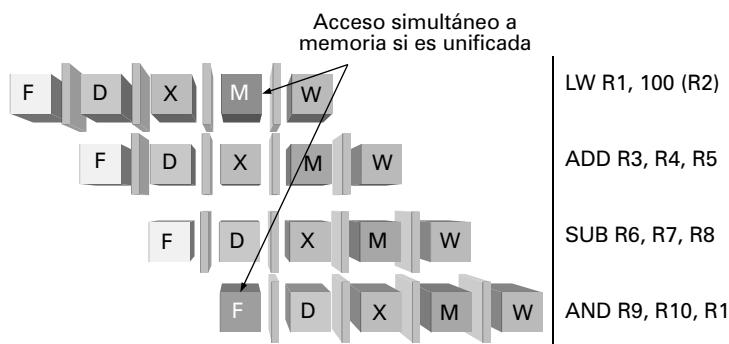


FIGURA 1.24

Ejemplo de riesgos estructurales en el nanoMIPS.

- **Riesgos de datos.** Se producen cuando dos o más instrucciones presentan dependencias de datos entre sí que, si no se resuelven correctamente, podrían llevar a la obtención de resultados erróneos en la ejecución del código por realizar operaciones de lectura y escritura en un orden diferente al indicado por la secuencia de instrucciones. Existen tres tipos de riesgos de datos, según el tipo de dependencia que los provoque:
 - RAW o Read After Write (Lectura después de Escritura).
 - WAR o Write After Read (Escritura después de Lectura).
 - WAW o Write After Write (Escritura después de Escritura).

Las dependencias RAW son las reales, ya que las WAW y las WAR, aunque pueden provocar riesgos, se deben a la reutilización de los registros visibles para el programador, pero no existe un flujo real de información entre las instrucciones que provocan la dependencia.

En el caso del nanoMIPS sólo pueden producirse riesgos RAW, ya que al pasar todas las instrucciones por el mismo número de etapas y terminar en orden, las dependencias WAR y WAW nunca provocan riesgos. En la figura 1.25 se observa un ejemplo de las distintas dependencias y de los riesgos que provocan. La dependencia RAW entre el LW y el ADD sí que es un riesgo, y si no se resuelve (como ocurre en la figura), el resultado de la ejecución sería erróneo, ya que uno de los operandos de la suma no tiene el valor correcto. Pero las dependencias por R3 (WAW) y por R5 (WAR), no suponen ningún riesgo ya que las lecturas y escrituras siempre quedarán en el orden adecuado.

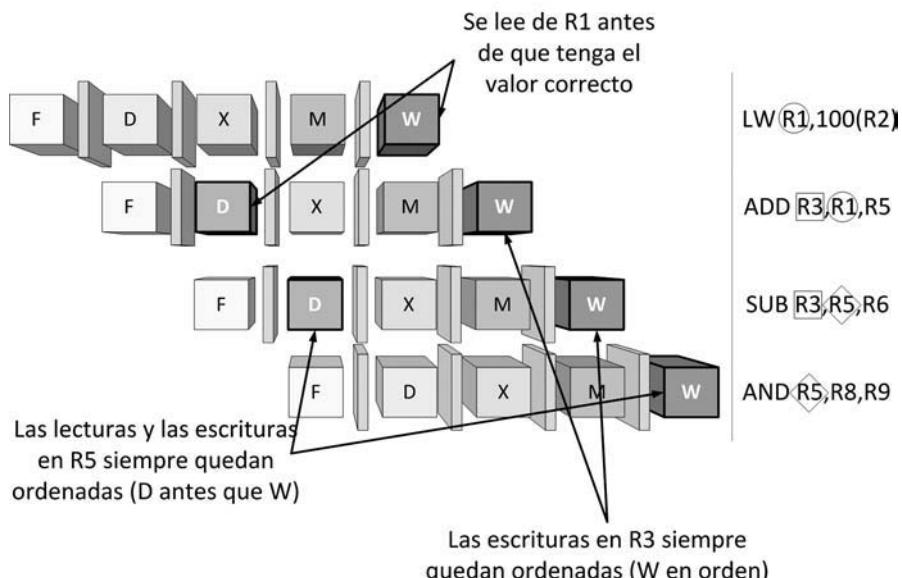


FIGURA 1.25

Ejemplo de dependencias y riesgos de datos en el nanoMIPS.

- **Riesgos de control.** Se producen cuando una instrucción que modifica el valor del PC todavía no lo ha hecho cuando se tiene que comenzar la ejecución de la siguiente instrucción. En el caso del nanoMIPS, la instrucción que provoca este tipo de riesgo es el BEQ, ya que hasta la fase M no carga el valor adecuado para el PC, y en los tres ciclos que tarda en llegar a esta fase el procesador no puede comenzar la ejecución de ninguna instrucción (figura 1.26).



FIGURA 1.26

Ejemplo de riesgo de control en el nanoMIPS.

Existe una técnica universal para resolver estos tres tipos de riesgos, la introducción de paradas en la ruta de datos (se muestran ejemplos para los tres tipos de riesgo en las figuras 1.27, 1.28 y 1.29). Con esta técnica se asegura la obtención de resultados correctos y se evitan los conflictos por recursos y por incertidumbres en cuanto al valor del PC. Es la unidad de control la encargada de realizar la detección de riesgos en la etapa D (con una lógica específica para esta tarea que evalúa estos riesgos cuando se decodifica la instrucción) y de inhibir el avance de las instrucciones por la ruta de datos hasta que estos queden resueltos.

TABLA 1.9

Penalizaciones debidas a riesgos de datos en el nanoMIPS cuando la instrucción origen del dato es una aritmético-lógica o un load.

	Aritmético-lógica	Load
Tipo de instrucción destino	Aritmético-lógica, load, store, salto condicional	Aritmético-lógica, load, store, salto condicional
Registro que provoca el riesgo	Registro entero R	Registro entero R
Etapas de escritura del registro	W	W
Etapas de lectura del registro	D	D
Número de paradas	2	2

El problema de esta técnica es que los riesgos de datos y de control son muy comunes, y si se resuelven siempre mediante la introducción de paradas, el CPI del procesador se aleja mucho del valor ideal de 1 que se busca con la segmentación (tabla 1.9).

Por ello se utilizan otras técnicas, tanto hardware como software, que aunque aumentan ligeramente la complejidad del diseño del procesador y del compilador, permiten resolver los riesgos manteniendo el CPI lo más cercano a 1 que sea posible.

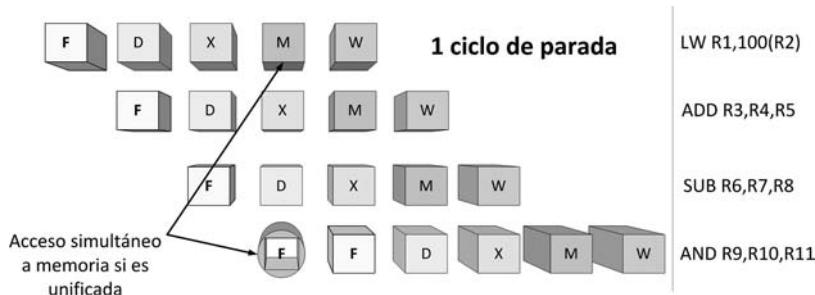


FIGURA 1.27

Ejemplo de resolución de riesgos estructurales en el nanoMIPS con una única memoria mediante paradas en la ruta de datos.

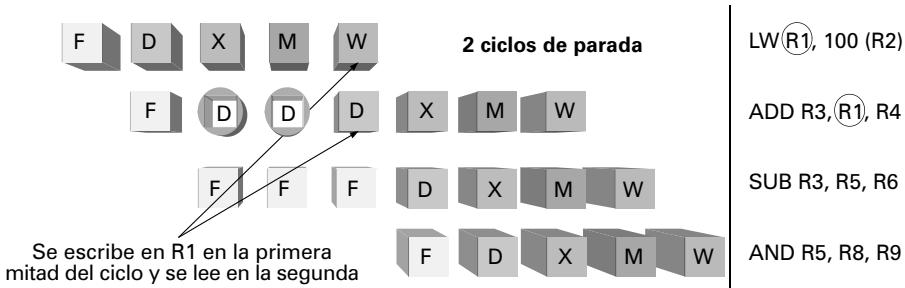


FIGURA 1.28

Ejemplo de resolución de riesgos de datos en el nanoMIPS mediante paradas en la ruta de datos.

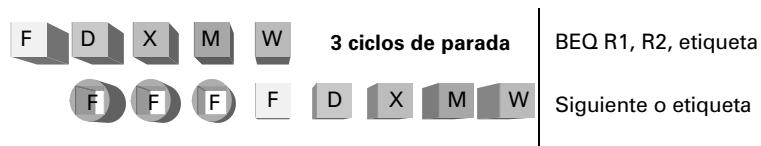


FIGURA 1.29

Ejemplo de resolución de riesgos de control en el nanoMIPS mediante paradas en la ruta de datos.

1.5.2.1. RESOLUCIÓN DE RIESGOS ESTRUCTURALES

La resolución de estos riesgos suele ser la más sencilla ya que basta con duplicar los recursos hardware que provocan los conflictos, segmentarlos o realizar turnos para acceder a ellos.

Son las soluciones que se han utilizado en el diseño del nanoMIPS para evitar los riesgos estructurales por la memoria y por el banco de registros, separando entre las memorias de instrucciones y datos y realizando turnos para leer y escribir en el banco de registros (las escrituras siempre se hacen en la primera mitad de los ciclos de reloj y las lecturas, en la segunda).

1.5.2.2. RESOLUCIÓN DE RIESGOS DE DATOS

Para resolver los riesgos de datos RAW que aparecen en el nanoMIPS existe una técnica hardware y otra software.

La técnica hardware se denomina adelantamiento o cortocircuito y consiste en pasar directamente el resultado obtenido con una instrucción a las instrucciones que lo necesitan como operando. Es decir, estas instrucciones no tienen que esperar a que la que produce el resultado lo escriba en el registro destino, sino que reciben su valor en cuanto está disponible.

Esta técnica puede mejorar mucho el rendimiento de un procesador segmentado y es fácil de implementar, ya que lo único que hay que hacer es identificar todos los posibles adelantamientos necesarios para el repertorio que ejecuta el procesador y comunicar los registros de segmentación involucrados (la ruta de datos modificada para implementar el adelantamiento se muestra en la figura 1.30). Pero no siempre se puede aplicar esta técnica evitando con ello que se produzcan paradas (tabla 1.10).

Si se estudia en profundidad el caso del nanoMIPS tenemos las siguientes posibilidades para riesgos RAW (figura 1.31):

- LW seguido de SW: El operando está disponible a la salida de la etapa M de la instrucción LW y se necesita a la entrada de la etapa M de la instrucción SW.

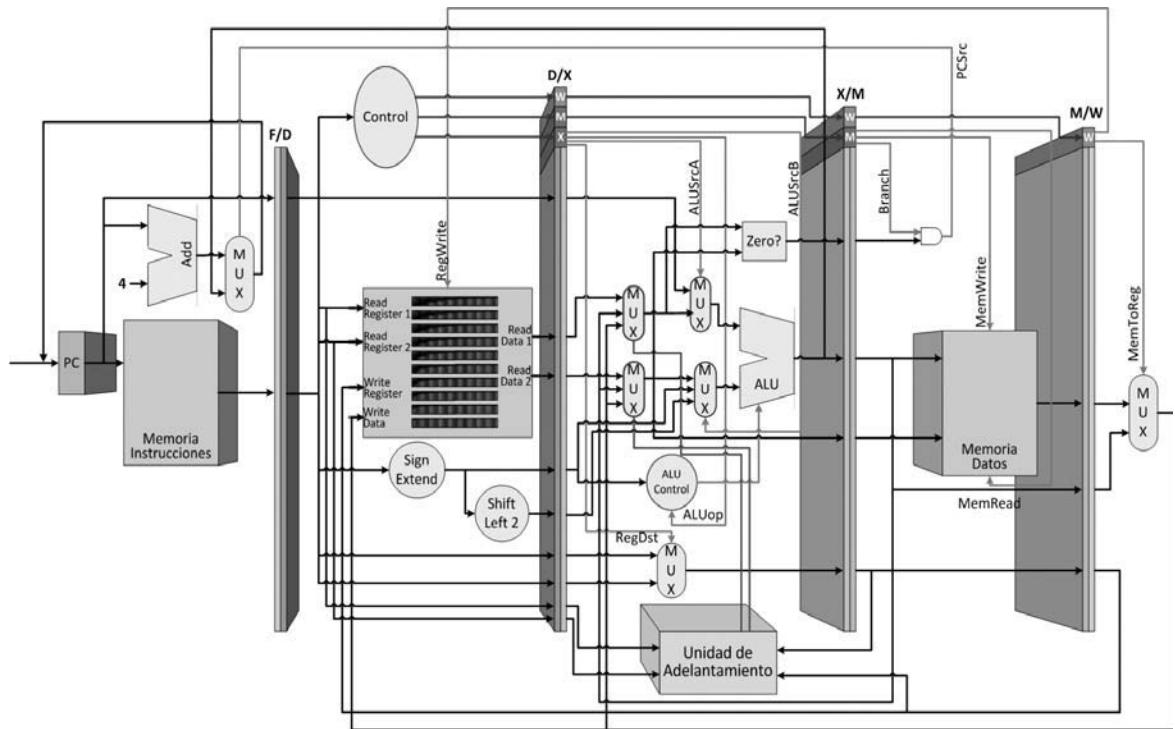


FIGURA 1.30

Ruta de datos del nanoMIPS con adelantamiento.

TABLA 1.10

Penalizaciones debidas a riesgos de datos en el nanoMIPS cuando la instrucción origen del dato es una aritmético-lógica o un load y se utiliza adelantamiento para resolver los riesgos de datos.

	Aritmético-lógica	Load	Load
Tipo de instrucción destino	Aritmético-lógica, load, store, salto condicional	Aritmético-lógica, load, salto condicional	Store
Registro que provoca el riesgo	Registro entero R	Registro entero R	Registro entero R
Etapa de escritura del registro	W	W	W
Etapa de lectura del registro	D	D	D
Adelantamiento desde la salida de las etapas	X o M	M	M
Adelantamiento hacia la entrada de las etapas	X o M	X	M
Número de paradas	0	1	0

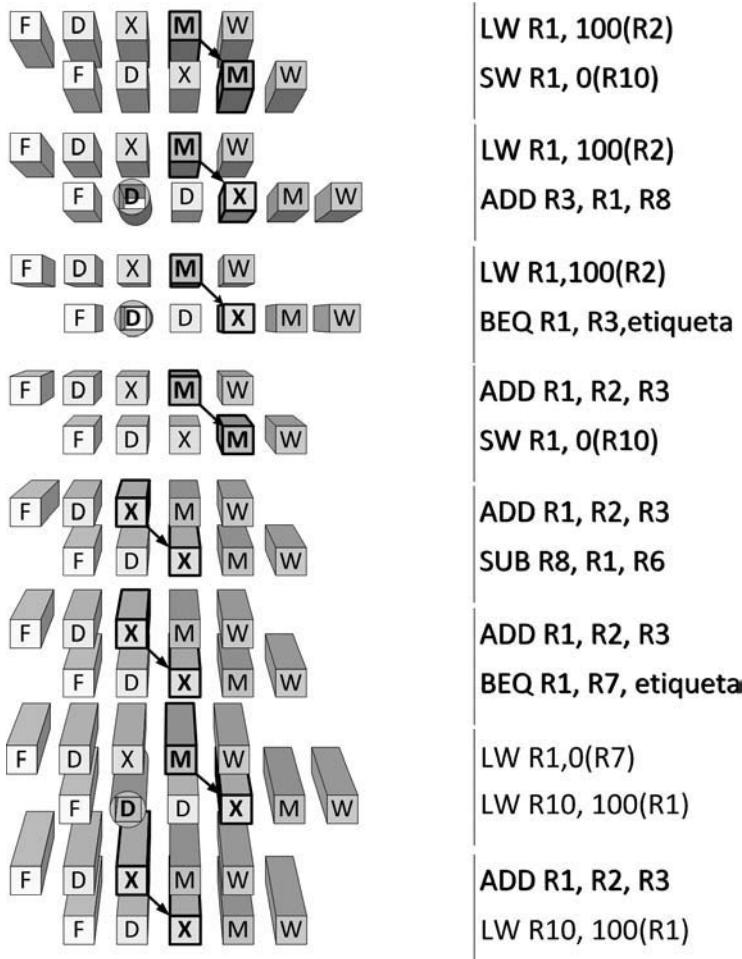


FIGURA 1.31

Todos los riesgos RAW posibles en el nanoMIPS y su resolución con adelantamiento.

- LW seguido de aritmético-lógica o de BEQ: El operando está disponible a la salida de la etapa M de la instrucción LW y se necesita a la entrada de la etapa X de la instrucción aritmético-lógica o BEQ.
- Aritmético-lógica seguida de SW: El operando está disponible a la salida de la etapa X de la instrucción aritmético-lógica y se necesita a la entrada de la etapa M de la instrucción SW.
- Aritmético-lógica seguida de aritmético-lógica o BEQ: El operando está disponible a la salida de la etapa X de la instrucción aritmético-lógica y se necesita a la entrada de la etapa X de la instrucción BEQ.
- LW seguido de acceso a memoria: El operando está disponible a la salida de la etapa M de la instrucción LW y se necesita a la entrada de la etapa X de la instrucción de acceso a memoria para calcular la dirección.
- Aritmético-lógica seguida de acceso a memoria: El operando está a la salida de la etapa X de la instrucción aritmético lógica y se necesita a la entrada de la etapa X de la instrucción de acceso a memoria para calcular la dirección.

Ejemplo 1.6

Comparación de una versión del nanoMIPS segmentado sin adelantamiento y con adelantamiento.

Supongamos que se desea ejecutar el siguiente código en el nanoMIPS:

1	LW	R1,0(R0)
2	LW	R2,100(R0)
3	ADD	R3,R1,R2
4	SUB	R4,R1,R2
5	AND	R5,R3,R4
6	ADD	R6,R1,R4
7	SW	R6,200(R0)
8	BFO	R6,R0,salt

Este es el diagrama de ejecución sin adelantamiento (cada columna es un ciclo de ejecución) y sin tener en cuenta de momento el riesgo de control provocado por la instrucción de salto:

	1	F	D	X	M	W									
	2		F	D	X	M	W								
INSTRUCCIONES	3			F	D	D	D	X	M	W					
	4				F	F	F	D	X	M	W				
	5						F	D	D	D	X	M	W		
	6							F	F	F	D	X	M	W	
	7									F	D	D	D	X	M
	8											F	F	F	D

Con adelantamiento tendríamos:

	1	F	D	X	M	W							
	2		F	D	X	M	W						
INSTRUCCIONES	3			F	D	D	X	M	W				
	4				F	F	D	X	M	W			
	5					F	D	X	M	W			
	6						F	D	X	M	W		
	7							F	D	X	M	W	
	8								F	D	X	M	W

Observamos que en un código tan sencillo y con tan pocas instrucciones ya se obtiene una gran ganancia gracias al adelantamiento, reduciendo las paradas por riesgos de datos (sombreadas parcialmente en gris) de 6 ciclos a 1.

Es decir:

$$CPI_{\text{sin adelantamiento}} = \frac{\text{ciclos}}{I} = \frac{18}{8} = 2.25$$

$$CPI_{con\ adelantamiento} = \frac{ciclos}{l} = \frac{13}{8} = 1.625$$

Por lo tanto gracias al adelantamiento se ha conseguido un speedup de 1.38 en la ejecución de este código.

Además siempre existe un tipo especial de dependencia que puede provocar riesgos RAW. Se puede comprender con la siguiente secuencia de instrucciones:

SW R5,0(R10)
LW R1400(R7)

Si se permitiera que la instrucción de carga se ejecutara antes que la de almacenamiento, por ejemplo, porque el valor de R5 no estuviera todavía disponible para ejecutar la instrucción de almacenamiento, podría ocurrir que al calcular la dirección de acceso a memoria de este almacenamiento coincidiera con la de la carga ($0+[R10]=400+[R7]$), y hubiera un riesgo RAW a través de la memoria no resuelto correctamente.

Pero en el caso del nanoMIPS segmentado que se ha diseñado hasta el momento, este riesgo no se produce, ya que todas las instrucciones se comienzan y se terminan en orden, pasando todas ellas por el mismo número de etapas, así que es seguro que la etapa M de la instrucción de store se realiza antes que la etapa M de la instrucción de load, por lo que la dependencia de datos a través de la memoria no supone ningún riesgo.

Aclarado esto, en el nanoMIPS, el origen de un adelantamiento siempre estará en el registro X/M o en el registro M/W y el destino de un adelantamiento siempre estará en el registro D/X o en el X/M. Siempre se procurará que el operando se adelante justo a la entrada de la etapa que lo necesita.

Las modificaciones que hay que realizar en la ruta de datos no son complicadas, la lógica de adelantamiento suele estar compuesta por nuevos multiplexores para controlar las entradas de las etapas que pueden necesitar un adelantamiento, y unidades hardware sencillas que permitan comunicar las salidas de las etapas que producen los resultados con las entradas de estos multiplexores. También es necesario un nuevo hardware de control que al decodificar la instrucción en la etapa D, detecte los adelantamientos y las paradas que van a ser necesarias para la ejecución de la instrucción. Ya que se observa que en algunos casos el adelantamiento no es suficiente para evitar las paradas en la ruta de datos, es típico el ejemplo de la combinación load y aritmético-lógica, que siempre obliga a realizar un ciclo de parada para resolver el riesgo.

En la figura 1.30 se muestra una ruta de datos del nanoMIPS simplificada con las modificaciones necesarias para poder adelantar operandos a la etapa X: los dos nuevos multiplexores a la entrada de la etapa X y la unidad de adelantamiento en la propia etapa X que controla estos multiplexores, además de la unidad de detección de adelantamientos y paradas que se incluye en la unidad de control en la etapa D y que no se aprecia en la figura. El resto de la ruta de datos no se modifica, al igual que las señales de control.

Las mismas modificaciones se deberían realizar para adelantar operandos a la etapa M (a las instrucciones de store) o a cualquier otra que según el repertorio de instrucciones y el diseño de la ruta de datos pudiera necesitar un adelantamiento (en la figura 1.31 se muestran todos los adelantamientos posibles).

La alternativa software para la resolución de riesgos de datos es responsabilidad del compilador, y en realidad no es una técnica que resuelva los riesgos sino que más bien los evita. Se trata de reordenar el código, de manera que la reordenación no afecte a los resultados (para ello a veces es necesario el renombrado de registros), y evitar así las combinaciones de instrucciones que provocan paradas en la ruta de datos (ejemplo en la figura 1.32). La norma general es separar lo más posible las instrucciones con dependencia RAW que generan el riesgo.

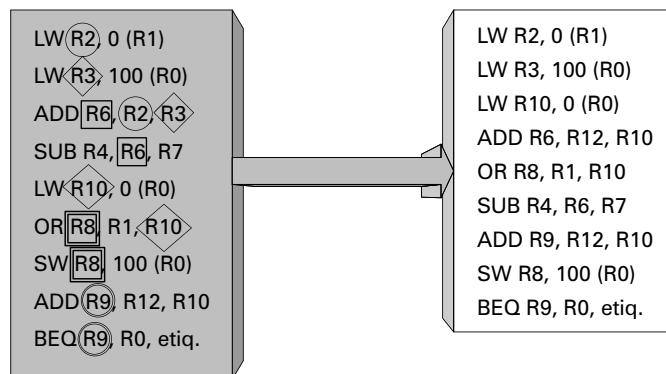


FIGURA 1.32

Ejemplo de reordenación de un fragmento de código realizada por el compilador para separar lo más posible las instrucciones con dependencias sin modificar el resultado.

1.5.2.3. RESOLUCIÓN DE RIESGOS DE CONTROL

De nuevo existen dos tipos de técnicas para resolver estos riesgos, una hardware y otra software. Además, se puede modificar ligeramente la ruta de datos del procesador, para que, en el caso que no haya otra solución además de la parada de la ruta de datos, por lo menos ésta sea lo más corta posible.

Se ha mostrado que al resolver un riesgo de control en el nanoMIPS mediante paradas, son necesarios tres ciclos de espera hasta que la instrucción de salto BEQ carga el valor adecuado en el PC y se puede continuar con la ejecución.

Existe una sencilla modificación de la ruta de datos del nanoMIPS que puede reducir esta parada a un único ciclo. La idea es sencilla: adelantar tanto como sea posible la resolución de los saltos. Como es en la etapa D en la que se decodifica la instrucción y se sabe que es un salto, en esta misma etapa se añade el hardware necesario para evaluar la condición del salto (un restador, el módulo Zero?) y para calcular la dirección destino del salto (un sumador). La señal de control Branch pasa a generarse directamente para ser utilizada en la etapa D, y la señal de control ALUSrcA para la etapa X ya no es necesaria. Además, si se realiza esta modificación en la ruta de datos, aparece un nuevo adelantamiento a la etapa D (cuando hay un riesgo de datos RAW con uno de los registros necesarios para la evaluación de la condición del salto), lo que exige añadir el hardware necesario para realizar los adelantamientos también en esta etapa si se utiliza esta técnica para resolver los riesgos de datos.

Con la ruta de datos modificada del nanoMIPS (figura 1.33), sólo es necesario un ciclo de parada para resolver el riesgo de control.

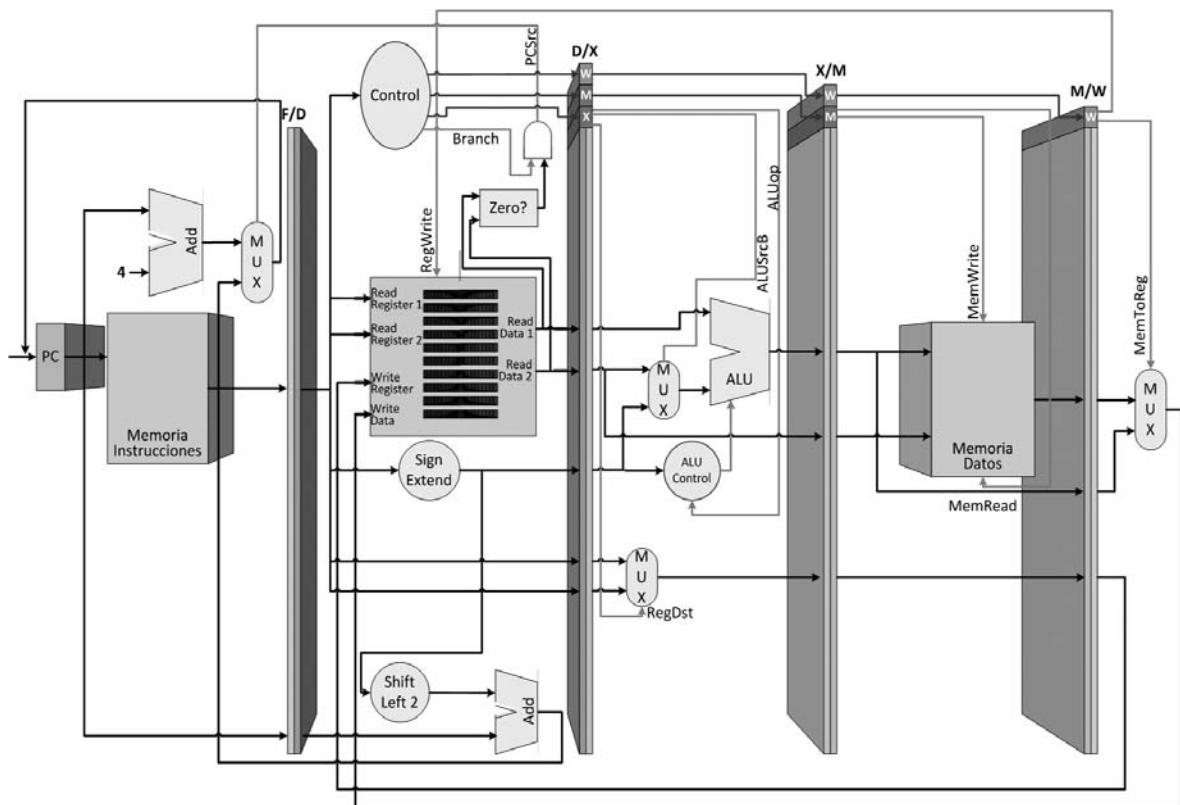


FIGURA 1.33

Ruta de datos del nanoMIPS modificada para que las paradas por riesgo de control sean de un ciclo en lugar de tres.

Aún con esta modificación, cada vez que se ejecuta una instrucción de salto, el procesador tiene que parar un ciclo (figura 1.34), y las instrucciones de salto son muy frecuentes, por lo que estas paradas empeoran significativamente el rendimiento de la segmentación.

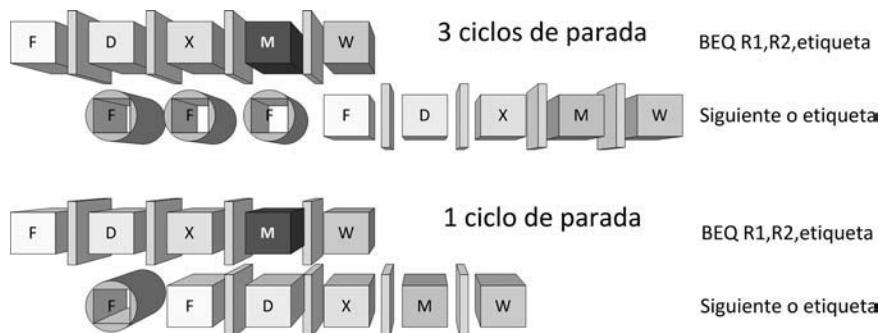


FIGURA 1.34

Comparación entre el nanoMIPS y el nanoMIPS modificado para que las paradas por riesgo de control sean de un ciclo en lugar de tres.

La técnica hardware que intenta evitar esta parada se basa en realizar una predicción para el salto. Las predicciones siempre consiguen una reducción de la penalización por salto cuando aciertan. Las predicciones más sencillas son las estáticas, en las que el procesador está diseñado para predecir en todos los casos que el salto se va a tomar o que el salto no se va a tomar. Es decir, la predicción es siempre la misma para cualquier salto.

Con la predicción de salto no tomado en el nanoMIPS se consigue que la penalización sea de un ciclo para saltos tomados y de ningún ciclo para saltos que no se toman (figura 1.35). Mientras se decodifica y se hace efectivo el salto, se va buscando la siguiente instrucción. Si finalmente el salto no se toma y la predicción se ha acertado, se continúa normalmente y no se ha perdido ningún ciclo. Si por el contrario el salto se toma y la predicción falla, se busca la instrucción destino y se ha perdido un ciclo haciendo un trabajo que no era necesario, pero que se hubiera perdido igualmente si no se hubiera realizado la predicción.

Es decir, gracias a la predicción de salto no tomado sólo se tiene penalización por salto cuando los saltos se toman (la predicción falla).



FIGURA 1.35

Predicción de salto no tomado en el nanoMIPS.

La implementación de esta técnica es sencilla, en la ruta de datos no hay que hacer prácticamente ninguna modificación y la unidad de control sólo tiene que añadir la lógica necesaria para continuar con el camino de salto no tomado si la predicción acierta, o para desechar la instrucción que se había buscado y comenzar el camino de salto tomado, buscando la instrucción destino de salto, si la predicción falla. Obviamente, si no se incorporan en la ruta de datos mecanismos para solucionarlo, no se debe dejar que las instrucciones predichas modifiquen el estado del procesador y/o la memoria, es decir, se debe parar su ejecución antes de que lleguen a las etapas en las que se escribe en memoria y en el banco de registros. Pero en el nanoMIPS esto no supone ningún problema porque si el salto se resuelve durante la etapa D, la instrucción predicha sólo realiza la etapa F. Incluso si la ruta de datos no se modifica y el salto se resuelve en la etapa M, la instrucción predicha que más avanza sólo llega a la etapa X, por lo que no se escribe en memoria o en registros.

En la arquitectura del nanoMIPS y en otras similares no tiene sentido utilizar la predicción de salto tomado porque supone una penalización de un ciclo en todos los casos, independientemente de si se acierta la predicción o no. Esto se debe a que no se conoce la dirección destino de salto hasta la etapa D, por lo que no se puede buscar la instrucción que se predice que se va a ejecutar hasta que el salto no se resuelve realmente.

Pero en otros tipos de arquitecturas en las que se conozca la dirección destino de salto antes de conocer si el salto se toma o no, este tipo de predicción sí que puede mejorar el rendimiento del procesador al igual que la de salto no tomado.

Ejemplo 1.7

Predicción estática de salto en una arquitectura segmentada de 6 etapas.

Supongamos que tenemos un procesador segmentado en 6 etapas: F, D, IS, X, M y W. En este procesador las instrucciones de salto condicional calculan la dirección destino de salto en la etapa D y evalúan la condición y cargan el nuevo PC en la etapa X. Vamos a comparar el rendimiento de la predicción estática de salto no tomado con la de salto tomado.

En el caso de la predicción estática, si ésta es de salto no tomado, siempre se buscan las instrucciones a continuación de la de salto hasta que el salto se resuelve:

Salto NT	F	D	IS	X	M	W
PC+4		F	D	IS		
PC+8			F	D		
PC+12				F		

Y como es un salto no tomado, cuando se resuelve el salto (indicado con la flecha vertical en el diagrama), simplemente se continúa con la ejecución de las instrucciones que se habían buscado después del salto.

En el caso de que sea un salto que se toma:

Salto T	F	D	IS	X	M	W
PC+4		F	D	IS		
PC+8			F	D		
PC+12				F		
Destino de salto					F	

Y la penalización es de 3 ciclos, la misma que se hubiera producido sin realizar la predicción con motivo del riesgo de control.

I. CONCEPTOS BÁSICOS DE PROCESADORES

Si se hace predicción de salto tomado, tenemos que buscar siempre a continuación de la instrucción de salto la instrucción destino de salto y las siguientes. Pero para comenzar estas búsquedas, es necesario conocer la dirección destino de salto, que en este caso se calcula en la etapa D:

Salto NT	F	D	IS	X	M	W
Destino de salto		—	↓ F	Đ		
Destino+4				↓ F		
PCsalto+4					↓ F	

Salto T	F	D	IS	X	M	W
Destino de salto		—	↓ F	D		
Destino+4				↓ F		
Destino+8					↓	

Es decir, hay una penalización de 3 ciclos en los saltos no tomados porque se falla la predicción y es como si no se hubiera hecho, y de 1 ciclo en el caso de los saltos tomados (acierto de la predicción) porque se tarda un ciclo en tener disponible la dirección destino de salto para poder hacer esta predicción.

En resumen:

Tipo de predicción	Penalización en acierto	Penalización en fallo
Salto no tomado	0	3
Salto tomado	1	3

Supongamos que en media, con los códigos ejecutados en este procesador el 75% de los saltos se toman y el 25% de los saltos no. Entonces, para la predicción de salto no tomado tenemos una penalización media por riesgo de control de $0.75 \cdot 3 = 2.25$ ciclos. Sin embargo, utilizando la predicción de salto tomado la penalización media es de $0.75 \cdot 1 + 0.25 \cdot 3 = 1.5$ ciclos, por lo que se consiguen mejores resultados con este tipo de predicción aunque haya 1 ciclo de penalización en los aciertos.

Existen otro tipo de predicciones estáticas algo más sofisticadas, que en lugar de realizar siempre la misma predicción, tienen en cuenta características del salto para reducir la tasa de fallos mediante heurísticas. En muchos de estos casos la predicción ya no la realiza el hardware exclusivamente sino que se involucra también al compilador. Las predicciones más utilizadas se basan en:

- El opcode de la instrucción de salto, ya que hay estudios que demuestran que algunos tipos de saltos se toman con más probabilidad que otros (esto tiene que ver con el estilo de programación de los desarrolladores y con el diseño de los compiladores y cómo traducen ciertas estructuras de alto nivel a lenguaje ensamblador).
- La condición de la instrucción de salto (si un registro es igual a 0 o diferente de 0, si dos registros son iguales o diferentes, etc), ya que hay estudios que demuestran que algunas condiciones se cumplen en muchas más ocasiones que otras (por los mismos motivos mencionados para el opcode).
- La longitud y la dirección del desplazamiento del salto, ya que esto permite detectar los saltos al final de los bucles fácilmente. La norma general suele ser que los saltos hacia atrás (los que están al final del bucle) se predicen como tomados, y los saltos hacia delante, se predicen como no tomados.
- La estructura del programa, proporcionando un interfaz al compilador para predecir los saltos basándose en una comprensión a alto nivel de la estructura de la aplicación. Un ejemplo son las denominadas heurísticas de Ball y Larus, un conjunto de nueve reglas que permiten tener en

cuenta los lenguajes de programación habituales y los comportamientos más usuales de las aplicaciones para que el compilador pueda predecir con la información de alto nivel de la estructura del programa si los saltos condicionales se van a tomar o no.

En cuanto a la técnica software para la resolución de los riesgos de control, de nuevo es responsabilidad del compilador y se suele denominar técnica de salto retardado o de relleno de ranura.

Con esta técnica se evita la penalización por salto introduciendo justo a continuación de la instrucción de salto, instrucciones que se van a ejecutar en cualquier caso. Es decir, la idea básica es conseguir que el procesador realice trabajo útil mientras se resuelve el salto.

Se denomina ranura a la penalización por salto, así que suponiendo que tenemos el nanoMIPS modificado para resolver los saltos en la etapa D, la ranura es de 1 ciclo y es necesario que el compilador busque una instrucción para llenarla. Esta instrucción no puede ser otro salto y puede ser una instrucción anterior a la de salto, la instrucción destino del salto o una instrucción del camino que se sigue cuando el salto no se toma.

Si la ruta de datos no se hubiera modificado, la ranura sería de 3 ciclos y en este caso sería bastante más complicado encontrar tres instrucciones independientes del salto que permitieran realizar trabajo útil mientras éste se resuelve.

En la figura 1.36 se muestran las tres alternativas típicas para llenar la ranura de salto con una instrucción en el nanoMIPS. Hay que tener en cuenta que la ranura siempre puede llenarse con una instrucción anterior al salto y que no tenga dependencias con él. Sin embargo, no siempre se puede llenar la ranura con la instrucción destino del salto o con una instrucción del camino que se sigue cuando el salto no se toma, porque en este caso se está ejecutando en todos los casos una instrucción que no siempre se ejecutaría, y esto podría llegar a modificar los resultados de la ejecución.

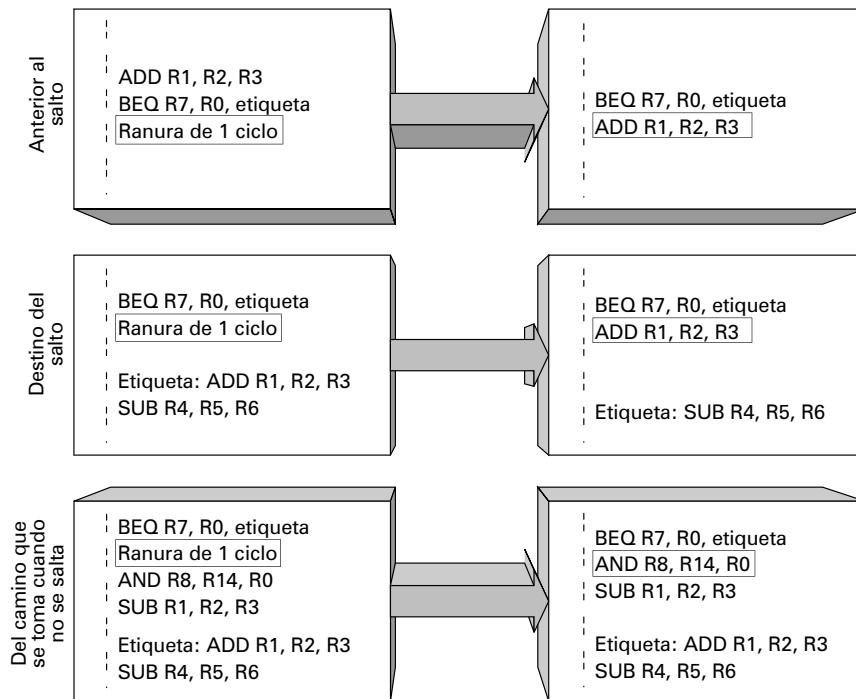


FIGURA 1.36

Las tres alternativas para el salto retardado suponiendo ranura de un ciclo.

En algunos procesadores existe la posibilidad de anular la ejecución de la instrucción que está rellenando la ranura si se observa, que con el camino que finalmente ha tomado el salto, esa instrucción no debería finalizar para no llevar a resultados erróneos. Es decir, se puede plantear hacer lo mismo que con las instrucciones predichas cuando la predicción de salto ha fallado. Pero esta técnica implica un nuevo aumento de complejidad en la unidad de control del procesador, a la que se debe informar de qué instrucciones van rellenando la ranura (el salto retardado es una técnica muy relacionada con el compilador), por lo que no suele utilizarse mucho.

Ejemplo 1.8

Utilización de la técnica de salto retardado.

Vamos a llenar la ranura de salto para el siguiente código con todas las técnicas que sea posible, suponiendo que el procesador es un nanoMIPS con adelantamiento y con ranura de salto de 1 ciclo (los saltos se resuelven en la etapa D)

1	AND	R6,R4,R5
2	LW	R1,0(R10)
3	LW	R2,100(R10)
4	BEQ	R2,R0,salto
ranura de salto		
5	SUB	R8,R1,R2
6	ADD	R2,R8,R2
7	salto:	SUB R9,R11,R1
8	ADD	R12,R9,R1

Comencemos por intentar llenar la ranura de salto con la técnica que sabemos que se puede utilizar siempre, utilizar una instrucción anterior al salto.

Observando las instrucciones que se encuentran antes del BEQ (todas ellas se ejecutan independientemente de si el salto se toma o no) se suele comenzar por las que están inmediatamente antes del salto:

- La instrucción 3 no puede moverse después del salto ya que la condición del salto depende de ella.
- La instrucción 2 puede moverse a la ranura de salto, pero si después el salto no se toma, estamos creando un riesgo LW seguido de aritmético-lógica, y a pesar del adelantamiento esto supone un ciclo de parada. Así que lo que ganaríamos por un lado lo perderíamos por otro.
- La instrucción 1 puede moverse a la ranura de salto sin problemas, el resultado del código sería exactamente el mismo.

Por lo tanto el código quedaría:

2	LW	R1,0(R10)
3	LW	R2,100(R10)
4	BEQ	R2,R0,salto
1	AND	R6,R4,R5
5	SUB	R8,R1,R2
6	ADD	R2,R8,R2
7	salto:	SUB R9,R11,R1
8	ADD	R12,R9,R1

Y evitariamos de esta manera perder un ciclo por el riesgo de control, la instrucción AND que rellena la ranura se ejecuta siempre completa pase lo que pase con el salto.

Ahora podemos intentar llenar la ranura con la instrucción destino del salto (marcada con la etiqueta "salto" en este caso). Tendríamos:

1	AND	R6,R4,R5
2	LW	R1,0(R10)
3	LW	R2,100(R10)
4	BEQ	R2,R0,salto+4
7	SUB	R9,R11,R1
5	SUB	R8,R1,R2
6	ADD	R2,R8,R2
8	salto+4: ADD	R12,R9,R1

Observamos que en este caso hay que modificar ligeramente el código, ya que la etiqueta de salto se tiene que mover a la siguiente instrucción a la destino de salto (sumando 4 porque las instrucciones del nanoMIPS siempre ocupan 4 bytes). Esto es así porque la instrucción destino de salto se ejecuta siempre completa durante la ranura de salto.

En este caso puede utilizarse la técnica de instrucción destino de salto porque:

- Si el salto se toma, la instrucción 7 tenía que ejecutarse igualmente.
- Si el salto no se toma, se ha ejecutado una resta con la instrucción 7 que no debería haberse ejecutado nunca. Pero lo que se ha modificado es el valor del registro R9, que como no se utiliza después en el código, no tiene efectos en los resultados obtenidos.

Por último, vamos a intentar utilizar una instrucción del camino que se sigue cuando el salto no se toma:

1	AND	R6,R4,R5
2	LW	R1,0(R10)
3	LW	R2,100(R10)
4	BEQ	R2,R0,salto
5	SUB	R8,R1,R2
6	ADD	R2,R8,R2
7	salto: SUB	R9,R11,R1
8	ADD	R12,R9,R1

En este caso la instrucción 5 rellena la ranura y se ejecuta completamente en todos los casos, ocurra lo que ocurra con el salto. Podemos hacer una reflexión similar a la que hacíamos en el caso anterior:

- Si el salto no se toma, la instrucción 5 tenía que ejecutarse igualmente.
- Si el salto se toma, se ha ejecutado una resta con la instrucción 5 que no debería haberse ejecutado nunca. Pero lo que se ha modificado es el valor del registro R8, que como no se utiliza después en el código, no tiene efectos en los resultados obtenidos.

Esta situación tan ideal no suele producirse muy a menudo. Veamos como con un código muy similar, llenar la ranura puede ser una tarea mucho más complicada.

1	AND	R1,R4,R5
2	LW	R1,0(R10)
3	LW	R2,100(R10)
4	BEQ	R2,R1,salto

		ranura de salto
5	SUB	R8,R1,R2
6	ADD	R2,R8,R9
7	salto:	SUB
8	ADD	R9,R8,R1
		R12,R9,R1

Inicialmente, debido a las dependencias entre instrucciones, ninguna de las tres posibilidades permite llenar la ranura. Habría que estudiar primero si estas dependencias son reales o sólo a través de los nombres de los registros para ver si con un renombrado sería posible evitarlas, pero tal y como está el código:

- Ni las instrucciones 1, 2 ó 3, anteriores al salto, permiten llenar la ranura de salto por las dependencias que existen entre ellas o entre ellas y el salto.
- La instrucción 5 no puede llenar la ranura porque modifica el contenido del registro R8, y si el salto se toma, este registro se utiliza. La 6 tampoco, ya que depende del resultado de la 5 y no puede ejecutarse antes que ella.
- La instrucción 7 no puede llenar la ranura porque modifica el contenido de R9 y, si el salto no se toma, este registro se utiliza.

Lo habitual es que nos encontremos situaciones intermedias, no en los dos extremos mostrados en este ejemplo, pero son este tipo de consideraciones las que el compilador y/o el desarrollador tendrán que realizar para utilizar esta técnica de resolución de los riesgos de control.

Ejemplo 1.9

Ejecución de un código en el nanoMIPS con resolución de riesgos mediante paradas.

Supongamos, por comodidad, que se añaden al repertorio del nanoMIPS la instrucción ADDI para poder realizar sumas con operandos inmediatos (etiquetados con el modificador # en el pseudocódigo) y la instrucción de salto BNE. En esta arquitectura se ejecuta el siguiente código:

1	ADDI	R10,R0,#0
2	ADDI	R11,R0,#4000
3	bucle:	LW
4		R1,A(R10)
5	BEQ	R1,R0,salto
6	LW	R2,B(R10)
7	ADD	R3,R1,R2
8	SW	R1,B(R10)
9	SW	R3,A(R10)
10	salto:	ADDI
		R10,R10,#4
		BNE
		R10,R11,bucle

Donde A y B son las direcciones en las que se almacenan los elementos iniciales de los vectores de enteros A y B con los que se está trabajando. El 75% de los elementos del vector A son diferentes de 0, con esta información podemos calcular el CPI que se obtiene al ejecutar este código sin más herramientas para resolver los riesgos que aparezcan que las paradas. Observamos por el valor inicial de R11 que el bucle se repite 1000 veces.

Veamos los diagramas de ejecución cuando se salta en el primer salto y cuando no se salta, es decir, sepáremos los distintos casos de ejecución. Supongamos que son iteraciones intermedias del bucle, ya que la primera es diferente con la ejecución de las instrucciones 1 y 2, un riesgo de datos RAW que surge entre las instrucciones 1 y 3, y los cuatro ciclos que tarda en llenarse la ruta de datos completamente.

Caso 1: Elemento de A igual a 0

3	F	D	X	M	W										
4		F	D	D	D	X	M	W							
9					—	F	D	X	M	W					
10							F	D	D	D	X	M	W		
											—				

La última instrucción que se ejecuta puede ser la siguiente al salto o el destino de salto, pero de cualquier forma siempre hay una parada por riesgo de control.

Caso 2: Elemento de A distinto de 0

3	F	D	X	M	W										
4		F	D	D	D	X	M	W							
5				—	F	D	X	M	W						
6						F	D	D	D	X	M	W			
7							F	F	F	D	X	M	W		
8										F	D	D	X	M	W
9										F	F	D	X	M	W
10											F	D	D	D	X
															—

Igual que en el caso anterior, la última instrucción que se ejecuta puede ser la siguiente al salto o el destino de salto, pero de cualquier forma siempre hay una parada por riesgo de control.

Tenemos, para las iteraciones intermedias del bucle:

Caso 1 (Elemento de A igual a 0)

- Instrucciones = 4
- Ciclos = 4 (un ciclo por instrucción) + 0 paradas por riesgos estructurales + 4 paradas por riesgos de datos + 2 paradas por riesgos de control = 10

Caso 2 (Elemento de A distinto de 0)

- Instrucciones = 8
- Ciclos = 8 (un ciclo por instrucción) + 0 paradas por riesgos estructurales + 7 paradas por riesgos de datos + 2 paradas por riesgos de control = 17

Y para la ejecución de las instrucciones 1 y 2 en la primera iteración, teniendo en cuenta los 4 ciclos que la ruta de datos tarda en llenarse y el riesgo RAW entre las instrucciones 1 y 3:

- Instrucciones = 2
- Ciclos = 2 (un ciclo por instrucción) + 4 ciclos de llenado + 0 paradas por riesgos estructurales + 1 parada por riesgo de datos + 0 paradas por riesgos de control = 7

Como sabemos el número de veces que tenemos cada caso ($0.25 \cdot 1000 = 250$ iteraciones son el caso 1, $0.75 \cdot 1000 = 750$ iteraciones son el caso 2):

$$CPI = \frac{\text{ciclos}}{\text{instrucciones}}$$

$$CPI = \frac{\text{ciclos extras 1ª iteración} + 250 \cdot \text{ciclos caso 1} + 750 \cdot \text{ciclos caso 2}}{\text{instrucciones extras 1ª iteración} + 250 \cdot \text{instrucciones caso 1} + 750 \cdot \text{instrucciones caso 2}}$$

$$= \frac{7 + 250 \cdot 10 + 750 \cdot 17}{2 + 250 \cdot 4 + 750 \cdot 8} = 2.18$$

Lejos del CPI ideal de 1 que se busca con la segmentación.

Ejemplo 1.10

Ejecución de un código en el nanoMIPS con resolución de riesgos mediante adelantamiento y predicción de salto no tomado (técnicas hardware).

Supongamos que ejecutamos exactamente el mismo código pero con las técnicas hardware que permiten reducir el número de paradas del ejemplo anterior:

1						ADDI	R10,R0,#0
2						ADDI	R11,R0,#4000
3		bucle:		LW	R1,A(R10)		
4				BEQ	R1,R0,salto		
5				LW	R2,B(R10)		
6				ADD	R3,R1,R2		
7				SW	R1,B(R10)		
8				SW	R3,A(R10)		
9		salto:		ADDI	R10,R10,#4		
10				BNE	R10,R11,bucle		

Caso 1: Elemento de A igual a 0

3	F	D	X	M	W											
4		F	D	D	D	X	M	W								
5						F										
9							F	D	X	M	D					
10								F	D	D	X	M	W			
									F							

Observamos que a pesar de tener disponible el adelantamiento, en el riesgo RAW de las instrucciones 3 y 4 éste debería hacerse entre la salida de la fase M y la entrada de la fase D (porque es en esta fase en la que se resuelve el salto), por lo que no evitamos las 2 paradas por riesgo de datos.

En el RAW entre las instrucciones 9 y 10 conseguimos ahorrarnos una de las paradas, aunque la otra es necesaria para poder adelantar el operando de la salida de la etapa X a la entrada de la etapa D.

La predicción de salto falla siempre para el salto de la instrucción 4 (en este caso de ejecución siempre se salta), y en todas las iteraciones menos en la última del bucle para el salto de la instrucción 10.

En ambos saltos se busca siempre la instrucción siguiente y cuando se comprueba que la predicción ha fallado, se desecha lo que se había buscado.

Caso 2: Elemento de A distinto de 0

3	F	D	X	M	W											
4		F	D	D	D	X	M	W								
5			F	F	F	D	X	M	W							
6						F	D	D	X	M	W					
7							F	F	D	X	M	W				
8								F	D	X	M	W				
9									F	D	X	M	W			
10										F	D	D	X	M	W	
										F						

En este caso tenemos los mismos riesgos de datos que el caso 1 excepto el RAW entre las instrucciones 5 y 6, en el que el adelantamiento evita uno de los ciclos de parada pero no el otro. Y hay otro riesgo RAW entre las instrucciones 6 y 8, pero en este caso el adelantamiento evita todas las paradas.

En lo que se refiere a los riesgos de control, la predicción del primer salto siempre se acierta, lo que evita la parada por riesgo de control en todos los casos. La predicción del salto de la instrucción 10 siempre falla excepto en la última iteración del bucle.

Tenemos, para las iteraciones intermedias del bucle:

Caso 1 (Elemento de A igual a 0)

- Instrucciones = 4.
- Ciclos = 4 (un ciclo por instrucción) + 0 paradas por riesgos estructurales + 3 paradas por riesgos de datos + 2 ciclos perdidos por fallo de predicción = 9.

Caso 2 (elemento de A distinto a 0).

- Instrucciones = 8.
- Ciclos = 8 (un ciclo por instrucción) + 0 paradas por riesgos estructurales + 4 paradas por riesgos de datos + 1 ciclo perdido por fallo de predicción = 13.

Y para la ejecución de las instrucciones 1 y 2 en la primera iteración, teniendo en cuenta los 4 ciclos que la ruta de datos tarda en llenarse y que el riesgo RAW entre las instrucciones 1 y 3 se resuelve completamente con un adelantamiento:

- Instrucciones = 2.
- Ciclos = 2 (un ciclo por instrucción) + 4 ciclos de llenado + 0 paradas por riesgos estructurales + 0 paradas por riesgo de datos + 0 paradas por riesgos de control = 6.

En cuanto a la última iteración del bucle, sólo en ese caso, la predicción del salto de la instrucción 10 acierta y nos ahorraremos 1 ciclo, que no hace falta que tengamos en cuenta para el cálculo del CPI ya que no se va a notar nada en el resultado final.

Como sabemos el número de veces que tenemos cada caso:

$$CPI = \frac{\text{ciclos}}{\text{instrucciones}}$$

$$CPI = \frac{\text{ciclos extras 1ª iteración} + 250 \cdot \text{ciclos caso 1} + 750 \cdot \text{ciclos caso 2}}{\text{instrucciones extras 1ª iteración} + 250 \cdot \text{instrucciones caso 1} + 750 \cdot \text{instrucciones caso 2}}$$

$$= \frac{6 + 250 \cdot 9 + 750 \cdot 13}{2 + 250 \cdot 4 + 750 \cdot 8} = 1.71$$

Es decir, gracias a las técnicas hardware para la resolución de riesgos se ha conseguido un speedup de $2.18/1.71 = 1.27$ en la ejecución de este código.

Y con la ayuda del compilador, mediante reordenación de instrucciones y salto retardado, todavía podría intentarse incrementar algo más esta mejora.

1.5.3. Procesadores segmentados multifuncionales

Hasta el momento se han estudiado técnicas de diseño para procesadores segmentados que sólo incorporan en su repertorio de instrucciones operaciones con números enteros y que por lo tanto, tienen una única unidad funcional, la ALU de enteros.

Si se plantea la posibilidad de añadir al repertorio de instrucciones del nanoMIPS instrucciones que operen con números en coma flotante, por ejemplo, la suma, la resta, la multiplicación y la división para precisión doble (ADD.D, SUB.D, MUL.D y DIV.D respectivamente), así como las instrucciones que permiten realizar la carga y el almacenamiento de los operandos en coma flotante (L.D y S.D), aparecen nuevas dificultades que deben ser solucionadas.

La principal es que no se puede pretender fijar la duración del ciclo de reloj del procesador para que en la etapa X se pueda completar una división en coma flotante completa, operación con una latencia importante sea cual sea la implementación hardware de la unidad funcional que la realice.

La ruta de datos que se ha mostrado para el nanoMIPS segmentado hasta el momento, se trata de una ruta de datos lineal o de única función, en la que una determinada instrucción está obligada a pasar

en orden por todas las etapas de la ruta de datos y a tardar un ciclo de reloj en completar cada una de ellas. Esto es así porque el repertorio de instrucciones del procesador sólo necesita una ALU de enteros para realizar todas las operaciones incluidas en él.

Cuando el repertorio de instrucciones de los procesadores es más complejo, la ruta de datos suele convertirse en multifuncional (también llamada a veces no lineal o bifurcada), de manera que algunas etapas puedan tardar más de un ciclo en ejecutarse o puedan ser reutilizadas por una misma instrucción. En los diseños más complicados, incluso se permite que una misma instrucción pueda estar ejecutando más de una misma etapa al mismo tiempo o que se pueda modificar el orden de ejecución de las etapas.

En una primera aproximación, para realizar operaciones en coma flotante en la ruta de datos del nanoMIPS, se puede permitir que estas instrucciones repitan la etapa X tantas veces como sea necesario (figura 1.37). Se utilizan varias unidades funcionales de coma flotante de manera que la ALU de enteros se encarga de las instrucciones de acceso a memoria, de las instrucciones aritmético-lógicas con enteros y de las instrucciones de salto. Para las operaciones en coma flotante se añaden un sumador/restador en coma flotante, un multiplicador y un divisor. Esto implica que se deben incluir nuevos registros de segmentación y que se añaden nuevas conexiones entre todos los registros para implementar los nuevos adelantamientos que sean necesarios.

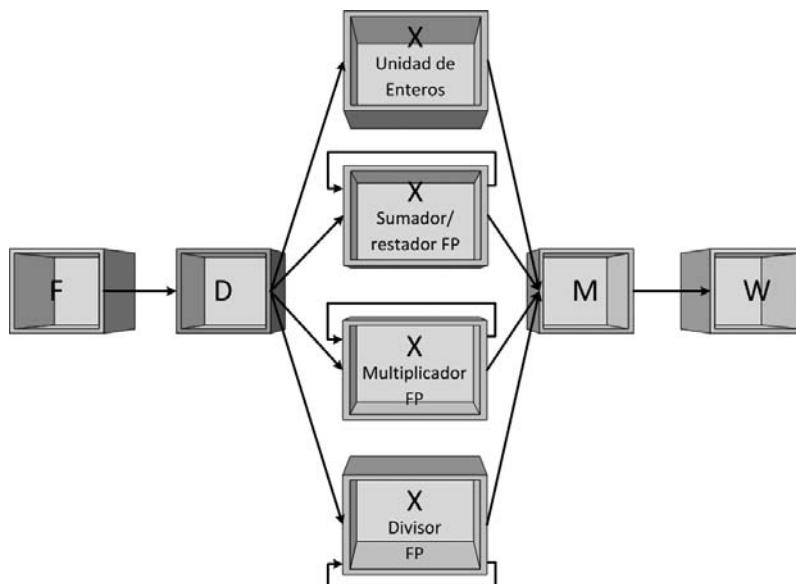


FIGURA 1.37

Fases de ejecución de instrucciones en el nanoMIPS multifuncional.

Este tipo de diseño es sencillo, pero plantea un problema de rendimiento con los riesgos estructurales por las unidades funcionales en coma flotante. Si se tiene que el sumador/restador tiene una latencia de 4 ciclos, el multiplicador de 10 ciclos y el divisor de 20 ciclos, ¿qué ocurre si en un código aparecen dos sumas en coma flotante seguidas? Obviamente, la segunda suma tendrá que esperar 4 ciclos (en lugar de 1) a que termine la primera:

ADD.D	F1,F2,F3	F	D	A	A	A	A	M	W		
ADD.D	F4,F5,F6	F	D	D	D	D	A	A	A	M	W

Para evitar todas estas paradas por riesgos estructurales debidos a las unidades funcionales, se suele optar por incluir unidades funcionales de coma flotante segmentadas dentro de la ruta de datos (figu-

ra 1.38). De esta manera, la segunda suma podría empezar a ejecutar en el siguiente ciclo que la primera, y lo mismo ocurriría con la multiplicación. Si se diseña así el procesador, la etapa X para las instrucciones enteras se convierte en cuatro etapas diferentes (A1, A2, A3 y A4) para la suma en coma flotante o en 10 para la multiplicación (M1, M2, etc).

ADD.D	F1,F2,F3	F	D	A1	A2	A3	A4	M	W
ADD.D	F4,F5,F6	F	D	A1	A2	A3	A4	M	W

La única diferencia sustancial entre unas rutas de datos y otras suele estar en la segmentación del divisor, que no siempre se incorpora por tratarse de una operación que es poco frecuente, por lo que el riesgo estructural por esta unidad es muy raro que se produzca.

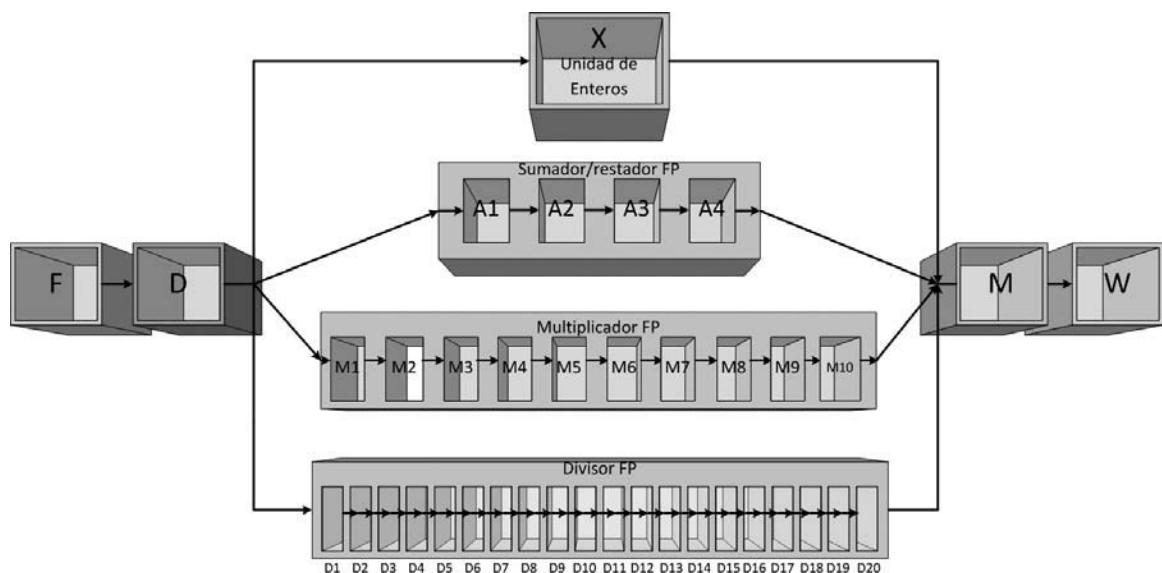


FIGURA 1.38

Fases de ejecución de instrucciones en el nanoMIPS multifuncional con unidades funcionales segmentadas.

En una ruta de datos multifuncional siguen apareciendo los riesgos de datos RAW y los de control, pero también aparecen nuevos riesgos que habrá que solucionar:

- Puede haber riesgos estructurales con las unidades funcionales que no estén segmentadas.
- Además las instrucciones tienen distintas duraciones por lo que puede que se necesite más de una escritura en memoria o en los registros en un determinado ciclo de reloj. Es decir, aparecen los riesgos estructurales por el banco de registros y por la memoria de datos.
- Hay que comprobar si las dependencias de datos RAW a través de la memoria pueden convertirse en este tipo de procesador en riesgos.
- Por último, habrá que comprobar también si las dependencias WAR y WAW pueden originar riesgos, porque en la ruta de datos lineal estos riesgos no eran posibles debido a que las instrucciones pasaban obligatoriamente por las mismas etapas, con la misma duración y en el mismo orden. Pero esto no es así en las rutas multifuncionales.

Los riesgos estructurales por las unidades funcionales que no están segmentadas se resuelven siempre mediante paradas. En el caso de los riesgos estructurales por la memoria y por el banco de registros, existe

la posibilidad de incorporar a la ruta de datos del procesador memorias y bancos de registros multipuerto, que puedan ser accedidos por más de una instrucción en un mismo ciclo. Pero esta solución suele ser cara, así que en muchos casos, por ejemplo, en el del nanoMIPS, se opta por identificar los posibles riesgos de este tipo en la etapa D de las instrucciones y resolverlos con paradas cuando sea necesario.

Una vez discutidas las posibles soluciones para los nuevos riesgos estructurales de los procesadores segmentados multifuncionales, siempre hay que evaluar la posibilidad de que aparezcan en este tipo de procesador nuevos riesgos de datos debidos a las dependencias RAW a través de la memoria, WAR y WAW.

Tomando el ejemplo del nanoMIPS, se observa que todas las instrucciones siguen realizando las etapas de F, D, M y W en el mismo orden y con la misma duración, lo que ha cambiado es que cada tipo de instrucción puede tardar un número diferente de ciclos en completar la fase de ejecución en la correspondiente unidad funcional. Esto hace que las dependencias WAR y las dependencias RAW a través de la memoria sigan sin suponer un riesgo (en todas las combinaciones posibles de instrucciones las etapas que deben estar ordenadas de una determinada manera para que no se produzca un riesgo lo están), pero las WAW sí que pueden provocar riesgos en este caso.

Por ejemplo, en la siguiente combinación de instrucciones:

ADD.D	F1,F2,F3	F	D	A1	A2	A3	A4	M	W
L.D	F1,0(R0)	F	D	X	M		W		

Se puede observar que si no se resuelve correctamente el riesgo WAW, el valor que quedará escrito en el registro F1 es el resultado de la suma en lugar del resultado de la lectura de memoria.

Algunas arquitecturas incorporan el hardware necesario para evitar que la escritura realizada por la suma se haga efectiva en el banco de registros. Pero en el caso del nanoMIPS se opta por una solución similar a la que se adopta para otros riesgos: se detecta el riesgo WAW en la fase D de la instrucción de lectura de memoria y se realizan paradas para que su escritura tenga lugar justo a continuación de la que realiza la instrucción de suma:

ADD.D	F1,F2,F3	F	D	A1	A2	A3	A4	M	W	
L.D	F1,0(R0)	F	D	D	D	D	D	X	M	W

Tanto los nuevos riesgos estructurales como estos nuevos riesgos WAW se pueden solucionar mediante paradas ya que la probabilidad de ocurrencia de todos ellos es bastante baja.

Sin embargo, para los riesgos de datos RAW y para los riesgos de control, ambos tremadamente frecuentes, se siguen empleando en los procesadores multifuncionales las técnicas ya estudiadas, tanto hardware como software, para evitar estas paradas.

Por tanto, para evitar las paradas por riesgos de datos se siguen empleando el adelantamiento y la reordenación de código. El problema es que con instrucciones de latencias mayores que un ciclo, las paradas por riesgos RAW que no se pueden evitar pueden llegar a ser de muchos ciclos, lo que empeora significativamente el CPI del procesador.

En el caso de los riesgos de control, se siguen empleando la predicción de salto y el salto retardado. Sólo hay que ser cuidadoso cuando se aplica la técnica de salto retardado, ya que no se debe llenar la ranura con una instrucción de coma flotante. No tiene sentido intentar ocultar el ciclo de espera hasta que se resuelve el salto con una instrucción de latencia superior a un ciclo que acaba introduciendo un retardo mayor.

En un procesador segmentado multifuncional es necesario identificar más tipos de riesgos diferentes y con un repertorio y una ruta de datos más compleja, por lo que la etapa D se vuelve una de las más largas en la ejecución de las instrucciones debido a la responsabilidad de gestionar todos estos riesgos. Normalmente se añade a la unidad de control una tabla de planificación en la que se van almacenando

las instrucciones que se emiten, los operandos que necesitan y el avance que se planifica para ellas por la ruta de datos. De esta manera se identifican los riesgos, y se resuelven con la técnica más adecuada en cada caso, con facilidad y eficiencia.

En el caso del nanoMIPS esta tabla debe permitir detectar los siguientes riesgos y realizar las siguientes acciones:

- Riesgos estructurales. Si la instrucción necesita una unidad funcional que no está segmentada y que ya está ocupada, tiene que parar su avance hasta que la unidad esté libre (la parada puede llegar a ser de tantos ciclos como la latencia de la operación más larga que realiza la unidad por la que se espera). Si va a haber conflictos en el acceso a memoria o al banco de registros, hay que parar también el avance de la instrucción (en este caso la parada suele ser de un único ciclo).
- Riesgos de datos RAW. Se debe parar el avance de la instrucción hasta que sea posible la realización de un adelantamiento.
- Riesgos de datos WAW. Se debe parar el avance de la instrucción hasta que las instrucciones anteriores con el mismo registro destino (sin lecturas intermedias) vayan a pasar a la etapa M.
- Riesgos de control. Se realiza la predicción de salto no tomado buscando siempre la instrucción siguiente al salto.

En el caso de algunos procesadores multifuncionales se permite que las paradas que se realizan para resolver los riesgos se hagan en la primera etapa de ejecución en lugar de en la etapa D para que instrucciones que utilizan otras unidades funcionales puedan seguir avanzando en su ejecución.

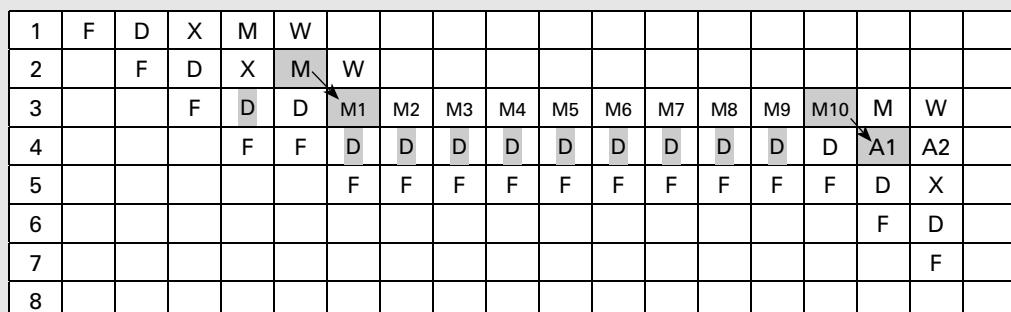
Ejemplo 1.11

Ejecución de un código en el nanoMIPS multifuncional con adelantamiento y predicción de salto no tomado (técnicas hardware).

Se ejecuta el siguiente código en el nanoMIPS que permite la ejecución de instrucciones en coma flotante:

1	bucle:	L.D	F1,0(R1)
2		L.D	F2,0(R2)
3		MUL.D	F10,F1,F2
4		ADD.D	F4,F6,F10
5		ADDI	R1,R1,#8
6		ADDI	R2,R2,#8
7		S.D	F4,0(R1)
8		BNE	R1,R10,bucle

Supongamos que inicialmente $R1=R2=0$ y $R10=800$. El diagrama de ejecución es:



1																			
2																			
3																			
4	A3	A4	M	W															
5	M	W																	
6	X	M	W																
7	D	D	X	M	W														
8	F	F	D	X	M	W													
Siguiente			-F-																

Observamos por primera vez un diagrama de ejecución en el que se desordena la terminación de instrucciones debido a las diferentes latencias de las instrucciones. Tenemos las siguientes paradas y adelantamientos:

- Adelantamiento entre el L.D de la instrucción 2 y la multiplicación de la instrucción 3. Como siempre, entre una lectura de memoria y una aritmético-lógica, no se puede evitar una parada. En este caso, el adelantamiento es entre la salida de memoria y la entrada de la primera etapa del multiplicador en coma flotante.
- Adelantamiento entre la multiplicación en coma flotante y la suma. A pesar de utilizar esta técnica hardware para resolver el riesgo RAW, la latencia de las operaciones en coma flotante hace que se introduzcan 9 ciclos de parada. El adelantamiento es entre la salida de la última etapa de la multiplicación en coma flotante y la entrada de la primera etapa del sumador en coma flotante.
- Parada de 1 ciclo de la instrucción 7 para evitar el riesgo estructural por el banco de registros (sin esa parada, coinciden las instrucciones 4 y 7 en las etapas M y W).
- Adelantamiento entre la suma en coma flotante y el S.D de la instrucción 7. Se pasa el operando a la etapa que lo necesita, por lo que se hace de la salida de memoria a la entrada de memoria.
- Fallo en la predicción de salto no tomado.

Con todo esto, el CPI del procesador en la ejecución de este bucle, que se repite 100 veces, es:

- Instrucciones = 8
- Ciclos = 8 (un ciclo por instrucción) + 1 parada por riesgos estructurales + 10 paradas por riesgo de datos + 1 ciclo perdido por fallo de predicción = 19

Las únicas excepciones son la primera iteración, con los 4 ciclos de llenado, y la última iteración, en la que la predicción del salto acierta (no se toma):

$$CPI = \frac{\text{ciclos}}{\text{instrucciones}} = \frac{4 + 99 \cdot 19 + 18}{100 \cdot 8} = 2.38$$

Se observa que a pesar de ser un código tan sencillo y de ejecutarse en una versión del procesador capaz de resolver los riesgos de datos y de control con técnicas que pueden evitar paradas, la introducción de las nuevas unidades funcionales con latencia mayor que 1 aleja mucho el CPI del valor ideal de 1.

1.5.4. Tratamiento de excepciones

Las excepciones son más difíciles de tratar en procesadores segmentados que en secuenciales, ya que cuando una instrucción produce una excepción siempre hay otras instrucciones que se están ejecutando al mismo tiempo en el procesador y que todavía no han terminado.

En general, existen dos maneras de tratar las excepciones en este caso. Cuando se atienden las excepciones respetando el orden de las instrucciones que las provocan (no el orden en el que se producen las propias excepciones), se habla de excepciones precisas. Si no, las excepciones son imprecisas y se tratan en el orden en que se producen, lo que no añade ninguna complejidad al hardware.

Las excepciones precisas garantizan que el proceso que se interrumpe cuando se salta a la RTE continuará correctamente después del tratamiento de la excepción. La única manera de garantizar esto, es conseguir que el estado del procesador en el momento en el que se pasa a tratar la excepción sea idéntico al que existiría en un procesador secuencial. Es decir, todas las instrucciones con un PC anterior al de la instrucción que ha provocado la excepción deben completarse y las instrucciones con un PC posterior, no deben ejecutarse ni modificar el estado del procesador. Una vez que se haya tratado la excepción, se retomará la ejecución en la instrucción que la ha provocado, si no llegó a completarse, o justo en la siguiente, depende del tipo de excepción.

¿Cuándo se utiliza el tratamiento preciso y cuándo el impreciso? Depende del tipo de excepción. Por ejemplo, si la excepción la provoca un código de operación inválido o un overflow de la ALU de enteros, el tratamiento suele ser impreciso, ya que el proceso que se interrumpe para tratar la excepción luego no va a poder continuar en el punto en el que se interrumpió. Sin embargo, en un punto de ruptura para depurar código, en una llamada al SO o en una interrupción de E/S, lo ideal sería realizar un tratamiento preciso porque el proceso sí debe continuar con su ejecución y producir resultados correctos.

El principal problema para implementar excepciones precisas en procesadores segmentados es que se pueden producir excepciones simultáneamente en varias instrucciones o incluso fuera de orden. Sin embargo, para que las excepciones se traten de manera precisa es imprescindible que las excepciones se traten en el orden secuencial de las instrucciones.

Por ejemplo, en la siguiente secuencia de instrucciones del nanoMIPS se pueden producir excepciones en prácticamente todas las etapas: en F y en M un fallo de página en la búsqueda de la instrucción o un acceso a memoria no alineado, en D un código de operación ilegal o en X las excepciones aritméticas.

LW	R1,0(R0)	F	D	X	M	W	
ADD	R4,R3,R5		F	D	X	M	W

Pero siempre tendrán que tratarse antes las excepciones provocadas en cualquier etapa de la instrucción de LW que en cualquier etapa de la instrucción ADD.

La manera de atender a las excepciones en el orden de las instrucciones en un procesador segmentado de única función es asociar un vector de estado a cada instrucción. Este vector de estado se va pasando de unas etapas a otras junto con la instrucción y si ocurre una excepción, se señala la etapa correspondiente en el vector y se deshabilitan en la ruta de datos las escrituras en memoria y en los registros para que ninguna instrucción pueda modificar el estado del procesador.

Cuando se va a pasar de la etapa M a la etapa W (y este paso se realiza en orden para todas las instrucciones en un procesador segmentado con una única unidad funcional), se comprueba para cada instrucción su vector de estado y si ha habido alguna excepción, se pasa a tratarla. La instrucción que origina la excepción y todas las instrucciones posteriores se empezarán desde cero cuando se vuelva de la rutina de tratamiento de la excepción.

El vector de estado asegura que las excepciones se tratan en orden ya que una instrucción no modifica el estado del procesador hasta que no se hayan comprobado y tratado las excepciones de todas las instrucciones anteriores.

La implementación de excepciones precisas en el nanoMIPS de única función es relativamente sencilla aplicando esta técnica del vector de estado. En otras arquitecturas pueden utilizarse mecanismos similares, la única diferencia importante aparece cuando se puede modificar el estado del procesador en otras etapas de las instrucciones, ya que esto debería tenerse en cuenta.

Si el procesador segmentado es multifuncional, la terminación de instrucciones fuera de orden hace que la utilización de vectores de estado ya no sea una solución: no garantiza que las excepciones se resuelvan en orden.

En el nanoMIPS se emiten sólo las instrucciones para las que se sabe seguro que todas las anteriores no van a generar una excepción. Esta técnica puede llegar a producir muchas paradas del procesador, y

aunque puede degradar el rendimiento en ciertas ocasiones, es la opción más barata y más segura para realizar el tratamiento preciso de excepciones.

En el caso de otros procesadores más complejos, existe una posibilidad que consiste en utilizar buffers de almacenamiento temporales para guardar información relativa al estado del procesador hasta que todas las instrucciones anteriores finalicen y se hayan tratado las excepciones que éstas hayan provocado.

Hay dos variantes de esta técnica:

- **Registro de historia.** Se almacenan los valores que van tomando todos los registros por si fuera necesario recuperar el estado del procesador en un momento anterior.
- **Registro de futuro.** Se almacenan los nuevos valores de los registros (los resultados de las operaciones) temporalmente, y sólo se actualizan en el banco de registros cuando han finalizado las instrucciones anteriores y se han tratado sus excepciones.

El problema es que es una solución cara cuando hay mucha diferencia de latencias entre instrucciones, ya que el buffer de almacenamiento temporal tiene que guardar información relativa a un gran número de instrucciones. Además hay que implementar lógica adicional para realizar los adelantamientos que resuelven los riesgos de datos RAW.

También existen soluciones mixtas que tratan las excepciones como imprecisas pero almacenando información para que las rutinas de tratamiento de excepción creen secuencias precisas.

Para ello es necesario almacenar qué instrucciones estaban en la ruta de datos en el momento de la excepción. Entonces, después de tratar la excepción unas instrucciones comienzan desde el principio y otras no, dependiendo de las instrucciones que ya hayan finalizado cuando se trate la excepción, se crearán unas secuencias precisas u otras. Es la rutina de tratamiento de excepción (RTE) la que se encarga de ejecutar algunas de las instrucciones, simulando su ejecución, y esto suele ser también bastante complejo de implementar.

Resumen de decisiones de diseño del repertorio de instrucciones

Decisión	Alternativas	Decisiones asociadas
Almacenamiento de operandos	Pila	
	Acumulador	
	Registros de propósito general (GPR)	Registro-registro, registro-memoria o memoria-memoria
Modos de direccionamiento soportados	Inmediato	Qué instrucciones lo soportan y rango de valores para el operando inmediato
	Registro	
	Absolute o directo	
	Indirecto o indirecto con desplazamiento	Qué instrucciones lo soportan y rango de valores para el desplazamiento
	Indexado	
Ordenación de la palabra de memoria	Little Endian	
	Big Endian	
Alineación de los accesos a memoria		
Alineación de los datos en los registros del procesador		
Tipo y tamaño de los operandos		
Conjunto de operaciones soportadas		
Especificación de la condición en los saltos condicionales	Evaluar los códigos de condición de la ALU	
	Evaluar uno o varios registros	
	Especificar la comparación en la propia instrucción	
Especificación de la dirección destino de salto en los saltos	Relativo al PC	Rango de valores para el desplazamiento
	Indirecto con registro	
Codificación del repertorio de instrucciones	Longitud fija	
	Longitud variable	
	Híbrida	Tipos de formato de instrucción

Resumen de decisiones de diseño del procesador

PROCESADORES SECUENCIALES

Decisión	Alternativas		Decisiones asociadas
Metodología de temporización	Monociclo		
	Multiciclo		
Conjunto de módulos que componen la ruta de datos	Contador de programa Registros de propósito general y bancos de registros Memorias de instrucciones/datos Unidades aritmético/lógicas Sumadores, restadores, comparadores y desplazadores Elementos de almacenamiento intermedio Decodificadores y multiplexores		
Valores de los puntos de control en la ruta de datos			
Niveles de decodificación/ control	Control global único		
	Control global + uno o varios controles locales		
Implementación de la unidad de control	Monociclo	Circuito combinacional	
	Multiciclo	Máquina de estados	
		Microprogramada	Codificación y secuenciamiento de las microinstrucciones
Técnicas de tratamiento de excepciones			
Técnicas de optimización			

PROCESADORES SEGMENTADOS

Decisión	Alternativas		Decisiones asociadas
Número de etapas de segmentación y tareas que realiza cada etapa			
Tipo y número de unidades funcionales – Segmentación de unidades funcionales			
Técnicas de resolución de riesgos estructurales	Paradas		
	Recursos replicados		
	Establecimiento de turnos de utilización		

Técnicas de resolución de riesgos de datos	Paradas	Etapa en la que se producen las paradas
	Adelantamiento (técnica HW)	Etapas origen y destino de los adelantamientos
	Reordenación de código (técnica SW)	
Técnicas de resolución de riesgos de control	Paradas	
	Predicción de salto (técnica HW)	Tomado o no tomado
	Salto retardado (técnica SW)	
Técnicas de tratamiento de excepciones	Imprecisas	
	Precisas	Vector de estados Registro de historia Registro de futuro Paradas
	Técnicas mixtas	

BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS

- FARGUHAR, E. & BUNCE, P. J. (1994): *The MIPS Programmer's Handbook* (1.^a ed.), Morgan Kaufmann.
- FLYNN, M. J.; JOHNSON, J. D. & WAKEFIELD, S. P. (1985): «On Instruction Sets and Their Formats», *IEEE Transactions on Computers*, 34 (3), 242-254.
- HARTSTEIN, H. & PUZAK, T. (2002): «The optimum pipeline depth for a microprocessor», *Proceedings of the 29th International Symposium on Computer Architecture*, 7-13.
- KANE, G. & HEINRINCH, J. (1991): *MIPS RISC Architecture* (2.^a ed.), Prentice Hall.
- OMONDI, A. R. (1999): *The Microarchitecture of Pipelined and Superscalar Computers* (1.^a ed.), Springer.
- ORTEGA, J.; ANGUITA, M. & PRIETO, A. (2005): *Arquitectura de Computadores*, Thomson.
- PARHAMI, B. (2007): *Arquitectura de Computadoras. De los microprocesadores a las supercomputadoras* (1.^a ed.), McGraw Hill.
- PATTERSON, D. A. & HENNESSY, J. L. (2008): *Computer Organization and Design: The Hardware/Software Interface* (4.^a ed.), Morgan Kaufmann.
- SHEN, J. P. & LIPASTI, M. (2006): *Arquitectura de Computadores. Fundamentos de los procesadores superescalares* (1.^a ed.), McGraw Hill.
- SILC, J.; ROBIC, B. & UNGERER, T. (1999): *Processor Architecture. From Dataflow to Superscalar and Beyond*, Springer.
- STALLINGS, W. (2007): *Organización y arquitectura de computadores* (7.^a ed.), Pearson - Prentice Hall.
- The Morgan Kaufmann Series in Computer Architecture and Design* (1999). See *MIPS run* (1.^a ed.), Morgan Kaufmann.
- WALL, D. (1991). *Limits of instruction-level parallelism*. *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 176-188.

PROBLEMAS

- 1.1.** Se diseñan tres repertorios de instrucciones GPR diferentes, todos ellos con tres operandos y registro-memoria. En los tres casos se tarda 1 ciclo de reloj en acceder a un operando almacenado en un registro y 10 ciclos en acceder a un operando almacenado en memoria. Utilizar los datos de la siguiente tabla para calcular el CPI medio de cada uno de los procesadores:

Repertorio	N.º de registros del procesador	Ciclos por instrucción	Probabilidad de encontrar un operando en un registro
A	0	3	0
B	16	5	0.4
C	32	8	0.8

- 1.2.** Se diseña un repertorio de instrucciones GPR registro-registro para un procesador con un banco de 32 registros. En media, si se deja que las instrucciones utilicen el formato que les es más «natural», el 20% de las instrucciones tienen un operando fuente y otro destino, el 30% tienen dos operandos fuente y uno destino, el 25% tiene un operando fuente, otro destino y un inmediato y el 25% restante tiene un operando destino y uno inmediato. En el repertorio se utilizan operandos inmediatos de 16 bits y con el conjunto de operaciones soportadas, son necesarias 98 instrucciones diferentes.
- a) ¿Cuántos bits se necesitan para codificar cada uno de los cuatro tipos de instrucción?
 b) ¿Cómo se diseñaría este repertorio con codificación de longitud fija? ¿Y variable? ¿E híbrida? Comparar la eficiencia de las tres codificaciones.

- 1.3.** Se diseña un procesador nanoMIPS monociclo con las siguientes latencias para los módulos de la ruta de datos:

- Lectura de la memoria de instrucciones: 0.3 ns.
- Lectura de la memoria de datos: 0.4 ns.
- Escritura en la memoria de datos: 0.5 ns.
- Lectura y escritura en el banco de registros: 0.15 ns.
- Operación aritmético-lógica con la ALU: 0.25 ns.
- Suma para preparar el PC siguiente: 0.1 ns.
- Suma del PC y el desplazamiento del salto: 0.1 ns.

Con estos datos ¿cuál es la frecuencia máxima de funcionamiento del procesador suponiendo que todos los módulos se utilizan secuencialmente?

- 1.4.** Se realiza un diseño multiciclo del procesador del ejercicio anterior según las etapas estudiadas para el nanoMIPS. Si las frecuencias de aparición de las instrucciones del repertorio son las que se detallan a continuación, ¿cuál de las dos versiones, la monociclo o la multiciclo, obtiene un mejor rendimiento? ¿Cuánto mejor?

- Lectura de memoria: 35%.
- Escritura en memoria: 10%.
- Aritmético-lógicas: 35%.
- Saltos condicionales: 20%.

- 1.5. Modificar la implementación de la ruta de datos y la unidad de control del nanoMIPS monociclo para que pueda ejecutar las instrucciones J (salto incondicional con destino de salto relativo al PC) y JR (salto incondicional con destino de salto indirecto con registro).
- 1.6. Modificar la implementación de la ruta de datos y la unidad de control del nanoMIPS multiciclo para abaratar sus costes utilizando un banco de registros con un único puerto de lectura.
- 1.7. En el nanoMIPS multiciclo con unidad de control cableada (basada en máquina de estados) se desea tener en cuenta un nuevo tipo de excepción, que se produce cuando se detecta un error en la codificación de una palabra en la única memoria de la ruta de datos. Discutir las modificaciones que deberían hacerse en la ruta de datos y en la unidad de control para poder realizar el tratamiento de esta excepción cuando se produce en una palabra que almacena una instrucción o en una palabra que almacena un dato. Utilizar una señal ErrorCaché que activa la propia memoria cuando detecta un error de este tipo.
- 1.8. Un procesador secuencial monociclo tiene un tiempo de ciclo de 2.4 ns y su ruta de datos está constituida por módulos de latencias 0.1, 0.4, 0.3, 0.5, 0.8, 0.2 y 0.1 ns que se utilizan en ese orden durante la ejecución de instrucciones. Al segmentar el procesador no es posible modificar el orden de esos módulos ni dividir la utilización de un módulo en etapas diferentes. Suponiendo que los registros de segmentación introducen 0.05 ns de retardo, discutir cuál es el tiempo de ciclo mínimo que puede alcanzarse segmentando el procesador. Si el procesador se divide en el número menor de etapas que permita lograr este periodo mínimo ¿cuál es la latencia de una instrucción en este procesador?
- 1.9. Discutir qué información es necesario almacenar en cada uno de los registros de segmentación del nanoMIPS segmentado.
- 1.10. El siguiente fragmento de código se ejecuta en el nanoMIPS segmentado.

1	ADD	R5,R4,R2
2	LW	R1,0(R5)
3	AND	R6,R1,R3
4	ADD	R8,R6,R1
5	OR	R5,R4,R9
6	SW	R6,0(R0)
7	SW	R8,100(R0)

Calcular el número de ciclos necesarios para ejecutar el código y justificarlo con un diagrama de ejecución:

- Si no existe posibilidad de adelantar operandos ni de reordenar el código.
- Si se puede utilizar adelantamiento de operandos.
- Si además de permitir el adelantamiento, se permite reordenar el código para que el número de ciclos sea mínimo, ¿cuál sería la mejor reordenación? ¿cuántos ciclos serían necesarios en este caso para ejecutar este código?

- 1.11. Calcular el número de ciclos necesarios para ejecutar el mismo código que en el ejercicio anterior en un nanoMIPS que permite adelantamiento y en el que la memoria de instrucciones y la de datos están unificadas. Mantener la reordenación de código realizada en al apartado c de ese ejercicio.

- 1.12.** Calcular el CPI que se obtiene en el nanoMIPS al ejecutar el siguiente código si el valor inicial de R10 es 1000 y el de R1 es 0. El procesador utiliza el adelantamiento para resolver los riesgos de datos y la predicción de salto no tomado para resolver los riesgos de control (los saltos se resuelven en la etapa D):

1	loop:	LW	R2,A(R1)
2		LW	R3,B(R1)
3		ADD	R2,R2,R3
4		SUB	R3,R3,R4
5		SW	R2,C(R1)
6		SW	R3,D(R1)
7		ADDI	R1,R1,#4
8		BNE	R1,R10,loop

- 1.13.** Mostrar el diagrama de ejecución y calcular el CPI medio que se obtiene al ejecutar el siguiente código en el nanoMIPS segmentado con adelantamiento y resolución de saltos en la etapa D. Suponer que el 50% de las veces A(i)=B(i):

1	loop:	ADDI	R10,R0,#4000
		ADDI	R1,R0,#0
2		LW	R2,A(R1)
3		LW	R3,B(R1)
4		SUB	R4,R2,R3
5		BEQ	R4,R0,else
6		SW	R3,A(R1)
7	else:	J	final
8		LW	R5,C(R1)
9		ADD	R2,R2,R5
10		SW	R2,A(R1)
11	final:	SW	R2,B(R1)
12		ADDI	R1,R1,#4
		BNE	R1,R10,loop

¿Qué mejora podría obtenerse en la ejecución de este código si se utiliza la técnica de salto retardado? Discutir las diferentes posibilidades que ofrece este código para llenar la ranura.

- 1.14.** Se ejecuta el siguiente código en un nanoMIPS segmentado multifuncional en el que se incluye una ALU de enteros con latencia 1 ciclo, un sumador/restador en coma flotante con latencia 2 ciclos, un multiplicador en coma flotante con latencia 5 ciclos y un divisor en coma flotante con latencia 10 ciclos. Ninguna de las unidades funcionales en coma flotante está segmentada y se utiliza el adelantamiento para resolver los riesgos de datos. Los saltos se resuelven en la etapa D de las instrucciones. Calcular el CPI que se obtiene al ejecutar el código si el registro R10 contiene un 0 inicialmente y el registro R1 un 800:

1	loop:	L.D	F2,A(R10)
2		L.D	F3,B(R10)
3		ADD.D	F4,F2,F3
4		SUB.D	F5,F2,F3
5		MUL.D	F6,F4,F5
6		S.D	F6,C(R10)
7		ADDI	R10,R10,#8
8		BNE	R1,R10,loop

- 1.15.** Calcular la ganancia que se obtendría al ejecutar el código anterior si se segmentaran las unidades funcionales en coma flotante y se utilizara la técnica de salto retardado. Discutir las diferentes posibilidades que existen para llenar la ranura de salto.
- 1.16.** Mostrar el diagrama de ejecución del siguiente código si se ejecuta en el nanoMIPS segmentado multifuncional con una ALU de enteros con latencia 1 ciclo, un sumador/restador en coma flotante con latencia 2 ciclos, un multiplicador en coma flotante con latencia 6 ciclos y un divisor en coma flotante con latencia 8 ciclos. Todas las unidades en coma flotante están segmentadas, se utiliza el adelantamiento para resolver los riesgos de datos y los saltos se resuelven en la etapa D de las instrucciones.

1	loop:	L.D	F2,0(R8)
2		L.D	F4,8(R8)
3		DIV.D	F4,F4,F2
4		DIV.D	F8,F4,F6
5		MUL.D	F1,F8,F2
6		ADD.D	F6,F1,F4
7		S.D	F6,32(R8)
8		ADDI	R8,R8,#8
9		BNE	R8,R10,loop

- 1.17.** Se diseña una versión del nanoMIPS segmentado en la que se utiliza el modo de direccionamiento indirecto en lugar del indirecto con desplazamiento. Discutir las diferencias entre el diseño de este nuevo procesador (ruta de datos, unidad de control, gestión de riesgos, etc) y el nanoMIPS clásico.
- 1.18.** Se diseña un procesador segmentado registro-memoria con dos tipos diferentes de formato de instrucción, uno registro-memoria y otro registro-registro. El único modo de direccionamiento que se soporta en este procesador es el indirecto con desplazamiento. Además, los saltos evalúan su condición comparando dos registros y especifican la dirección destino de salto relativa al PC. El repertorio de instrucciones inicial para este procesador incluye las mismas operaciones que el del nanoMIPS (lectura y escritura en memoria, instrucciones aritmético-lógicas básicas con números enteros y un salto condicional). Se decide que la segmentación utilice seis etapas diferentes:
- BI: Búsqueda de instrucción en la memoria de instrucciones.
 - DLO: Decodificación y lectura de operandos en registros.
 - ALU1: Cálculo de la dirección de acceso a memoria y de la dirección destino de salto.
 - MEM: Acceso a memoria de datos.
 - ALU2: Operación aritmético-lógica y evaluación de la condición en las instrucciones de salto.
 - ER: Escritura de resultados en registros.
- a) Discutir los módulos hardware que deben incluirse en la ruta de datos para evitar riesgos estructurales.
- b) Determinar los adelantamientos que se necesitan para eliminar o reducir las paradas del procesador por riesgos de datos. Justificar la necesidad de cada adelantamiento con una secuencia de instrucciones.
- c) Mostrar los riesgos de control que se pueden producir y calcular los ciclos de penalización que suponen. Proponer una predicción, salto tomado o no tomado, para intentar eliminar o reducir las paradas por riesgos de control. Justificar cuál de las dos predicciones sería la más adecuada para este procesador.

AUTOEVALUACIÓN

1. ¿Qué diferencias fundamentales hay entre un repertorio de instrucciones RISC y uno CISC?
2. ¿Cuáles son las principales ventajas de una arquitectura GPR?
3. ¿De qué tres factores depende el tiempo que tarda en ejecutarse un código en un procesador según la ecuación de prestaciones?
4. ¿Qué ventajas e inconvenientes tienen las implementaciones monociclo y multiciclo de un procesador secuencial?
5. ¿Qué dos alternativas existen para la implementación de la Unidad de Control de un procesador secuencial multiciclo?
6. ¿Cuál es la ganancia ideal que se obtiene segmentando un procesador secuencial multiciclo? ¿Qué factores impiden que se consiga esta ganancia en la realidad?
7. ¿Qué técnicas pueden utilizarse para resolver riesgos de datos RAW en un procesador segmentado?
8. ¿En qué consiste la técnica de salto retardado?
9. ¿Qué riesgos aparecen en el nanoMIPS segmentado multifuncional que no aparecen en el de única función? ¿Cómo se resuelven?
10. ¿Cómo se hace un tratamiento preciso de excepciones en el nanoMIPS segmentado de única función? ¿Se puede utilizar la misma técnica en el caso del nanoMIPS multifuncional? ¿Por qué?

2

Conceptos básicos de Memoria y E/S

Contenidos

- 2.1. Diseño de una jerarquía de memoria básica
- 2.2. Mecanismo completo de acceso a memoria
- 2.3. Evaluación de prestaciones de la jerarquía de memoria
- 2.4. Niveles de la jerarquía de memoria
- 2.5. Diseño de un sistema de E/S básico
- 2.6. Mecanismo completo de una operación de E/S
- 2.7. Evaluación de prestaciones del sistema de E/S
- 2.8. Buses de E/S
- 2.9. Gestión del sistema de E/S

En el capítulo anterior se han incluido en la ruta de datos de los diferentes procesadores estudiados una o varias memorias capaces de funcionar a la misma velocidad que el resto de componentes hardware que se incluyen dentro del procesador. Esto es así realmente, pero estas memorias tienen un tamaño muy limitado que no permite diseñar sistemas con las prestaciones requeridas hoy en día.

En este capítulo se justifica la necesidad de una jerarquía de memoria dentro de las arquitecturas de computadores actuales. A continuación, se explica el mecanismo completo de acceso de memoria en este tipo de jerarquía y se proporcionan herramientas cuantitativas para la evaluación de su rendimiento.

Una vez adquiridos estos conceptos básicos acerca del diseño y evaluación de la jerarquía de memoria, se analizan los aspectos tecnológicos y de diseño de cada uno de los niveles que la componen: memoria caché, memoria principal y memoria virtual.

El resto del capítulo se dedica al estudio del sistema de E/S. De poco sirve un diseño sofisticado de procesador y de jerarquía de memoria si los mecanismos de E/S que permiten al usuario introducir datos en el computador y obtener de él los resultados esperados se convierten en el principal cuello de botella.

Por eso se estudia el diseño del sistema de E/S analizando los tres grandes aspectos de diseño que influyen en su rendimiento: los buses de interconexión, los dispositivos de E/S y la gestión de la E/S, es decir, la manera en la que el dispositivo se relaciona con el procesador y la jerarquía de memoria del sistema para realizar las operaciones de E/S.

Como se hizo en el capítulo 1, se comienza por estudiar los conceptos básicos de diseño del sistema, para pasar después a comprender el mecanismo completo necesario para llevar a cabo una operación de E/S y a proponer herramientas de evaluación de rendimiento que permitan decidir qué opciones de diseño obtienen un mejor rendimiento en cada caso.

A continuación se estudian las técnicas de diseño de buses de interconexión y los mecanismos de gestión de E/S tradicionales, ya que el estudio del diseño de los propios dispositivos de E/S queda fuera del alcance de este libro.

2.1 Diseño de una jerarquía de memoria básica

El sistema de memoria formado por una memoria principal única que se propuso en los primeros modelos de computador sencillo quedó descartado rápidamente por sus bajas prestaciones.

La alternativa a esta memoria única es una jerarquía de memoria organizada en niveles, cuanto más cercanos al procesador, más pequeños, rápidos y caros. Hoy en día esta jerarquía incluye, en casi todos los casos, tres tipos de memorias: memoria caché, memoria principal y memoria virtual (figura 2.1).

Cada uno de estos niveles está ubicado físicamente en un lugar distinto, se fabrica con una tecnología diferente y se gestiona de manera independiente:

- **Memoria caché (MC).** Ubicada en el mismo chip que el procesador, fabricada con memoria RAM estática (SRAM) y controlada por el controlador de caché incluido en el mismo chip. Hoy en día lo habitual es que haya varios niveles de memoria caché. En el capítulo 1 se ha estado utilizando una memoria caché para datos y otra para instrucciones, ese es el nivel de memoria que aparecía en la ruta de datos del procesador, una memoria caché de nivel 1.
- **Memoria principal (MP).** Ubicada en un chip diferente al procesador, fabricada con memoria RAM dinámica (DRAM) y controlada por el controlador de memoria principal. Este controlador es muy importante para el rendimiento de la jerarquía de memoria ya que se encarga de la planificación de los accesos a memoria principal, como se estudiará más adelante. Hoy en día puede ubicarse en el mismo chip que el procesador y la memoria caché, o en otro chip como el chipset norte o el MCH.
- **Memoria virtual (MV).** Ubicada en la actualidad en el disco duro, se fabrica por lo tanto con tecnología magnética y se controla desde el sistema operativo a través del controlador del disco duro.

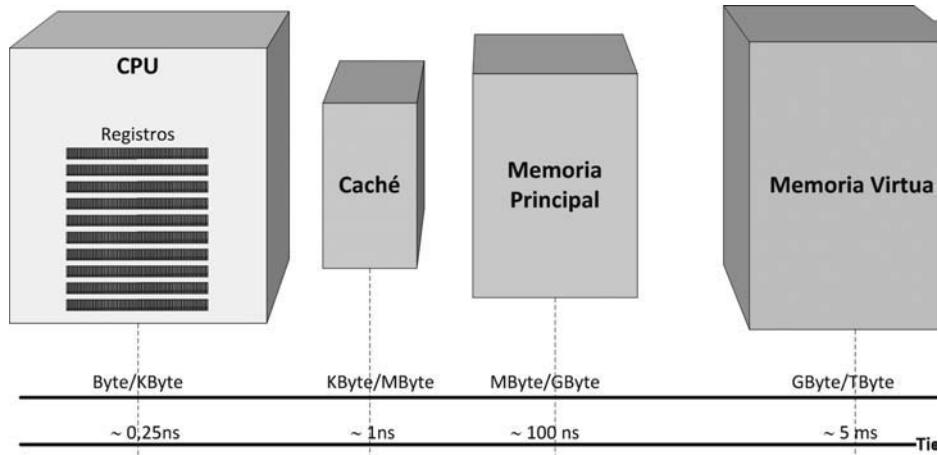


FIGURA 2.1

Órdenes de magnitud de capacidades y tiempos de acceso para los distintos niveles de la jerarquía de memoria.

El objetivo es conseguir una estructura de memoria de gran capacidad, con un coste casi tan bajo como el del nivel más barato de la jerarquía, pero con latencia comparable a la del nivel más rápido (figura 2.2).

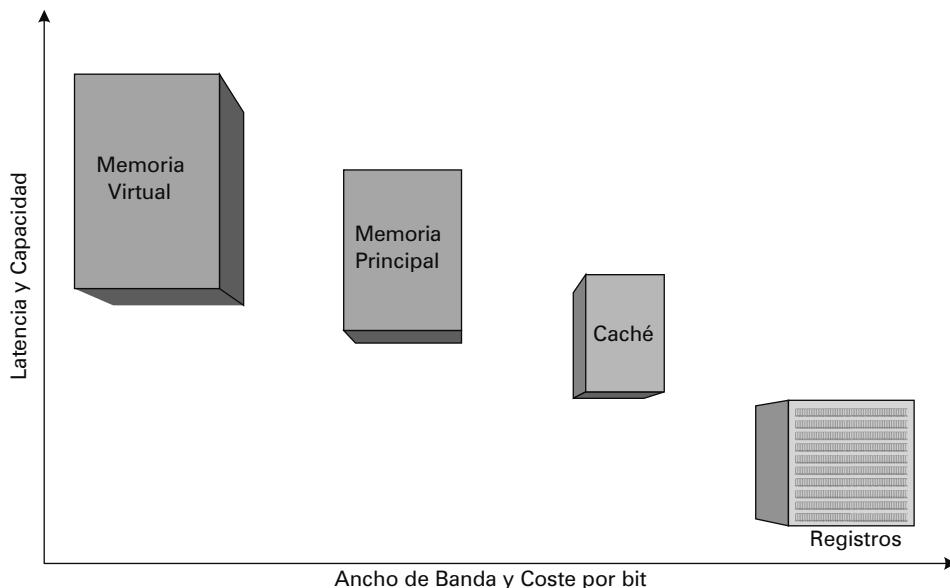


FIGURA 2.2

Tecnologías y características de los diferentes niveles de la jerarquía de memoria.

Hay dos propiedades que la jerarquía debe cumplir para que su funcionamiento sea el adecuado. La primera, denominada inclusión, implica que cualquier información contenida en un nivel de la jerarquía debe estar también en los niveles superiores (en los que están más lejos del procesador a

partir de él). La segunda, denominada coherencia, garantiza que las copias de la misma información en los diferentes niveles de la jerarquía son coherentes entre sí, es decir, que almacenan los mismos valores.

Además, en cualquier jerarquía de memoria debe haber una correspondencia de direcciones entre los distintos niveles de la jerarquía.

Cuando el procesador realiza un acceso a memoria, primero se busca la palabra que hay que leer o escribir en la memoria caché (MC). Si esta palabra se encuentra en la caché, ha ocurrido un acierto. Si por el contrario, no se encuentra, ha ocurrido un fallo.

En este último caso, se traerá de memoria principal (MP) un bloque que contendrá varias palabras, entre ellas, la que ha producido el fallo.

Llegados a este punto, se puede plantear la siguiente pregunta: ¿por qué la jerarquía de memoria supone un beneficio si es posible que en todos los accesos a memoria la memoria caché falle y al final haya que acceder siempre a la memoria principal? Es más, ¿por qué cuando se produce un fallo en la memoria caché se trae un bloque con varias palabras desde la memoria principal en lugar de traer sólo la palabra que el procesador ha solicitado? ¿no es esto una pérdida de tiempo y de recursos?

Las respuestas a estas preguntas se pueden encontrar en lo que se denomina el principio de localidad. Este principio tiene dos aspectos diferentes:

- **Localidad Espacial.** Tendencia del procesador a referenciar elementos de memoria cercanos a los últimos que han sido referenciados.
- **Localidad Temporal.** Tendencia del procesador a referenciar elementos de memoria que han sido referenciados recientemente.

El principio de localidad temporal asegura que si el procesador ha solicitado una palabra para leer o escribir en ella y se trae esta palabra a memoria caché, es muy probable que esta palabra vuelva a utilizarse recientemente. Por eso la tasa de aciertos de la memoria caché es lo suficientemente importante como para mejorar el rendimiento de la jerarquía de memoria.

Y por el principio de localidad espacial, si en lugar de traer sólo la palabra solicitada por el procesador a la memoria caché, se trae un bloque de palabras contiguas en la memoria, con toda probabilidad alguna otra palabra de este bloque se utilizará poco tiempo después, lo que también aumenta la probabilidad de que se produzcan aciertos de la memoria caché.

Por lo tanto, cuando hay un acierto (y esto debería ser bastante frecuente gracias a la localidad), el tiempo que supone el acceso a memoria es muy pequeño ya que los accesos terminan en la memoria caché, que es la más rápida de la jerarquía.

Sin embargo, cuando hay un fallo, este tiempo es bastante mayor, ya que hay que traer un bloque de memoria principal. La penalización por fallo dependerá de la latencia de memoria principal y de su ancho de banda.

Al acceder a memoria principal también pueden producirse aciertos y fallos, ya que no toda la información cabe al mismo tiempo en este nivel. La relación entre la memoria principal y la memoria virtual es similar a la que existe entre la memoria caché y la principal.

La memoria principal se divide en páginas o segmentos, de mayor tamaño que los bloques de caché. Cuando se produce un fallo con una página o segmento que no se encuentra en la memoria principal, deberá traerse de la memoria virtual.

La principal diferencia con los fallos de la memoria caché está en que la penalización por este tipo de fallos es bastante grande, ya que la memoria virtual es la más lenta de toda la jerarquía. Además en la gestión de este último nivel interviene en gran medida el sistema operativo, ya no se trata de una gestión llevada a cabo exclusivamente desde la propia arquitectura. Por ello el procesador suele pasar a realizar otro tipo de tareas, realizando un cambio de contexto, hasta que la página o segmento esté disponible en la memoria principal.

2.2 Mecanismo completo de acceso a memoria

Antes de comenzar a estudiar en profundidad cada uno de los niveles que componen una jerarquía de memoria habitual, es necesario comprender todas las etapas en las que se descompone un acceso a memoria completo.

En primer lugar, el procesador genera la dirección virtual de la palabra que se debe leer o escribir. Normalmente, lo primero que se hace es traducir esta dirección virtual a una dirección física comprensible para la jerarquía de memoria.

Si se consigue realizar esta traducción de dirección, es porque la palabra que se está buscando se encuentra actualmente en la memoria principal. Con esta dirección se accede a la memoria caché, y si la palabra buscada se encuentra en este nivel, el acceso a memoria ya ha finalizado.

Si por el contrario, se produce un fallo, se busca la palabra en el siguiente nivel, en este caso en la memoria principal porque de momento se supone que sólo existe un nivel de memoria caché. Existen tres tipos de fallos en la memoria caché:

- **Iniciales.** La primera vez que se referencia una palabra, ésta no puede encontrarse en la caché y se produce un fallo en todos los casos.
- **De capacidad.** Cuando en la caché no caben todos los bloques necesarios para la ejecución de un programa y deben producirse reemplazamientos.
- **De conflicto.** Cuando varios bloques tienen la misma localización en la memoria caché y se van reemplazando unos a otros debido a esta colisión.

Sea cual sea el tipo de fallo, su resolución siempre implica buscar el bloque necesario en la memoria principal. Para ello hay que pasar por el controlador de memoria principal (que puede estar o no en el mismo chip que el procesador), que mapeará la dirección física buscada a la ubicación física de la palabra dentro de los chips DRAM. Además, el controlador planificará el acceso a la memoria principal, ya que otros dispositivos, como la tarjeta gráfica o la de red, también acceden a la memoria principal.

Cuando se realice este acceso, todo el bloque en el que se incluye la palabra solicitada por el procesador se envía a la memoria caché para resolver su fallo, y el acceso puede completarse con éxito.

Si por el contrario, la palabra no se encuentra en memoria principal y desde un principio no se pudo traducir su dirección virtual a dirección física por este motivo, se debe resolver el fallo de página o segmento desde la memoria virtual. El sistema operativo realiza un cambio de contexto para que otro proceso pase a ejecutarse en el procesador mientras se trae desde memoria virtual la página o segmento que hace falta para resolver el fallo. Una vez que esté en memoria principal esta página o segmento, ya puede llevarse el bloque correspondiente a memoria caché y cuando el proceso que había provocado el fallo vuelva a pasar a ejecución, el acceso puede completarse con éxito.

Si en lugar de un único nivel de caché hubiera dos, el mecanismo de acceso sería el mismo excepto porque cuando ocurre un fallo en el primer nivel de caché, primero se intenta resolver desde el segundo nivel de caché. Si la palabra que ha provocado el fallo se encuentra en este segundo nivel de caché, se envía su bloque completo al primer nivel y el acceso puede completarse con éxito sin necesidad de salir fuera del chip del procesador. En caso contrario, se debe intentar resolver el fallo desde la memoria principal igual que en el caso de un único nivel de caché.

Por el principio de inclusión, el bloque que contiene la palabra que ha provocado el fallo se lleva desde la memoria principal hasta la caché de segundo nivel, y desde ésta, a la caché de primer nivel para que el procesador pueda finalizar el acceso con éxito (figura 2.3). Como se verá más adelante, en este proceso suelen manejarse bloques de tamaños diferentes.

Si el procesador está segmentado, todo el mecanismo de acceso a memoria hasta llegar a la memoria principal (porque si la memoria principal falla, hay que ir a memoria virtual y el sistema operativo provo-

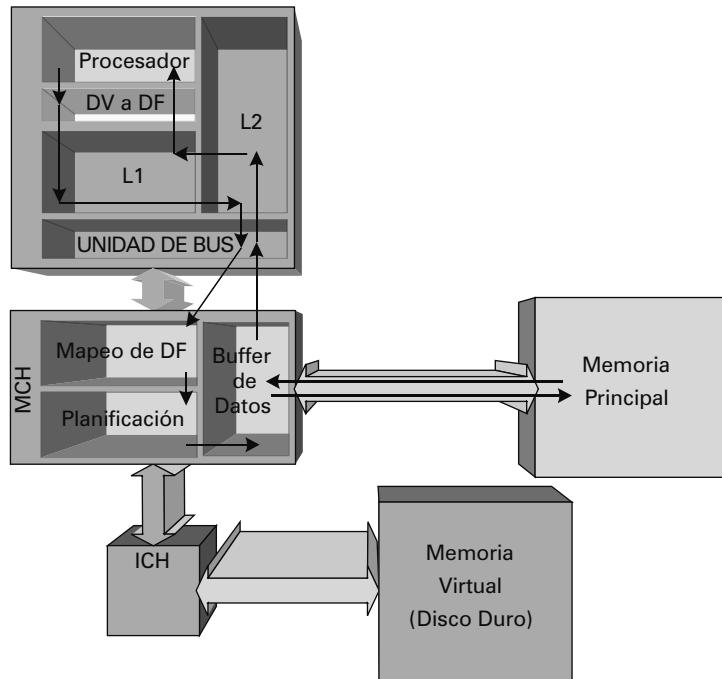


FIGURA 2.3

Mecanismo completo de acceso a memoria con el esquema de jerarquía de memoria tradicional en un PC y dos niveles de caché.

ca un cambio de contexto) debería completarse en la etapa de acceso a memoria, es decir, en un ciclo de reloj. En el caso del nanoMIPS, en la etapa M, como puede verse en la figura 2.4.

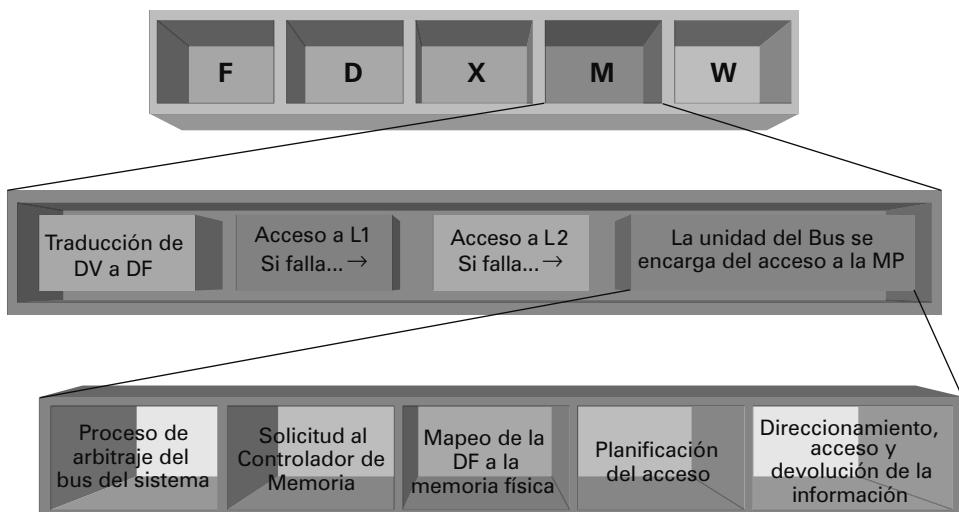


FIGURA 2.4

Trabajo realizado en la etapa M del nanoMIPS segmentado.

Esto haría que la etapa M limite la duración del ciclo de reloj del procesador segmentado, por lo que habría que optimizar al máximo la jerarquía de memoria de manera que no afectara negativamente al rendimiento del procesador. Para ello se estudiarán técnicas de optimización de la jerarquía de memoria en el capítulo 4 de este libro.

Aún utilizando estas técnicas, incluir los accesos a memoria completos en la etapa M aumenta mucho el valor del CPI, por lo que en la mayor parte de los diseños en la etapa M sólo se tienen en cuenta los accesos a caché, y si hay que resolver un fallo, el tiempo que se tarda en resolver este fallo se considera un tiempo extra que hay que sumar al tiempo que tarda en ejecutarse una instrucción. Además, como se estudiará en el capítulo 4, muy a menudo se utiliza una técnica, denominada de caché no bloqueante, que permite que las instrucciones avancen por la ruta de datos del procesador mientras una de ellas está resolviendo un fallo. Esto implica que el tiempo extra que se invierte en resolver el fallo se solapa con la ejecución de otras instrucciones.

2.3 Evaluación de prestaciones de la jerarquía de memoria

En el caso de la jerarquía de memoria también se puede definir una ecuación de prestaciones que proporcione una herramienta cuantitativa de evaluación de rendimiento.

En este caso lo que interesa es saber cuánto tiempo le cuesta en media al procesador realizar un acceso a memoria. Si la memoria caché fuera perfecta y no fallara nunca, el tiempo medio de acceso a memoria sería justo el tiempo de acierto a la memoria caché. Pero como se producen fallos, el tiempo medio de acceso a memoria se calcula teniendo en cuenta estos fallos y el tiempo que se invierte en resolverlos, lo que se ha denominado penalización por fallo con anterioridad:

$$t_{MEM} = t_{aciertoMC} + TF \cdot pF$$

$t_{aciertoMC}$: Tiempo de acierto de MC

$$TF : Tasa de fallos de MC \Rightarrow TF = \frac{\text{número de fallos}}{\text{número total de accesos a memoria}}$$

pF : Penalización por fallo en MC

En el capítulo 1, para calcular el tiempo que un determinado fragmento de código tardaba en ejecutarse en el procesador, se suponía que este código sólo invertía tiempo en la propia ejecución. Pero tras estudiar las dos primeras secciones de este capítulo queda claro que los accesos a memoria llevan un tiempo asociado y que no se puede suponer que la jerarquía de memoria es perfecta.

Por lo tanto, para ser realistas, hay que tener en cuenta el tiempo invertido en acceder a la memoria con los posibles fallos que se produzcan. O bien porque están incluidos en el CPI cuando se aplica la ecuación de prestaciones del procesador o bien porque se suma al tiempo de CPU (calculado con el CPI que tiene en cuenta que todos los accesos al primer nivel de la memoria caché acierrtan), el tiempo que se tarda en resolver los fallos de memoria que se producen. Este segundo caso, $t=t_{CPU}+t_{MEM}$, es el más habitual por lo discutido en la sección anterior.

Si lo que interesa es evaluar las prestaciones de la memoria principal o de la memoria virtual como niveles aislados de la jerarquía, las métricas que suelen utilizarse son:

- **Latencia.** Tiempo que transcurre desde que un acceso a memoria comienza hasta que finaliza. Está muy relacionada con la tecnología con la que está fabricada la memoria.
- **Ancho de banda.** Cantidad de información por unidad de tiempo que puede transferirse desde/hacia la memoria. En este caso está muy relacionado con la organización de la memoria más que con la tecnología.

Ejemplo 2.1**Comparación entre una jerarquía de memoria perfecta y una jerarquía de memoria real.**

Un procesador que funciona con una frecuencia de reloj de 2 GHz y con un CPI ideal de 1, ejecuta un código de 100 instrucciones. Este procesador incorpora dos memorias caché, una para instrucciones (MI) y otra para datos (MD), que inicialmente se suponen ideales. Es decir, se supone que su tiempo de acceso es despreciable y que nunca fallan. Por lo tanto, el tiempo de ejecución de este código es el tiempo de CPU:

$$t = t_{CPU} = I \cdot CPI \cdot T = 100 \cdot 1 \cdot \frac{1}{2 \cdot 10^9} = 50 \cdot 10^{-9} s = 50 \text{ ns}$$

Pero en realidad estas dos memorias caché no son perfectas, y aunque supongamos que sus tiempos de acceso son despreciables, en ocasiones fallan obligando a realizar accesos a memoria principal.

Supongamos que la memoria caché de instrucciones tiene una tasa de fallos del 4% y una penalización por fallo de 100 ns; y que la memoria caché de datos tiene una tasa de fallos del 6% y una penalización por fallo de 115 ns.

Teniendo en cuenta que para ejecutar este código, hacen falta 100 accesos a la memoria de instrucciones (la búsqueda de cada instrucción) y 25 accesos a la memoria de datos (25 instrucciones del código son load o store), el tiempo que se invierte en acceder a memoria es:

$$t_{MEM} = \text{n.º de accesos a MI } (t_{acuerdo\ MI} + TF_{MI} \cdot pF_{MI}) + \text{n.º de accesos a MD } (t_{acuerdo\ MD} + TF_{MD} \cdot pF_{MD}) = \\ 100 (0 + 0.04 \cdot 100) + 25 (0 + 0.06 \cdot 115) = 572.5 \text{ ns}$$

Por lo que el tiempo de CPU de este código es realmente:

$$t = t_{CPU} + t_{MEM} = 50 \text{ ns} + 572.5 \text{ ns} = 622.5 \text{ ns}$$

2.4 Niveles de la jerarquía de memoria

2.4.1. Diseño de la memoria caché

La memoria caché almacena en cada momento unos determinados bloques de información, por lo tanto se divide en marcos capaces de albergar estos bloques en su interior.

La caché no sólo se compone de marcos, ya que para determinar qué bloque está ocupando un determinado marco en un instante concreto se utilizan etiquetas o tags que también deben almacenarse en la memoria. Estas etiquetas se comparan con la del bloque que se está buscando para determinar si éste se encuentra o no en la memoria caché y si por lo tanto se ha producido un acierto o un fallo (figura 2.5).

Para comprender el funcionamiento de la memoria caché y para diseñar este nivel de la jerarquía de manera adecuada hay que entender cuatro aspectos básicos de su diseño.

2.4.1.1. ORGANIZACIÓN DE LA MEMORIA CACHÉ

El primer aspecto importante que hay que decidir es el tamaño de la memoria caché. Si la caché es demasiado pequeña, esto afectará negativamente a la tasa de fallos, ya que no se capturará bien la localidad, y muchos de los accesos a memoria realizados por el procesador terminarán produciendo fallos de capacidad.

Sin embargo, la caché tampoco puede ser demasiado grande. En primer lugar, porque se integra en el mismo chip que el procesador, lo que implica que hay que tener en cuenta el consumo de área y el de

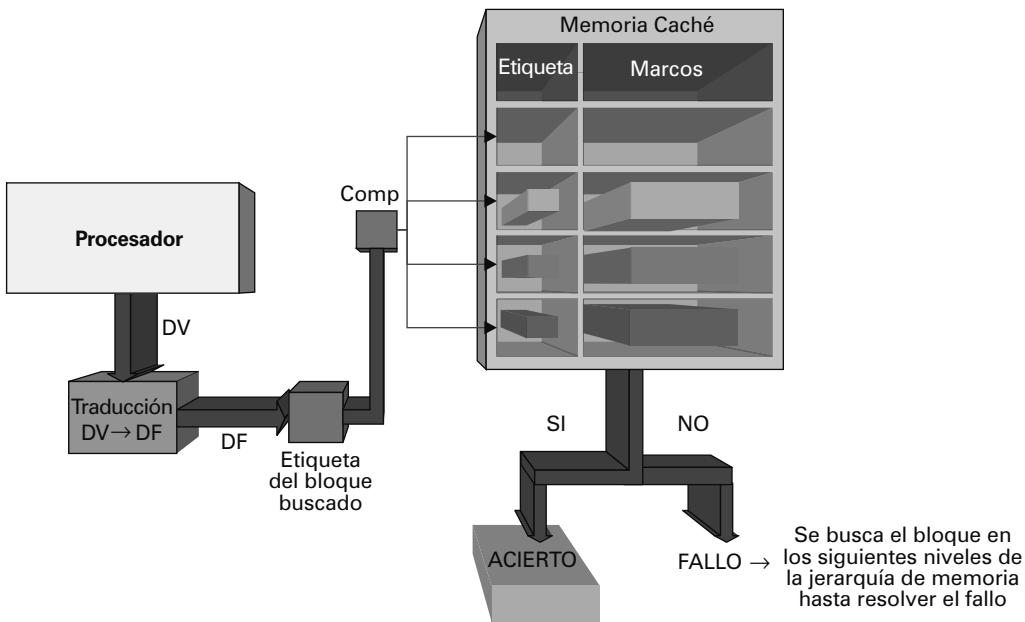


FIGURA 2.5

Funcionamiento de la memoria caché.

potencia, y en general, las restricciones en el coste del diseño. Y en segundo lugar, porque las cachés más grandes son más lentas, ya que la lógica necesaria para su gestión es más compleja, puede que esto implique más comparaciones de etiquetas, etc. Por lo tanto hay que escoger cuidadosamente el tamaño de esta memoria para cada arquitectura.

Una vez que se ha decidido el tamaño de la caché, es necesario escoger el tamaño de marco (y de bloque) que se va a manejar. De nuevo es una decisión delicada que afecta al rendimiento de la memoria. Utilizando bloques de gran tamaño se captura mejor la localidad espacial y se pueden reducir los fallos iniciales.

Pero hay que tener en cuenta que al aumentar el tamaño del bloque también se aumenta la penalización por fallo, ya que se necesita más tiempo para traer los bloques del siguiente nivel de la jerarquía de memoria.

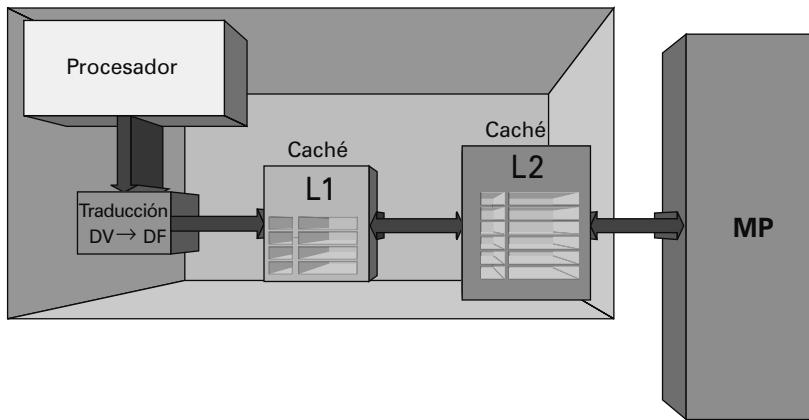
Habrá que llegar a un compromiso entre ambos aspectos teniendo en cuenta la latencia y el ancho de banda de la conexión con el siguiente nivel de memoria.

Otro de los aspectos fundamentales en la organización de la caché es la unificación o división de las instrucciones y los datos.

Según lo estudiado en el capítulo 1 de este libro, la segmentación del procesador obligará en muchos casos a dividir la caché de instrucciones y la de datos para evitar los riesgos estructurales entre las etapas de búsqueda de instrucción y las de acceso a memoria para lectura o escritura de operandos/resultados (las etapas de F y M del nanoMIPS). Pero cuando no se trate del primer nivel de memoria caché, se puede optar por seguir manteniendo esta separación o por unificar en bloques de información comunes las instrucciones y los datos.

Todas estas decisiones de diseño suelen tomarse con ayuda de herramientas de simulación y siguiendo reglas empíricas y heurísticas que han surgido de la experiencia de otros diseñadores.

Por último, destacar dentro de la organización de la caché uno de los aspectos que más influye en el rendimiento de la jerarquía de memoria: la implementación de cachés multinivel.

**FIGURA 2.6**

Utilización de dos niveles de memoria caché.

Si se opta por utilizar un único nivel de memoria caché, cuando este nivel falle, habrá que ir directamente a memoria principal a buscar el bloque necesario para resolver este fallo. La penalización que implica buscar este bloque y traerlo a la memoria caché puede ser excesiva en muchos casos.

Además, cuando se diseña una memoria caché de un único nivel se plantea la siguiente cuestión, ¿es mejor una memoria caché pequeña y rápida que pueda trabajar a la velocidad del procesador o una memoria de tamaño mayor, más lenta pero que capture y aproveche mejor el principio de localidad y reduzca la tasa de fallos?

La solución a las dos cuestiones es casi siempre la utilización de más de un nivel de memoria caché (figura 2.6). Normalmente se utilizan dos niveles de memoria caché (aunque muchas arquitecturas de hoy en día empiezan a incorporar un tercero):

- **Nivel 1 (L1).** Este nivel es el más cercano al procesador, por lo tanto, esta memoria caché es pequeña y rápida.
- **Nivel 2 (L2).** Es el siguiente nivel de la jerarquía, de mayor tamaño (por lo que aprovecha mejor el principio de localidad) y por lo tanto, más lento aunque con menos fallos de capacidad.

De esta manera, cuando se produzca un fallo en el nivel 1 de la memoria, la penalización de este fallo será menor que si sólo hubiera un nivel de caché, porque en lugar de ir a la memoria principal irá a la caché de nivel 2 (siempre con menor tiempo de acceso que la memoria principal y además sin necesidad de salir del chip en el que se encuentra el procesador).

Ejemplo 2.2

Comparación entre una jerarquía de memoria con un nivel de caché con otra jerarquía de memoria con dos niveles de caché.

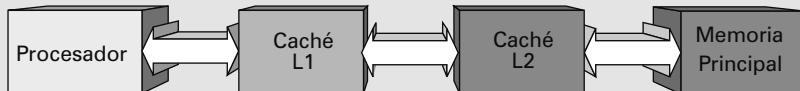
Inicialmente tenemos un procesador con un único nivel de memoria caché, unificada, con tiempo de acceso de 1 ns, tasa de fallos del 5% y penalización por fallo de 90 ns.



En esta situación el tiempo medio de acceso a memoria para realizar una lectura es:

$$t_{MEM}(\text{lectura}) = t_{\text{acceso } L1} + TF_{L1} \cdot pF_{L1} = 1 + 0.05 \cdot 90 = 5.5 \text{ ns}$$

Para mejorar las prestaciones de esta jerarquía de memoria se decide incluir un segundo nivel de memoria caché, también unificada, y en este caso con un tiempo de acceso de 12 ns y una tasa de fallos del 10%.



Gracias a la introducción de este nivel, la penalización por fallo que antes tenía el primer nivel de memoria caché, debida a la necesidad de traer un bloque desde memoria principal, ahora pasa a ser la penalización por fallo del segundo nivel de caché. El tiempo medio de acceso a memoria para lectura es (suponiendo que los movimientos de información entre los niveles de la jerarquía de memoria implican un tiempo despreciable):

$$t_{MEM}(\text{lectura}) = t_{\text{acceso } L1} + TF_{L1} \cdot pF_{L1} = 1 + 0.05 \cdot pF_{L1}$$

$$pF_{L1} = t_{L2} = t_{\text{acceso } L2} + TF_{L2} \cdot pF_{L2} = 12 + 0.1 \cdot 90 = 21 \text{ ns}$$

Por lo que:

$$t_{MEM}(\text{lectura}) = 1 + 0.05 \cdot pF_{L1} = 1 + 0.05 \cdot 21 = 2.05 \text{ ns}$$

Es decir, la ganancia que se obtiene gracias a la introducción de un segundo nivel de memoria caché es de $5.5/2.05 = 2.68$.

2.4.1.2. POLÍTICA DE EMPLAZAMIENTO

Si en la memoria caché sólo se almacenan unos determinados bloques de información, ¿cuálquier bloque de datos puede alojarse en cualquier marco de la memoria caché?

La respuesta es que depende el tipo de política de emplazamiento que se decida implementar en la memoria caché. Hay tres tipos de emplazamiento:

- **Directo.** A cada bloque le corresponde un único marco de caché y sólo puede alojarse en este marco.
- **Asociativo.** Un bloque puede alojarse en cualquier marco de la memoria caché.
- **Asociativo por conjuntos.** La memoria caché se divide en conjuntos con un número determinado de marcos por conjunto (o vías). A un determinado bloque le corresponde un único conjunto, pero dentro de él, puede alojarse en cualquier marco. Es decir, es una solución intermedia entre los dos tipos de emplazamiento anteriores.

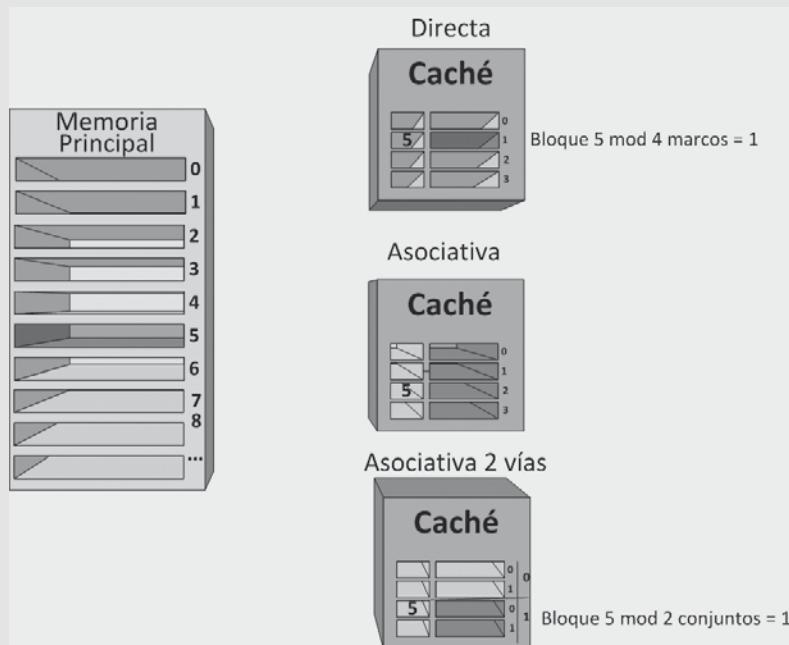
Ejemplo 2.3

Ejemplo de las tres políticas de emplazamiento.

Supongamos que tenemos una memoria caché muy sencilla de sólo cuatro marcos de bloque.

Veamos dónde se podría ubicar el bloque 5 de memoria principal en esta memoria caché si es directa, asociativa o asociativa por conjuntos de 2 vías.

En la figura aparecen sombreadas las posibles ubicaciones para el bloque 5 dentro de la caché.



Se puede observar que en el caso de la caché directa, una sencilla operación de módulo DIRECCIÓN DE BLOQUE mod NÚMERO DE MARCOS ($5 \text{ mod } 4 = 1$ en este caso), nos indica en qué marco de la caché debe ubicarse el bloque. Y esta es la única posibilidad de emplazamiento.

Sin embargo, en el caso de la memoria asociativa, el bloque 5 puede ubicarse en cualquiera de los cuatro marcos disponibles.

Por último, en el caso de la caché asociativa por conjuntos de 2 vías, de nuevo una operación de módulo nos indica en qué conjunto debe ubicarse el bloque: DIRECCIÓN DE BLOQUE mod NÚMERO DE CONJUNTOS ($5 \text{ mod } 2 = 1$ para el ejemplo). Dentro del conjunto 1, el bloque 5 puede ubicarse en cualquiera de los dos marcos disponibles.

Las diferentes alternativas para el emplazamiento en la memoria caché llevan a diferentes interpretaciones de la dirección física desde el punto de vista de esta memoria (figura 2.7). Según el tipo de emplazamiento de la memoria caché:

- **Directo.** El índice indica el marco que le corresponde a ese bloque de memoria.
- **Asociativo.** No existe este campo en la dirección, ya que el bloque puede alojarse en cualquier marco.
- **Asociativo por conjuntos.** El índice indica el conjunto que le corresponde a ese bloque de memoria.

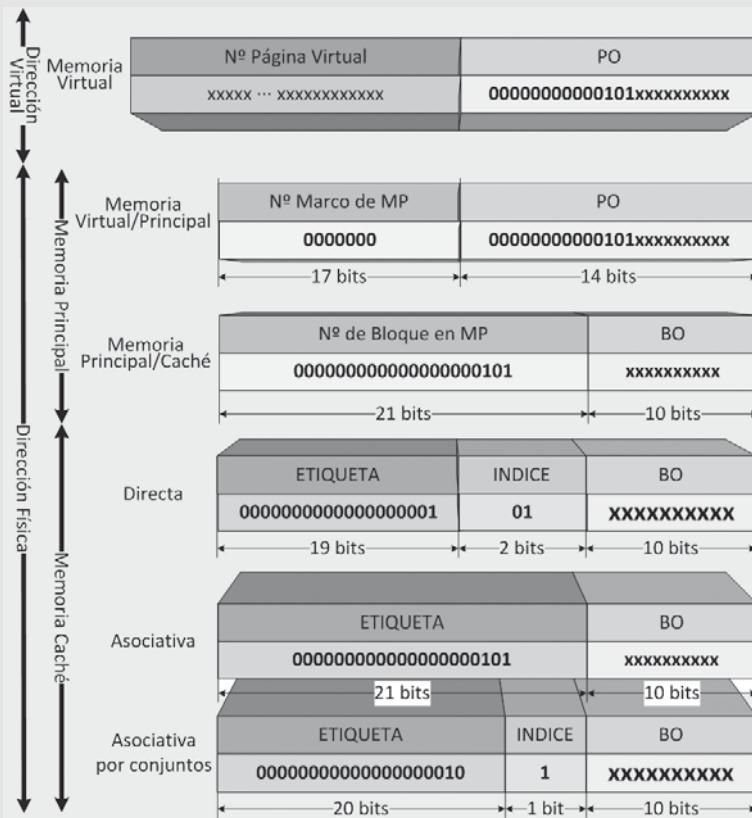
Ejemplo 2.4

Ejemplo de las tres políticas de emplazamiento utilizando direcciones de memoria.

Si volvemos al ejemplo 2.3, podemos realizar la misma discusión, pero utilizando en este caso la dirección física del bloque 5 de memoria principal.

2. CONCEPTOS BÁSICOS DE MEMORIA Y E/S

Supongamos que la memoria principal es de 2 GB (2^{31} B), que el tamaño de página es de 16 KB (2^{14} B) y que el tamaño de bloque es de 256 palabras (1024 o 2^{10} B). Entonces cualquier dirección que corresponda al bloque 5 de memoria tendrá el formato que se muestra en la figura.



Si la caché es de emplazamiento directo, al interpretar la dirección física desde el punto de vista de la caché, el índice será de 2 bits (ya que hay en la caché 2^2 marcos diferentes).

Y del índice se deduce que el bloque 5 debe ubicarse siempre en el marco 1 de caché.

En el caso de la cache asociativa, toda la dirección de bloque es etiqueta y no hay índice, ya que el bloque puede ubicarse en cualquier marco de la memoria caché.

Por último, en el caso de la memoria asociativa por conjuntos de 2 vías, el índice necesita 1 bit (tenemos 2^1 conjuntos diferentes en la caché). Y el valor del índice nos indica que el bloque 5 se debe ubicar siempre en un marco del conjunto 1 de la memoria caché.



FIGURA 2.7

Interpretación de la dirección física desde el punto de vista de la memoria caché.

En cuanto al rendimiento de los distintos tipos de emplazamiento, las memorias caché de mayor asociatividad reducen los fallos por conflicto. Pero son más lentas, porque cuanto mayor sea la asociatividad mayor será la longitud de las etiquetas que deben almacenarse y compararse. Y más comparaciones deberán realizarse.

Existen dos reglas empíricas que se utilizan mucho para escoger la asociatividad de la memoria caché:

- Una caché asociativa por conjuntos de 8 vías (8 marcos por conjunto) se comporta, en cuanto a la tasa de fallos, igual que una memoria completamente asociativa. Es decir, no compensa que las memorias sean completamente asociativas, ya que tienen los mismos fallos que las memorias asociativas por conjuntos de 8 vías pero son más lentas.
- Una caché directa de tamaño N tiene la misma tasa de fallos que una asociativa por conjuntos de 2 vías (2 marcos por conjunto) de tamaño $N/2$.

Y los simuladores son de nuevo herramientas muy potentes para tomar este tipo de decisión.

2.4.1.3. POLÍTICA DE REEMPLAZAMIENTO

Además de decidir la política de emplazamiento de la memoria caché, también es necesario definir qué ocurre cuando se produce un fallo. Si no se encuentra un determinado bloque de memoria en la caché, habrá que traerlo del siguiente nivel de la jerarquía de memoria. Pero, ¿qué bloque se reemplaza?

En el caso de una caché directa, no cabe ninguna duda, puesto que cada bloque sólo puede alojarse en un determinado marco. Pero en el caso de memorias con una asociatividad mayor, el reemplazamiento puede hacerse con distintos tipos de políticas.

Estas son las más comunes:

- **Aleatoriedad.** Se utiliza un generador de números aleatorios para escoger el bloque que se reemplaza.
- **LRU (Least Recently Used).** Se escoge el bloque que lleva más tiempo sin utilizarse. Así se minimiza la probabilidad de sustituir un bloque que vaya a necesitarse en el futuro (siguiendo el principio de localidad temporal).
- **FIFO (First In, First Out).** Se reemplaza el bloque que lleva más tiempo en la caché.

2.4.1.4. POLÍTICA DE ESCRITURA

Las lecturas son mucho más frecuentes que las escrituras en la memoria caché, pero no hay que desducidar estas últimas ya que suelen ser más costosas.

Las escrituras llevan más tiempo ya que no pueden solaparse con la comparación de las etiquetas como se hace con las lecturas. En muchos casos se recupera la información de un marco determinado de caché antes de saber si el bloque que está ubicado en él es el que se está buscando. Esto sólo puede hacerse con operaciones de lectura, ya que si finalmente el bloque no es el adecuado, no se ha modificado su contenido.

Pero una operación de escritura sólo puede hacerse si la comparación de etiquetas da como resultado que el bloque ubicado en ese marco es el que se estaba buscando; no se puede hacer trabajo en paralelo.

Hay dos formas de plantear las escrituras en la memoria caché:

- **Escritura Directa.** Cuando se modifica una palabra con una instrucción de almacenamiento, se realiza la escritura en el primer nivel de caché y en el siguiente nivel de la jerarquía.
- **Post-Escritura.** Cuando se modifica una palabra sólo se hace en el primer nivel de la memoria caché. Cuando este bloque modificado sea reemplazado, se actualizará el contenido del nivel siguien-

te. Para no actualizar la memoria con cada reemplazamiento que se haga (puede ser innecesario si en ese bloque no se ha llevado a cabo ninguna escritura), suele utilizarse un bit que indica si el bloque que está en caché ha sido modificado (sucio) y por lo tanto debe actualizarse la memoria, o no (limpio).

La política de escritura directa garantiza que las diferentes copias de una misma información que hay en la jerarquía de memoria se mantienen coherentes en todo momento y además es fácil de implementar.

Pero la política de post-escritura permite hacer escrituras consecutivas en un mismo bloque sin actualizar cada vez el siguiente nivel de la jerarquía, sólo se actualiza cuando el bloque es reemplazado. Esto permite consumir, por norma general, mucho menos ancho de banda del siguiente nivel de la jerarquía, aunque implica que durante un tiempo las escrituras sólo están almacenadas en un único nivel de la jerarquía, por lo que mantener la coherencia implica complicar el hardware.

Ejemplo 2.5

Comparación del ancho de banda de mejora principal consumido con las dos políticas de escritura.

Tenemos un procesador cuya frecuencia de reloj es de 2.5 GHz que realiza una media de $2 \cdot 10^7$ accesos a memoria por segundo. La jerarquía de memoria está compuesta por un único nivel de memoria caché unificada, con tasa de fallos del 12% y con un tamaño de bloque de 16 palabras. En este sistema, en media, un 70% de los accesos a memoria son lecturas y un 30% son escrituras.

Si la memoria caché utiliza escritura directa, podemos calcular el ancho de banda de memoria principal consumido haciendo las siguientes consideraciones:

Tipo de acceso	Acciones	Tráfico con MP
Acierto de lectura	Lectura en la memoria caché.	0
Fallo de lectura	Hay que traer 1 bloque de memoria principal para resolver el fallo. Lectura en la memoria caché.	16 palabras
Acierto de escritura	Escritura en la memoria caché y en la memoria principal (escritura directa).	1 palabra
Fallo de escritura	Hay que traer 1 bloque de memoria principal para resolver el fallo. Escribir en la memoria caché y en la memoria principal (escritura directa).	17 palabras

Por lo tanto el ancho de banda consumido es:

$$\begin{aligned}
 AB_{MP} &= \text{referencias a memoria por segundo} \\
 &\cdot (\% \text{fallos de lectura} \cdot 16 + \% \text{aciertos de escritura} \cdot 1 + \% \text{fallos de escritura} \cdot 17) \\
 &= 2 \cdot 10^7 \cdot (0.12 \cdot 0.7 \cdot 16 + 0.88 \cdot 0.3 \cdot 1 + 0.12 \cdot 0.3 \cdot 17) = 4.44 \cdot 10^7 \text{ palabras por segundo}
 \end{aligned}$$

Si por el contrario la memoria caché se diseña para utilizar una política de post-escritura, teniendo en cuenta que el 10% de los bloques son modificados mientras se encuentran ubicados en la caché:

Tipo de acceso	Acciones	Tráfico con MP
Acierto de lectura	Lectura en la memoria caché.	0
Fallo de lectura con reemplazamiento de bloque limpio (no ha sido modificado mientras estaba en memoria caché)	Hay que traer 1 bloque de memoria principal para resolver el fallo. Lectura en la memoria caché.	16 palabras
Fallo de lectura con reemplazamiento de bloque sucio (ha sido modificado mientras estaba en memoria caché)	Hay que llevar el bloque sucio a memoria principal para actualizar todas las escrituras que se habían hecho en memoria caché. Hay que traer 1 bloque de memoria principal para resolver el fallo. Lectura en la memoria caché.	32 palabras
Acierto de escritura	Escritura en la memoria caché.	0
Fallo de escritura con reemplazamiento de bloque limpio (no ha sido modificado mientras estaba en memoria caché)	Hay que traer 1 bloque de memoria principal para resolver el fallo. Escrutura en la memoria caché.	16 palabras
Fallo de escritura con reemplazamiento de bloque sucio (ha sido modificado mientras estaba en memoria caché)	Hay que llevar el bloque sucio a memoria principal para actualizar todas las escrituras que se habían hecho en memoria caché. Hay que traer 1 bloque de memoria principal para resolver el fallo. Escritura en la memoria caché.	32 palabras

Es decir, en el caso de la post-escritura, las lecturas y las escrituras tienen el mismo comportamiento respecto al consumo de ancho de banda de memoria principal. Tenemos:

$$\begin{aligned}
 AB_{MP} &= \text{referencias a memoria por segundo} \\
 &\cdot (\% \text{fallos con reemplazamiento de bloque limpio} \cdot 16 \\
 &+ \% \text{fallos con reemplazamiento de bloque sucio} \cdot 32) \\
 &= 2 \cdot 10^7 \cdot (0.12 \cdot 0.9 \cdot 16 + 0.12 \cdot 0.1 \cdot 32) \\
 &= 4.22 \cdot 10^7 \text{ palabras por segundo}
 \end{aligned}$$

Ejemplo 2.6

Comparación del tiempo medio de acceso a memoria con las dos políticas de escritura.

En el mismo caso del ejemplo anterior, vamos a calcular los tiempos medios de acceso a memoria con cada una de las políticas de escritura. Supongamos para ello que el tiempo de acceso a la memoria caché es de 4 ns, la latencia de acceso a memoria principal de 85 ns y que transferir una palabra entre la memoria principal y la memoria caché supone 0.5 ns (t_{bus}).

En el caso de la memoria caché con escritura directa tenemos:

$$t_{MEM}(\text{lectura}) = t_{acierto} + TF \cdot pF$$

$$t_{MEM}(\text{escritura}) = t_{acierto} + \text{latencia}_{MP} + TF \cdot pF$$

En ambos casos la penalización por fallo, teniendo en cuenta que el tamaño de bloque es de 16 palabras es, suponiendo que en cada acceso a memoria principal se recupera una única palabra:

$$pF = 16 \cdot (\text{latencia}_{MP} + t_{bus}) = 16 \cdot (85 + 0.5) = 1368 \text{ ns}$$

Aunque todavía no hemos estudiado optimizaciones para la jerarquía de memoria, sí que podemos imaginar que las latencias de acceso a memoria se pueden ir solapando con las transferencias por el bus, así que podemos suponer que:

$$pF = 16 \cdot (\text{latencia}_{MP}) = 16 \cdot (85) = 1360 \text{ ns}$$

Y:

$$t_{MEM}(\text{lectura}) = t_{acuerdo} + TF \cdot pF = 4 + 0.12 \cdot 1360 = 167.2 \text{ ns}$$

$$t_{MEM}(\text{escritura}) = t_{acuerdo} + \text{latencia}_{MP} + TF \cdot pF = 4 + 85 + 0.12 \cdot 1360 = 252.2 \text{ ns}$$

En media:

$$\begin{aligned} t_{MEM} &= \% \text{ lectura} \cdot t_{MEM}(\text{lectura}) + \% \text{ escritura} \cdot t_{MEM}(\text{escritura}) \\ &= 0.7 \cdot 167.2 + 0.3 \cdot 252.2 = 192.7 \text{ ns} \end{aligned}$$

En el caso de la post-escritura tenemos el mismo comportamiento para las lecturas y las escrituras:

$$t_{MEM} = t_{acuerdo} + TF \cdot pF$$

Pero la penalización por fallo se modifica respecto de la escritura directa:

$$\begin{aligned} pF &= 16 \cdot (\text{latencia}_{MP}) + \% \text{ sucios} \cdot 16 \cdot (\text{latencia}_{MP}) \\ &= 16 \cdot (1 + \% \text{ sucios}) \cdot \text{latencia}_{MP} = 16 \cdot 1.1 \cdot 85 \\ &= 1496 \text{ ns} \end{aligned}$$

Por lo que:

$$t_{MEM} = t_{acuerdo} + TF \cdot pF = 4 + 0.12 \cdot 1496 = 183.52 \text{ ns}$$

Se observa que las prestaciones de la memoria caché con la política de post-escritura son algo mejores que con la política de escritura directa, pero en cualquier caso, el tiempo medio de acceso a memoria que se consigue sin ninguna optimización es superior a la latencia de memoria principal, por lo que, tal y como se ha diseñado esta jerarquía de memoria, la introducción de un nivel de memoria caché no consigue mejorar los tiempos medios de acceso.

Ejemplo 2.7

Diseño de una jerarquía de memoria con dos niveles de caché: el primer nivel de escritura directa y el segundo nivel de post-escritura.

Supongamos ahora que tenemos dos niveles de memoria caché, ambos unificados. La caché de nivel 1 tiene un tamaño de bloque de 8 palabras, un tiempo de acceso de 1 ns y una tasa de fallos del 5%. La caché de nivel 2 tiene un tamaño de bloque de 16 palabras, un tiempo de acceso de 9 ns y una tasa de fallos del 9%.

Además la caché de nivel 1 utiliza escritura directa y la caché de nivel 2 utiliza post-escritura, con un 26% de bloques que son modificados mientras están ubicados en ella.

La transferencia de una palabra entre memoria principal y la caché de nivel 2 supone 0.5 ns y la transferencia de una palabra entre la memoria caché de nivel 2 y la de nivel 1 supone 0.1 ns, y estos tiempos deben tenerse en cuenta para realizar los cálculos (no se solapan accesos y transferencias). La latencia de acceso a memoria principal es de 85 ns.

Con todos estos datos podemos calcular los tiempos medios de acceso a memoria para lectura y escritura, teniendo en cuenta que el primer nivel de caché es de escritura directa:

$$t_{MEM}(\text{lectura}) = t_{acuerdoL1} + TF_{L1} \cdot pF_{L1}$$

$$t_{MEM}(\text{escritura}) = t_{acuerdoL1} + t_{acuerdoL2} + TF_{L1} \cdot pF_{L1}$$

En ambos casos la penalización por fallo del nivel 1 es la misma:

$$pF_{L1} = 8 \cdot (t_{L2} + t_{bus\ L1L2})$$

El tiempo medio de acceso a la caché de nivel 2 (ya que cuando la caché de nivel 1 falla y se busca el bloque en la caché de nivel 2, ésta puede acertar o fallar), se calcula de la misma manera:

$$t_{L2} = t_{acuerdoL2} + TF_{L2} \cdot pF_{L2}$$

Y como la caché de nivel 2 es de post-escritura:

$$pF_{L2} = 16 \cdot (1 + \%sucios) \cdot (latencia_{MP} + t_{bus\ L2MP})$$

* Cuidado, cuando hay más de un nivel de caché, se manejan distintos tamaños de bloque a lo largo de la jerarquía de memoria, por eso en esta expresión se utiliza el tamaño de bloque de 16 palabras, mientras que en la de pF_{L1} se utilizaba el tamaño de bloque de 8 palabras.

Por lo tanto:

$$pF_{L2} = 16 \cdot (1 + 0.26) \cdot (85 + 0.5) = 1723.68 \text{ ns}$$

$$t_{L2} = 9 + 0.09 \cdot 1723.68 = 164.13 \text{ ns}$$

$$pF_{L1} = 8 \cdot (164.13 + 0.1) = 1313.85 \text{ ns}$$

$$t_{MEM}(\text{lectura}) = 1 + 0.05 \cdot 1313.85 = 66.69 \text{ ns}$$

$$\begin{aligned} t_{MEM}(\text{escritura}) &= t_{acuerdoL1} + t_{acuerdoL2} + TF_{L1} \cdot pF_{L1} \\ &= 1 + 9 + 0.05 \cdot 1313.85 = 75.69 \text{ ns} \end{aligned}$$

En media, si de nuevo tenemos que un 70% de los accesos son lecturas y un 30% son escrituras:

$$\begin{aligned} t_{MEM} &= \%lectura \cdot t_{MEM}(\text{lectura}) + \%escritura \cdot t_{MEM}(\text{escritura}) \\ &= 0.7 \cdot 66.69 + 0.3 \cdot 75.69 = 69.39 \text{ ns} \end{aligned}$$

Es decir, gracias a la introducción de un segundo nivel de caché se consigue que la jerarquía de memoria obtenga un tiempo medio de acceso a memoria por debajo de la latencia de memoria principal. Aún así, se pasa de los 85 ns que tarda en realizarse un acceso a memoria principal, a 69.39 ns con la jerarquía de dos niveles de caché, no compensa introducir toda la complejidad de gestión que suponen estas memorias caché. Veremos un poco más adelante que existen técnicas de diseño muy sencillas que pueden mejorar significativamente estos tiempos.

Sea cual sea la política de escritura escogida, casi todas las jerarquías de memoria actuales incorporan una estructura hardware denominada buffer de escritura para reducir la penalización extra que una escritura suele implicar. En este buffer se realizan las escrituras en primera instancia para luego poder solapar la escritura en el siguiente nivel de memoria desde este buffer con la ejecución de las siguientes instrucciones. Obviamente, el buffer es una estructura pequeña y rápida, organizada como una memoria caché, por lo que realizar las escrituras en el buffer de escritura implica una penalización mucho menor que hacerlas directamente en el siguiente nivel de la jerarquía de memoria.

Hay dos posibilidades de utilización del buffer de escritura según el tipo de política de escritura escogida:

- **Escritura directa.** Cuando se utiliza un buffer de escritura en memorias con escritura directa, las escrituras se hacen palabra a palabra en memoria caché y en este buffer (en lugar de en el siguiente nivel de la jerarquía). El contenido de este buffer se volcará al siguiente nivel para actualizarlo en dos situaciones: cuando esté lleno o cuando se produzca un fallo de lectura y sea necesario actualizar el siguiente nivel con el contenido del buffer antes de resolver ese fallo (figura 2.8).

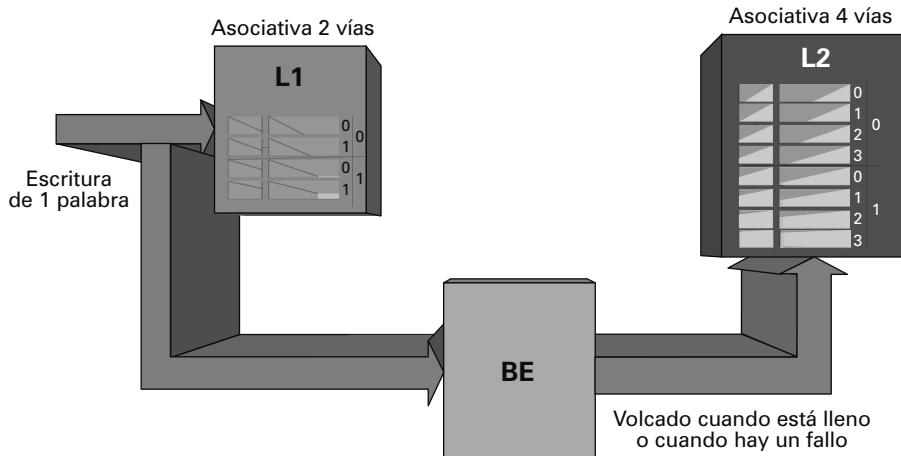


FIGURA 2.8

Utilización de un buffer de escritura con una caché de escritura directa.

- **Post-escritura:** En el caso de las memorias con post-escritura, el buffer de escritura se utiliza para volcar los bloques sucios que van a ser reemplazados en memoria caché (figura 2.9). De esta forma, cuando se trae un bloque a la caché que reemplaza a un bloque sucio, no hay que esperar a que se escriba el bloque sucio en el siguiente nivel, sino que se almacena temporalmente en el buffer. En realidad la utilización del buffer en estos casos suele considerarse una optimización de la memoria caché que permite reducir la penalización por fallo (dar prioridad a los fallos), por lo que volveremos a hablar de este caso en el capítulo 4 de este libro. De nuevo este buffer se vacía en dos situaciones: cuando se llena o cuando es necesario actualizar el siguiente nivel de la jerarquía de memoria antes de resolver un fallo.

En algunos casos, el buffer de escritura es más sofisticado y aprovecha para actualizar el siguiente nivel de la jerarquía de memoria en los momentos de inactividad.

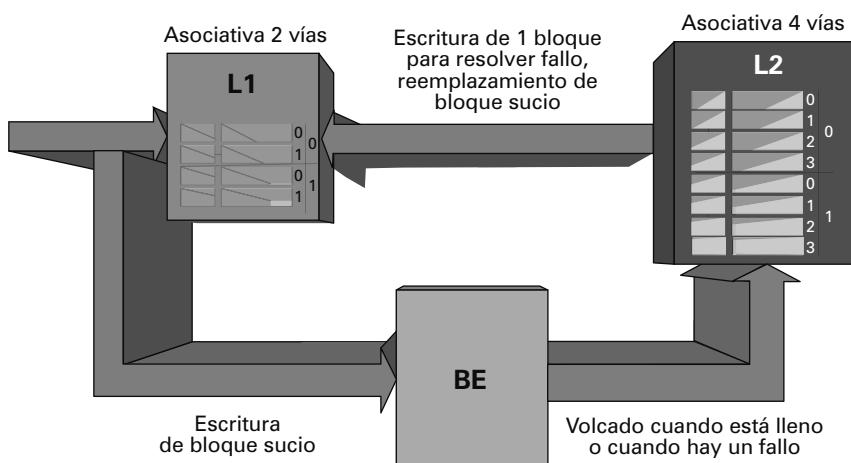


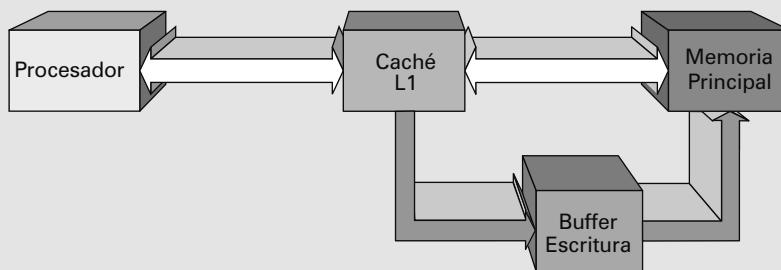
FIGURA 2.9

Utilización de un buffer de escritura con una caché de post-escritura.

Ejemplo 2.8**Ejemplo de utilización de un buffer de escritura con una caché de escritura directa.**

Supongamos que en el ejemplo 2.5, en el caso de la escritura directa, incluimos en el diseño de la jerarquía de memoria un buffer de escritura que en el 85% de los casos evita tener que realizar las escrituras en la memoria principal.

Es decir, el 85% de las veces que se va a realizar la escritura directa en memoria principal, se puede hacer en el buffer (cuyo tiempo de acceso es despreciable si se compara con la latencia de acceso de memoria principal) sin consumir ancho de banda de memoria principal. El otro 15% de las veces, el buffer está lleno u ocupado con tareas de gestión y/o actualización, por lo que no queda más remedio que realizar la escritura directa en memoria principal.



Sin utilizar buffer de escritura, el ancho de banda que se consumía era de $4.44 \cdot 10^7$ palabras/s. Veamos ahora:

Tipo de acceso	Acciones	Tráfico con MP
Acierto de lectura	Lectura en la memoria caché.	0
Fallo de lectura	Hay que traer 1 bloque de memoria principal para resolver el fallo. Lectura en la memoria caché.	16 palabras
Acierto de escritura con acierto del buffer de escritura	Escritura en la memoria caché y en el buffer de escritura (escritura directa).	0
Acierto de escritura con fallo del buffer de escritura	Escritura en la memoria caché y en la memoria principal (escritura directa).	1 palabra
Fallo de escritura con acierto del buffer de escritura	Hay que traer 1 bloque de memoria principal para resolver el fallo. Escrivtura en la memoria caché y en el buffer de escritura (escritura directa).	16 palabras
Fallo de escritura con fallo del buffer de escritura	Hay que traer 1 bloque de memoria principal para resolver el fallo. Escrivtura en la memoria caché y en la memoria principal (escritura directa).	17 palabras

Por lo tanto el ancho de banda consumido es:

$$AB_{MP} = \text{referencias a memoria por segundo} \cdot (\% \text{fallos} \cdot 16 + \% \text{escritura con fallo de BE} \cdot 1) = \\ 2 \cdot 10^7 \cdot (0.12 \cdot 16 + 0.3 \cdot 0.15 \cdot 1) = 3.93 \cdot 10^7 \text{ palabras por segundo}$$

Es decir, gracias al buffer de escritura se consume menos ancho de banda con memoria principal que en el caso de la escritura directa ($4.44 \cdot 10^7 / 3.93 \cdot 10^7 = 1.13$).

Ejemplo 2.9**Comparación con el tiempo medio de acceso a memoria con escritura directa sin y con buffer de escritura**

En el ejemplo 2.6 ya calculamos el tiempo medio de acceso a memoria con una caché de escritura directa sin buffer de escritura, veamos ahora cómo afecta a este tiempo la inclusión de este buffer.

Sólo afectará el tiempo medio de acceso para escritura, vamos a suponer que el tiempo de escritura en el buffer es despreciable si se compara con la latencia de memoria principal ($t_{BE} = 0$):

$$t_{MEM}(\text{escritura}) = t_{\text{acuerdo}} + TF_{BE}(\text{latencia}_{MP}) + (1 - TF_{BE}) \cdot t_{BE} + TF \cdot pF$$

Es decir, sólo se escribe en memoria principal cuando falla el buffer de escritura, el resto de escrituras directas se hacen sobre el buffer sin ninguna penalización en este caso.

La penalización por fallo no cambia respecto del caso sin buffer:

$$pF = 16 \cdot (\text{latencia}_{MP}) = 16 \cdot (85) = 1360 \text{ ns}$$

Y:

$$t_{MEM} = (\text{lectura}) = 167.2 \text{ ns}$$

$$\begin{aligned} t_{MEM}(\text{escritura}) &= t_{\text{acuerdo}} + TF_{BE}(\text{latencia}_{MP}) + (1 - TF_{BE}) \cdot t_{BE} + TF \cdot pF = \\ &4 + 0.15 \cdot (85) + 0.12 \cdot 1360 = 179.95 \text{ ns} \end{aligned}$$

Y en media:

$$t_{MEM} = \% \text{lectura} \cdot t_{MEM}(\text{lectura}) + \% \text{escritura} \cdot t_{MEM}(\text{escritura}) = 0.7 \cdot 167.2 + 0.3 \cdot 179.95 = 171.02 \text{ ns}$$

Es decir, gracias al buffer de escritura, el tiempo medio de acceso a memoria se mejora en un 13% ($192.7/171.02 = 1.13$). La mejora no es más significativa porque las escrituras son poco frecuentes.

Por último, en el caso de algunas jerarquías de memoria, para evitar la penalización extra que las escrituras implican se ignoran los fallos de escritura, especialmente en el caso de las memorias de escritura directa.

Por lo tanto suelen encontrarse dos opciones:

- **Memoria caché con asignación en escritura.** Los fallos de escritura se comportan como los fallos de lectura, se trae el bloque que ha provocado el fallo a la memoria caché y se realiza la escritura en función de la política utilizada.
- **Memoria caché sin asignación en escritura.** Cuando se produce un fallo de escritura, el bloque se modifica directamente en el siguiente nivel de la jerarquía de memoria y no se trae a la caché. Sólo se llevan a la caché los bloques que se solicitan para lectura, no para escritura.

Aunque cualquier combinación es posible, las cachés con política de escritura directa son las que suelen ser sin asignación en escritura, mientras que las cachés con post-escritura suelen ser con asignación.

Ejemplo 2.10**Ejemplo de caché con escritura directa sin asignación en escritura.**

Si utilizamos de nuevo el ejemplo 2.5 en el caso de escritura directa, veamos cómo se modificaría el ancho de banda con memoria principal consumido si no se resolvieran los fallos de escritura:

Tipo de acceso	Acciones	Tráfico con MP
Acierto de lectura	Lectura en la memoria caché.	0
Fallo de lectura	Hay que traer 1 bloque de memoria principal para resolver el fallo. Lectura en la memoria caché.	16 palabras
Acierto de escritura	Escritura en la memoria caché y en la memoria principal (escritura directa).	1 palabra
Fallo de escritura	Escritura en la memoria principal (escritura directa).	1 palabra

Por lo tanto el ancho de banda consumido es:

$$AB_{MP} = \text{referencias a memoria por segundo} \cdot (\% \text{fallos de lectura} \cdot 16 + \% \text{escritura} \cdot 1) = \\ 2 \cdot 10^7 \cdot (0.12 \cdot 0.7 \cdot 16 + 0.3 \cdot 1) = 3.29 \cdot 10^7 \text{ palabras por segundo}$$

En este caso conseguimos una mejora de un 35% ($4.44 \cdot 10^7 / 3.29 \cdot 10^7 = 1.35$).

Ejemplo 2.11

Comparación del tiempo medio de acceso a memoria con escritura directa sin y con asignación en escritura

En el ejemplo 2.6 ya calculamos el tiempo medio de acceso a memoria con una caché de escritura directa con asignación en escritura, veamos ahora cómo afecta a este tiempo el no resolver los fallos de escritura.

Sólo afectará el tiempo medio de acceso para escritura, y tenemos:

$$t_{MEM}(\text{escritura}) = (1 - TF) \cdot (t_{\text{acuerdo}} + latencia_{MP}) + TF \cdot (latencia_{MP})$$

Es decir, si se acierta en escritura, se escribe en memoria caché y en memoria principal (porque es escritura directa). Pero si se produce un fallo de escritura, no se resuelve y se escribe directamente en memoria principal.

Tenemos:

$$t_{MEM}(\text{lectura}) = 167.2 \text{ ns}$$

$$t_{MEM}(\text{escritura}) = (1 - TF) \cdot (t_{\text{acuerdo}} + latencia_{MP}) + TF \cdot (latencia_{MP}) = 0.88 \cdot (4 + 85) + 0.12 \cdot (85) = 88.52 \text{ ns}$$

Y en media:

$$t_{MEM} = \% \text{lectura} \cdot t_{MEM}(\text{lectura}) + \% \text{escritura} \cdot t_{MEM}(\text{escritura}) = 0.7 \cdot 167.2 + 0.3 \cdot 88.52 = 143.60 \text{ ns}$$

Es decir, gracias a la utilización de una caché sin asignación en escritura, el tiempo medio de acceso a memoria se mejora en un 34% ($192.7 / 143.60 = 1.34$).

Ejemplo 2.12

Diseño de una jerarquía de memoria con dos niveles de caché: el primer nivel de escritura directa sin asignación en escritura y el segundo nivel de post-escritura con asignación en escritura

Si utilizamos los mismos datos que en el ejemplo 2.7, con dos niveles de memoria caché, tenemos:

$$t_{MEM}(\text{lectura}) = t_{\text{acuerdoL1}} + TF_{L1} \cdot pF_{L1}$$

$$t_{MEM}(\text{escritura}) = (1 - TF_{L1}) \cdot (t_{\text{acuerdoL1}} + t_{\text{acuerdoL2}}) + TF_{L1} \cdot (t_{L2})$$

Es decir, sólo se modifica el tiempo medio para las escrituras. Si la caché de nivel 1 acierta en escritura, la escritura se realiza en el nivel 1 y en el nivel 2 (escritura directa), en el que siempre se acierta por el principio de inclusión. Sin embargo, si la caché de nivel 1 tiene un fallo en escritura, este fallo no se resuelve y se escribe directamente en el nivel 2. Pero en este caso, ya no se sabe seguro si se acierta o se falla, por eso aparece el tiempo medio acceso al nivel 2:

$$t_{L2} = t_{\text{acuerdoL2}} + TF_{L2} \cdot pF_{L2}$$

Por lo tanto:

$$pF_{L2} = 16 \cdot (1 + 0.26) \cdot (85 + 0.5) = 1723.68 \text{ ns}$$

$$t_{L2} = 9 + 0.09 \cdot 1723.68 = 164.13 \text{ ns}$$

$$pF_{L1} = 8 \cdot (164.13 + 0.1) = 1313.85 \text{ ns}$$

$$t_{MEM}(\text{lectura}) = 1 + 0.05 \cdot 1313.85 = 66.69 \text{ ns}$$

$$t_{MEM}(\text{escritura}) = (1 - TF_{L1}) (t_{\text{acuerdoL1}} + t_{\text{acuerdoL2}}) + TF_{L1} \cdot (t_{L2}) = (1 - 0.05) \cdot (1 + 9) + 0.05 \cdot (164.13) = 17.71 \text{ ns}$$

En media, si de nuevo tenemos que un 70% de los accesos son lecturas y un 30% son escrituras:

$$t_{MEM} = \% \text{lectura} \cdot t_{MEM}(\text{lectura}) + \% \text{escritura} \cdot t_{MEM}(\text{escritura}) = 0.7 \cdot 66.69 + 0.3 \cdot 17.71 = 52 \text{ ns}$$

Así que conseguimos un speedup de 1.33 gracias a la utilización de una caché de nivel 1 sin asignación en escritura en el primer nivel de la jerarquía ($69.39/52 = 1.33$).

2.4.1.5. MEJORAS SENCILLAS

En los ejemplos de la sección anterior se ha conseguido mejorar el tiempo medio de acceso a memoria que se obtiene cuando sólo se utiliza la memoria principal cuando se han utilizado dos niveles de memoria caché, pero no cuando se ha utilizado un único nivel. E incluso al utilizar dos niveles de memoria caché, esta mejora respecto de la memoria principal no ha sido suficientemente significativa como para justificar la introducción de una jerarquía de memoria.

Estos primeros ejemplos, tremadamente simplificados, no incorporan las técnicas de diseño básicas que siempre utilizan las memorias caché actuales. Aunque en el capítulo 4 se estudiarán multitud de técnicas de mejora de rendimiento para la memoria caché, algunas de estas técnicas (al igual que ocurre con la segmentación del procesador) se consideran básicas y se utilizan directamente en casi todos los diseños.

Entre estas técnicas se encuentran las de palabra crítica primero y rearanque rápido. Estas estrategias de diseño aprovechan el hecho de que normalmente el procesador está esperando sólo por una palabra del bloque por el que se ha producido un fallo, no por el bloque completo.

Por lo tanto, no es necesario esperar a que el bloque completo se traiga del siguiente nivel de memoria para que el procesador pueda comenzar a trabajar con la palabra que necesita.

Estas técnicas sólo suponen un beneficio cuando el tamaño del bloque de caché es considerable, pero en estos casos, aumentan significativamente las prestaciones si la localidad espacial del código que se ejecuta no es muy acusada. De ser así, hay que tener en cuenta que la siguiente palabra que solicitará el procesador será justo la siguiente palabra del mismo bloque, por lo que al final con toda probabilidad habrá que esperar a traer a la caché el bloque completo.

La diferencia entre estas dos técnicas es la siguiente:

- **Palabra crítica primero.** Se envía primero a la caché la palabra que ha provocado el fallo para que el procesador pueda completar su acceso y seguir trabajando, y luego se transfiere el resto del bloque.
- **Rearranque rápido.** En este caso el bloque se trae en orden a la memoria caché, y en cuanto llega la palabra crítica (la que ha provocado el fallo), se permite que el procesador siga trabajando.

La técnica de palabra crítica primero consigue mejores prestaciones, ya que sólo hay que esperar a que se envíe una palabra a la memoria caché mientras que con rearranque rápido, en media, hay que esperar a que se envíe la mitad del bloque. Sin embargo, con palabra crítica primero el hardware es algo más complicado, ya que hay que gestionar el envío de bloques desordenados a memoria caché.

Además de estas técnicas también se aprovecha, siempre que sea posible, el paralelismo entre los accesos a memoria y las transferencias de información entre los diferentes niveles de la jerarquía así como la posibilidad de ir solapando las transferencias que se producen entre los distintos niveles para resolver un fallo. Por ejemplo, si se trabaja con dos niveles de caché, se puede ir enviando de la caché de nivel 2 a la caché de nivel 1 lo que se está trayendo a la caché de nivel 2 desde la memoria principal.

Ejemplo 2.13

Ejemplo de utilización de la técnica de palabra crítica primero.

Volvamos al ejemplo 2.6. Recordemos que en este ejemplo tenemos un procesador cuya frecuencia de reloj es de 2.5 GHz que realiza una media de $2 \cdot 10^7$ accesos a memoria por segundo.

La jerarquía de memoria de este diseño está compuesta por un único nivel de memoria caché unificada, con tasa de fallos del 12% y con un tamaño de bloque de 16 palabras. En este sistema, en media, un 70% de los accesos a memoria son lecturas y un 30% son escrituras. Supongamos que estamos en el caso de escritura directa.

Además, el tiempo de acceso a la memoria caché es de 4 ns y la latencia de memoria principal es de 85 ns. El tiempo necesario para transferir una palabra entre memoria principal y memoria caché es de 0.5 ns.

Con esta jerarquía sin ninguna optimización (simplemente suponiendo que se podían solapar los accesos a memoria con las transferencias de bus), calculamos que:

$$pF = 16 \cdot (latencia_{MP}) = 16 \cdot (85) = 1360 \text{ ns}$$

Y:

$$t_{MEM}(lectura) = t_{acceso} + TF \cdot pF = 4 + 0.12 \cdot 1360 = 167.2 \text{ ns}$$

$$t_{MEM}(escritura) = t_{acceso} + latencia_{MP} + TF \cdot pF = 4 + 85 + 0.12 \cdot 1360 = 252.2 \text{ ns}$$

Por lo que en media teníamos:

$$t_{MEM} = \%lectura \cdot t_{MEM}(lectura) + \%escritura \cdot t_{MEM}(escritura) = 0.7 \cdot 167.2 + 0.3 \cdot 252.2 = 192.7 \text{ ns}$$

Ya habíamos avanzado que este tiempo medio de acceso a memoria es completamente intolerable para un procesador actual y además es mayor que la latencia de acceso a memoria principal.

Sin embargo, si utilizamos palabra crítica primero, la penalización por fallo sólo incluye el tiempo que se tarda en recuperar de memoria principal y en enviar a la memoria caché la palabra que ha provocado el fallo,

no es necesario esperar a recuperar y enviar el resto de palabras del bloque para que el procesador pueda seguir trabajando.

Por lo tanto:

$$pF = 1 \cdot (\text{latencia}_{MP} + t_{bus}) = 1 \cdot (85 + 0.5) = 85.5 \text{ ns}$$

Y:

$$t_{MEM}(\text{lectura}) = t_{acuerdo} + TF \cdot pF = 4 + 0.12 \cdot 85.5 = 14.26 \text{ ns}$$

$$t_{MEM}(\text{escritura}) = t_{acuerdo} + \text{latencia}_{MP} + TF \cdot pF = 4 + 85 + 0.12 \cdot 85.5 = 99.26 \text{ ns}$$

En media:

$$t_{MEM} = \% \text{ lectura} \cdot t_{MEM}(\text{lectura}) + \% \text{ escritura} \cdot t_{MEM}(\text{escritura}) = 0.7 \cdot 14.26 + 0.3 \cdot 99.26 = 39.76 \text{ ns}$$

Es decir, conseguimos un speedup de 4.83 gracias a esta optimización ($192.85/39.76 = 4.83$).

Ejemplo 2.14

Diseño de una jerarquía de memoria con dos niveles de caché: el primer nivel de escritura directa sin asignación en escritura y el segundo nivel de post-escritura con asignación en escritura. Ambos utilizan la técnica de palabra crítica primero.

Retomemos ahora el ejemplo 2.12, en el que habíamos conseguido el mejor tiempo de acceso a memoria hasta el momento:

$$t_{MEM}(\text{lectura}) = t_{acuerdoL1} + TF_{L1} \cdot pF_{L1}$$

$$t_{MEM}(\text{escritura}) = (1 + TF_{L1}) \cdot (t_{acuerdoL1} + t_{acuerdoL2}) + TF_{L1} \cdot (t_{L2})$$

Si se utiliza la técnica de palabra crítica primero, en el cálculo de las penalizaciones por fallo sólo hay que tener en cuenta el tiempo que se tarda en traer una palabra a cada nivel de memoria (la que ha provocado el fallo) en lugar del tiempo que se tarda en traer un bloque completo. El resto del bloque se trae, pero el procesador ya puede estar trabajando mientras tanto.

Por tanto:

$$pF_{L1} = t_{L2} + t_{busL1L2}$$

$$t_{L2} = t_{acuerdoL2} + TF_{L2} \cdot pF_{L2}$$

$$pF_{L2} = (\text{latencia}_{MP} + t_{busL2MP}) + \% \text{ sucios} \cdot 16 \cdot (\text{latencia}_{MP} + t_{busL2MP})$$

Así que:

$$pF_{L2} = (85 + 0.5) + 0.26 \cdot 16 \cdot (85 + 0.5) = 441.18 \text{ ns}$$

$$t_{L2} = 9 + 0.09 \cdot 441.18 = 48.71 \text{ ns}$$

$$pF_{L1} = (48.71 + 0.1) = 48.81 \text{ ns}$$

$$t_{MEM}(\text{lectura}) = 1 + 0.05 \cdot 48.81 = 3.44 \text{ ns}$$

$$t_{MEM}(\text{escritura}) = (1 - TF_{L1}) \cdot (t_{acuerdoL1} + t_{acuerdoL2}) + TF_{L1} \cdot (t_{L2}) = (1 - 0.05) \cdot (1 + 9) + 0.05 \cdot (48.71) = 11.94 \text{ ns}$$

En media, si de nuevo tenemos que un 70% de los accesos son lecturas y un 30% son escrituras:

$$t_{MEM} = \% \text{ lectura} \cdot t_{MEM}(\text{lectura}) + \% \text{ escritura} \cdot t_{MEM}(\text{escritura}) = 0.7 \cdot 3.44 + 0.3 \cdot 11.94 = 5.99 \text{ ns}$$

Que se acerca mucho más a los tiempos de acceso necesarios para los sistemas actuales que los tiempos obtenidos en el ejemplo 2.12.

2.4.2. Diseño de la memoria principal

Los chips de memoria se componen siempre de matrices cuadradas de celdas DRAM, cada una de estas celdas almacena un bit y se compone de un transistor y un condensador (figura 2.10).

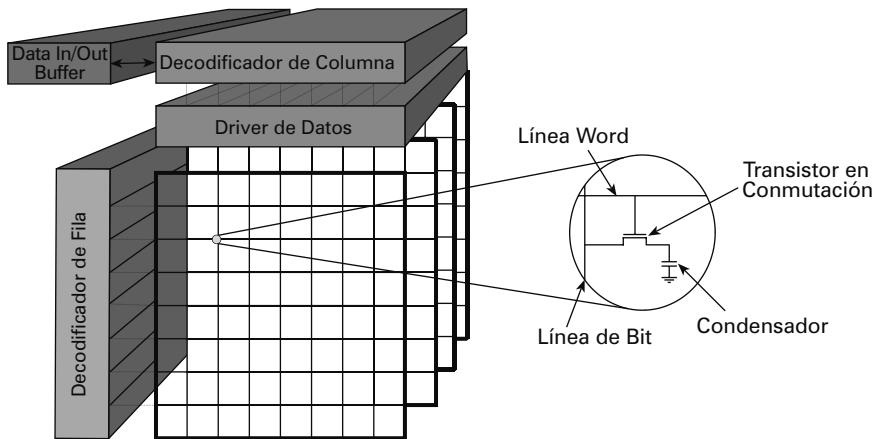


FIGURA 2.10

Organización de una memoria DRAM.

Cuando se produce un fallo en el último nivel de la memoria caché y se pasa a la memoria principal, casi siempre es necesario pasar por un controlador de memoria que gestione de manera adecuada los accesos a la memoria principal. Este controlador puede estar integrado dentro del propio procesador, o puede encontrarse en un chipset o hub externo.

Como se ha mencionado con anterioridad, el controlador de memoria se encarga de realizar la planificación del acceso y de mapear la dirección física a la ubicación física del dato en la memoria principal. Es decir, traduce la dirección física para saber qué chip o chips deben ser accedidos, qué matrices de memoria dentro de este chip o chips y en qué filas y columnas se encuentran los bits que deben recuperarse (en la figura 2.11 no se muestra la traducción de dirección virtual a física por simplificar, pero hay que recordar que el procesador no genera directamente direcciones físicas).

Para comprender el funcionamiento y el diseño de una memoria principal básica con un poco más de profundidad es necesario comprender la evolución de las diferentes tecnologías de memoria desde la DRAM convencional hasta las actuales memorias DDR3 o RDRAM.

Esto es debido a que en el caso de una memoria principal sin optimizar, la mayor parte de las decisiones están asociadas a la tecnología utilizada y por lo tanto, a la latencia de la memoria. En el capítulo 4 de este libro se estudiarán técnicas de diseño más sofisticadas que permitirán mejorar el rendimiento de estas memorias básicas en cuanto a su ancho de banda, y que no tendrán relación con la tecnología con la que se fabrica la memoria sino con su organización.

2.4.2.1. DRAM CONVENCIONAL

Este tipo de memoria ha quedado obsoleto, pero prácticamente todas las memorias que se utilizan en la actualidad provienen de realizar mejoras y optimizaciones a este modelo básico de funcionamiento para la memoria principal.

El procesador vuela una dirección en el bus de direcciones y el controlador de memoria decodifica esta dirección para determinar qué chip o chips deben ser accedidos y qué matrices de información dentro de estos chips (rango y banco).

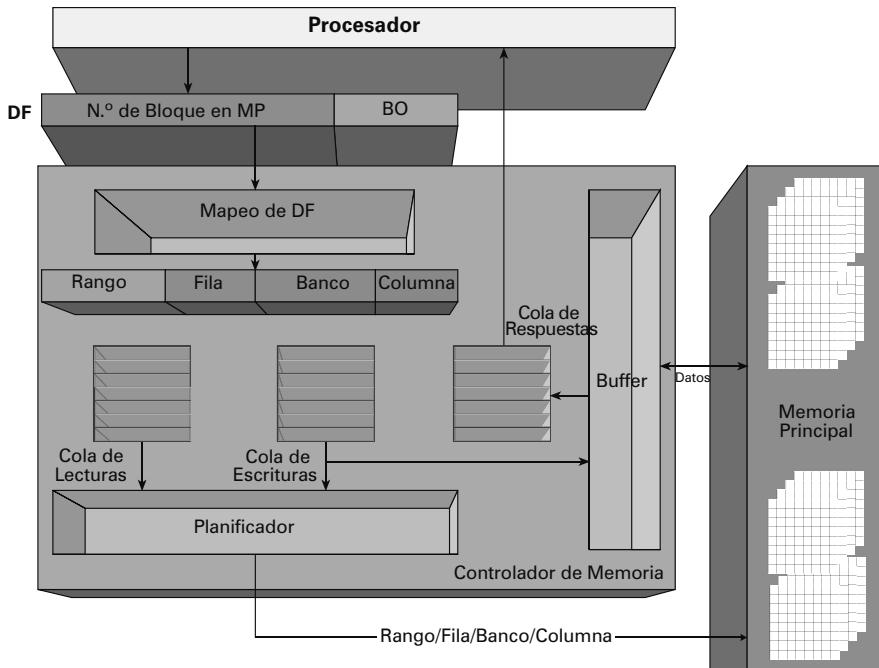


FIGURA 2.11
Organización del controlador de memoria.

A continuación, el controlador envía a estas matrices la dirección de fila y la valida mediante una señal de tipo RAS (Raw Address Strobe). Se recupera la fila completa y se envía la dirección de columna que corresponde a la información buscada, validándola de nuevo, en este caso con una señal de tipo CAS (Column Address Strobe).

La memoria DRAM convencional desaprovecha muchos recursos porque al direccionar una fila de una matriz, se recupera la fila completa, y a continuación se extrae la columna de esta fila a la que se quiere acceder (figura 2.12).

Pero si a continuación se accede a otra columna de la misma fila (algo muy común debido al principio de localidad), se vuelve a realizar el mismo proceso completo.

2.4.2.2 FPM DRAM

La FPM DRAM (Fast Page Mode DRAM) introduce dos mejoras a la DRAM convencional, la división de la memoria en páginas y los accesos en modo burst.

En primer lugar la memoria se divide en páginas y se introduce una electrónica que permite que los accesos que se encuentran en la misma página que un acceso previo se realicen con menos número de ciclos de espera. La manera de llevar esto a cabo es que cada página se corresponda justo con todas las celdas en una misma fila de la matriz del chip de memoria.

De esta manera, mientras los accesos se mantengan en la misma fila, se puede mantener la dirección de fila cargada y sólo hay que ir modificando la dirección de la columna.

En segundo lugar, los accesos en modo burst permiten que, una vez que se realiza el acceso a una determinada columna, se puede acceder a las tres siguientes columnas de esa fila sin necesidad de volver a cargar la fila en el driver de datos (figura 2.12). Las memorias típicas de esta tecnología son 5-3-3-3, es

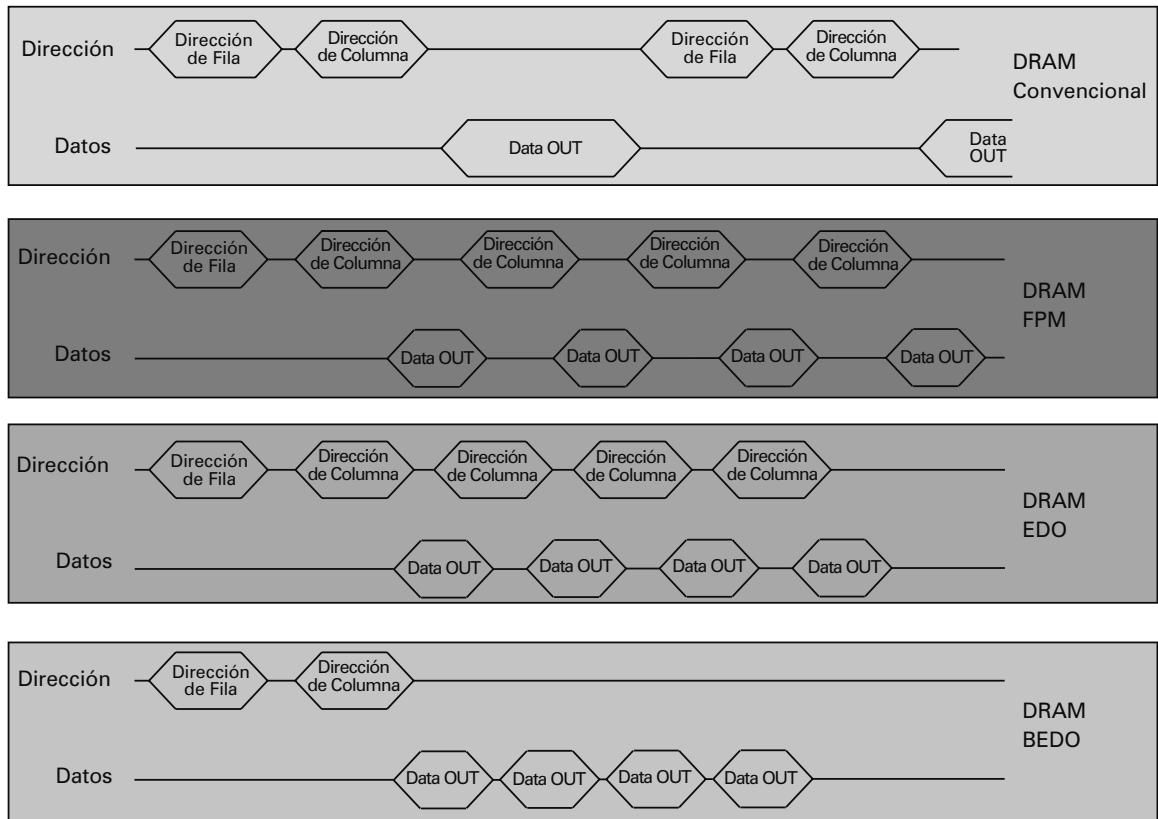


FIGURA 2.12

Cronogramas de acceso a memoria de las diferentes tecnologías asíncronas de memoria principal.

decir, el primer acceso tarda 5 ciclos y los tres siguientes tardan 3 ciclos porque ya no tienen que recuperar la fila completa (ya está disponible), sólo la columna.

2.4.2.3. EDO DRAM y BEDO DRAM

Las memorias Extended Data Out DRAM o EDO DRAM, mejoran la temporización de los accesos a memoria.

Con esta mejora se permite solapar la lectura del driver de datos por parte del controlador de memoria con el direccionamiento de la siguiente columna en los accesos en modo burst. Es decir, se puede comenzar un nuevo acceso sin que todavía haya finalizado el anterior (figura 2.12). Con esta técnica se consiguen memorias 5-2-2-2.

Obviamente el primer acceso sigue tardando 5 ciclos en media, pero al aprovechar el paralelismo para los tres accesos siguientes, estos mejoran sus prestaciones.

Las memorias Burst Extended Data Out (BEDO DRAM) dan un paso más y devuelven directamente los cuatro valores que vienen determinados por la dirección de fila y columna especificadas, ese valor en concreto y los tres siguientes (figura 2.12).

2.4.2.4. SDRAM o DRAM SÍNCRONA

Todas las tecnologías anteriores son asíncronas, es decir, utilizan sus propias señales de sincronización para realizar los accesos a memoria. Pero la utilización de estas señales introduce bastante latencia y retardos innecesarios.

Por este motivo la siguiente mejora que se introdujo en las memorias fue convertirlas en síncronas, utilizando para ello el reloj global del sistema (figura 2.13). Al no utilizar señales propias de sincronización, las memorias pueden funcionar en modo 5-1-1-1. A partir de este tipo de memoria, todas las tecnologías han sido síncronas.

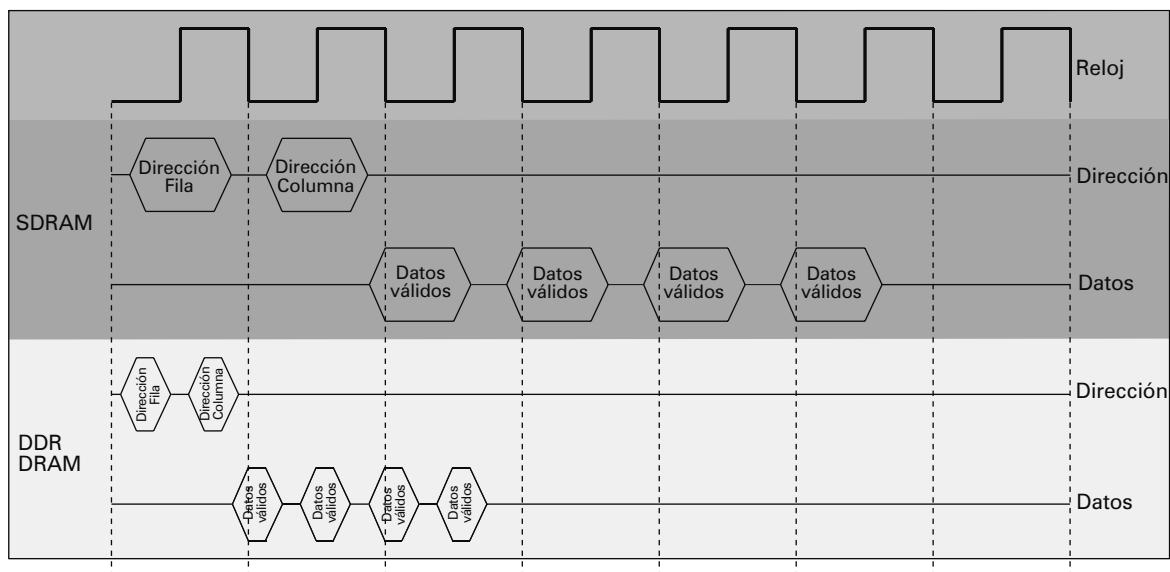


FIGURA 2.13

Cronogramas de acceso a memoria de las diferentes tecnologías síncronas de memoria principal.

2.4.2.5. DDR, DDR2 Y DDR3

La memoria DDR transfiere información dos veces en cada ciclo de reloj, una vez en el flanco de subida y otra vez en el flanco de bajada (figura 2.13).

Con esta técnica se puede llegar por primera vez a frecuencias de memoria principal de 400 MHz. Cuando fue necesario funcionar a frecuencias todavía mayores surgió la tecnología DDR2, que introdujo las siguientes mejoras:

- Las señales son diferenciales (esta técnica de señalización se estudiará en el capítulo 4), lo que permite trabajar a mayores frecuencias sin tener problemas con el ruido o las interferencias.
- Para trabajar a estas frecuencias tan altas y no tener problemas con el calor, se reduce la tensión de alimentación, de 2.5 V de las DDR a 1.8 V.

Y la tecnología DDR3 reduce de nuevo la tensión de alimentación para seguir aumentando la frecuencia de trabajo de la memoria hasta 1900 MHz. Pero tanto DDR2 como DDR3 siguen manteniendo las dos transferencias de datos por ciclo.

2.4.2.6. RDRAM y XDR DRAM

La tecnología RDRAM consigue un rendimiento bastante alto en lo que se refiere al ancho de banda, pero su elevado coste y consumo de potencia, sus elevadas latencias, así como la falta de apoyo de los fabricantes de chipsets y placas base impidió inicialmente que se extendiera en las arquitecturas de consumo.

Todas las tecnologías de memoria DRAM desarrolladas hasta el momento utilizan canales de memoria del mismo ancho que el bus de memoria y el bus del sistema (64 bits).

Pero esta tecnología propone aproximarse más a la comunicación serie, con un canal de memoria más estrecho pero más rápido. Este canal se denomina Rambus, tiene una anchura de 16 bits y funciona a frecuencias de hasta 533 MHz, permitiendo dos transferencias por ciclo de reloj. Este canal utiliza un protocolo de transferencia basado en paquetes y permite la multiplexación en el tiempo para la transferencia de direcciones y de datos.

Al Rambus pueden conectarse hasta 32 chips de RDRAM en serie, pero todas las transferencias se realizan entre el controlador de memoria y un chip, no entre los chips. Existen diferentes topologías de RDRAM según el número de canales que soporte el controlador de memoria.

La tecnología sucesora de la RDRAM ha sido XDR DRAM (eXtreme Data Rate DRAM), mucho más extendida ya que está siendo la tecnología más utilizada en las videoconsolas. Por ejemplo la PlayStation3 incorpora este tipo de memoria.

Las principales diferencias entre la memoria XDR DRAM y su predecesora RDRAM son la gran mejora en las latencias de los chips, el incremento de la frecuencia del bus hasta 800 MHz, la posibilidad de que este bus tenga una anchura de 32 bits y la utilización de ODR (Octal Data Rate), es decir, hacer ocho transferencias por ciclo de reloj.

2.4.3. Diseño de la memoria virtual

La memoria virtual se incluye en la jerarquía de memoria, principalmente por los siguientes tres motivos:

- Permite que varios procesos compartan la memoria principal del sistema de manera eficiente, habilitando mecanismos de protección del espacio de direcciones de cada uno de los procesos. Este problema no está resuelto en los niveles de la jerarquía que están más cercanos al procesador.
- Permite que un proceso que necesita más memoria principal que la disponible en el sistema pueda ejecutarse.
- Permite independencia de las referencias con respecto a la localización de los procesos en memoria principal.

Como se ha mencionado con anterioridad, la relación entre este nivel de la jerarquía y la memoria principal es parecida a la que existe entre la memoria caché y la principal. Pero también se ha comentado ya, que por otro lado existen diferencias importantes ya que la memoria virtual no se controla exclusivamente por hardware, sino que se encarga de ello el sistema operativo. Además, la memoria virtual no sólo se incluye como un nivel más de la jerarquía de memoria en el sistema, sino que tiene una función tan importante como el almacenamiento del sistema de ficheros.

Por último, la tecnología de la memoria virtual es hoy en día el almacenamiento magnético, y la unidad de información no es el bloque, sino la página o el segmento, ambas de tamaños mucho mayores que el bloque que se maneja entre la memoria principal y la memoria caché. Estas características hacen que el emplazamiento de páginas en la memoria principal siempre sea asociativo y que la política de escritura sea siempre de post-escritura.

Por todos estos motivos la memoria virtual es mucho más lenta y compleja de gestionar que el resto de niveles de la jerarquía (actualmente su latencia está en el orden de ms y no de ns, como en el entorno más cercano al procesador). Y resolver el fallo de página de un proceso siempre implica un cambio de contexto en el procesador para evitar la enorme penalización por fallo que se observaría al resolver este fallo de página sin realizar mientras tanto trabajo útil en el procesador. Además, el sistema operativo suele ejecutar algoritmos de prebúsqueda de páginas para evitar fenómenos como el de la hiperpaginación (trashing), en los que el procesador provoca constantes fallos de página y pasa desocupado la mayor parte del tiempo.

En este libro no se desea entrar en detalles tecnológicos ni en detalles acerca del funcionamiento del sistema operativo, por lo tanto, se consideran sólo los siguientes aspectos de diseño relacionados con la memoria virtual.

2.4.3.1. ORGANIZACIÓN DE LA MEMORIA VIRTUAL

Existen dos tipos de organización para la memoria virtual, la memoria paginada y la segmentada.

La memoria virtual paginada utiliza un tamaño fijo de bloque de información, denominado página, mientras que la segmentada utiliza un tamaño variable de bloque de información; cada uno de estos bloques se denomina segmento.

Las memorias segmentadas son más flexibles, ya que permiten que cada proceso utilice los tamaños de bloque de información más adecuados para sus necesidades, pero complican bastante el esquema de direccionamiento.

Por eso algunos diseños utilizan una técnica híbrida en la que los segmentos están siempre compuestos por un número entero de páginas. Esta técnica se denomina memoria virtual paginada/segmentada.

Lo que se busca en general es encontrar la organización de memoria virtual y los tamaños de página o segmento que permitan aprovechar al máximo todo el espacio disponible en la memoria principal, evitando en todo lo posible el fenómeno de la fragmentación y llegando a un compromiso entre el aprovechamiento de la localidad y la penalización por fallo (el razonamiento en cuanto al tamaño de página o segmento más adecuado es muy similar al del tamaño de bloque para la memoria caché).

La fragmentación no es más que el desaprovechamiento del espacio en la memoria principal (figura 2.14). Si la memoria virtual es paginada, al ubicar el conjunto de datos manejado por un proceso en una o más páginas, es muy probable que en la última de ellas quede espacio desaprovechado. Es lo que se denomina fragmentación interna, ya que dentro de las diferentes páginas ubicadas en la memoria principal puede haber espacio que no esté almacenando información útil. Cuanto mayor sea el tamaño de página, menor será el tamaño de la tabla de páginas (se verá en la sección siguiente), pero mayor será el ancho de banda necesario para transferir estas páginas y mayor será el problema de la fragmentación interna.

Para evitar esto puede optarse por una memoria virtual segmentada, que permita ajustar el tamaño de los segmentos justo a los conjuntos de datos que se manejan. De hecho, mientras que la división de la memoria en páginas suele ser transparente al usuario, la división en segmentos suele ser visible para él, de manera que pueda organizar óptimamente el espacio de memoria de cada proceso y asignar atributos de protección y privilegios a las instrucciones y los datos según sus necesidades. Pero al ubicar los diferentes segmentos en la memoria principal, quedarán espacios entre ellos que no se puedan aprovechar por no encontrarse otros segmentos de tamaño adecuado para ubicarlos en ellos. Esto se denomina fragmentación externa, ya que quedan espacios en la memoria principal desaprovechados porque no albergan ningún segmento. Los espacios libres de memoria suelen gestionarse mediante una o varias listas enlazadas y los algoritmos de emplazamiento más utilizados son los de Primer ajuste, Mejor ajuste, Peor ajuste y Binary-buddy, pero ninguno de ellos consigue evitar la fragmentación externa completamente.

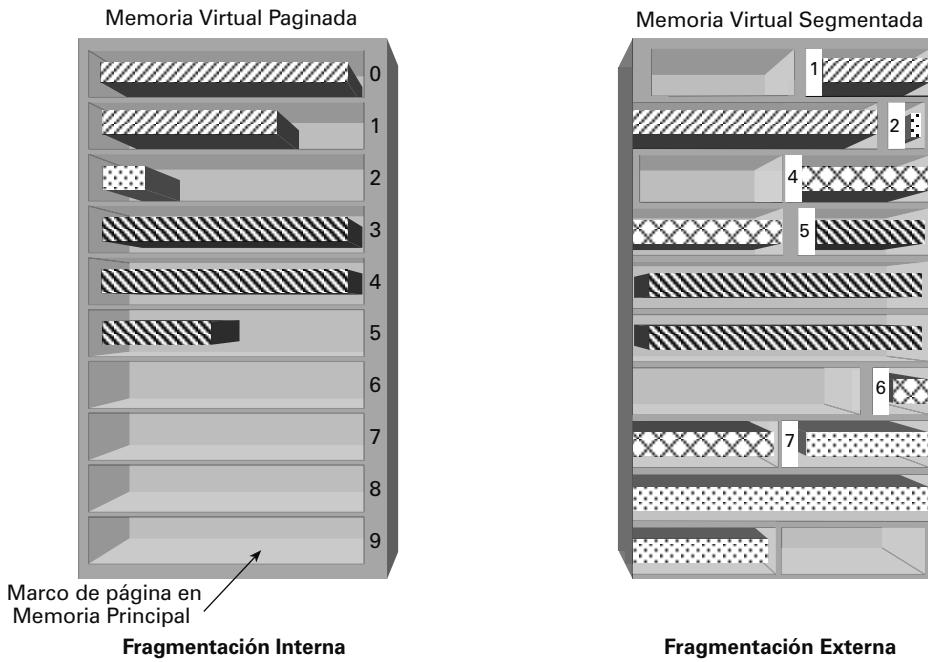


FIGURA 2.14

Fragmentación interna y externa.

2.4.3.2. TRADUCCIÓN DE DIRECCIONES VIRTUALES A FÍSICAS

La correspondencia entre dirección virtual y física que se ha mostrado a lo largo de este capítulo, y que en general se utilizará en todo este libro, es la que se produce cuando la memoria virtual es paginada, y por lo tanto, el tamaño del bloque de información que se maneja entre la memoria virtual y la principal es fijo.

En este caso, existen tres tipos de traducción de direcciones:

- **Traducción directa.** Cada proceso tiene un tabla de páginas que almacena en qué marco de la memoria principal se ha almacenado cada página de su espacio de direcciones, además de otra información como si la página ha sido utilizada, si ha sido modificada, etc. La tabla de páginas se almacena en memoria principal, apuntada por un registro que contiene su dirección base (figura 2.15).
- **Traducción asociativa.** Sólo se mantiene en la tabla de páginas una entrada para las páginas que en ese momento están ubicadas en la memoria principal (figura 2.16).
- **Traducción mixta.** Se mantiene una pequeña tabla asociativa con las páginas más activas, denominada TLB (Translation Lookahead Buffer) en una memoria caché cercana al procesador. Y se almacena la tabla de páginas completa en memoria principal para los fallos del TLB (figura 2.17).

La traducción directa de direcciones es sencilla pero implica siempre un acceso a memoria principal cuando se desea leer o escribir un dato, para leer la tabla de páginas. Además, la tabla de páginas de un proceso puede ocupar un tamaño excesivo en la memoria principal. Por otro lado, la traducción asociativa reduce mucho el tamaño de la tabla de páginas, pero obliga a comparar la página buscada con todas sus entradas para ver si se encuentra en la memoria principal, por lo que el proceso de traducción puede ser más lento.

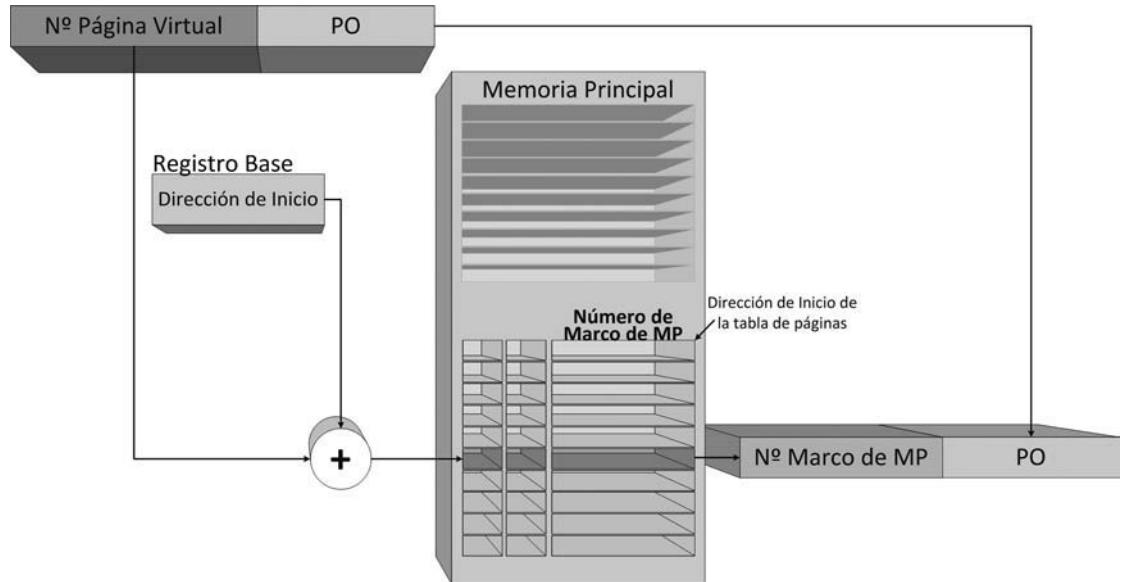


FIGURA 2.15

Traducción directa de DV a DF en memoria paginada.

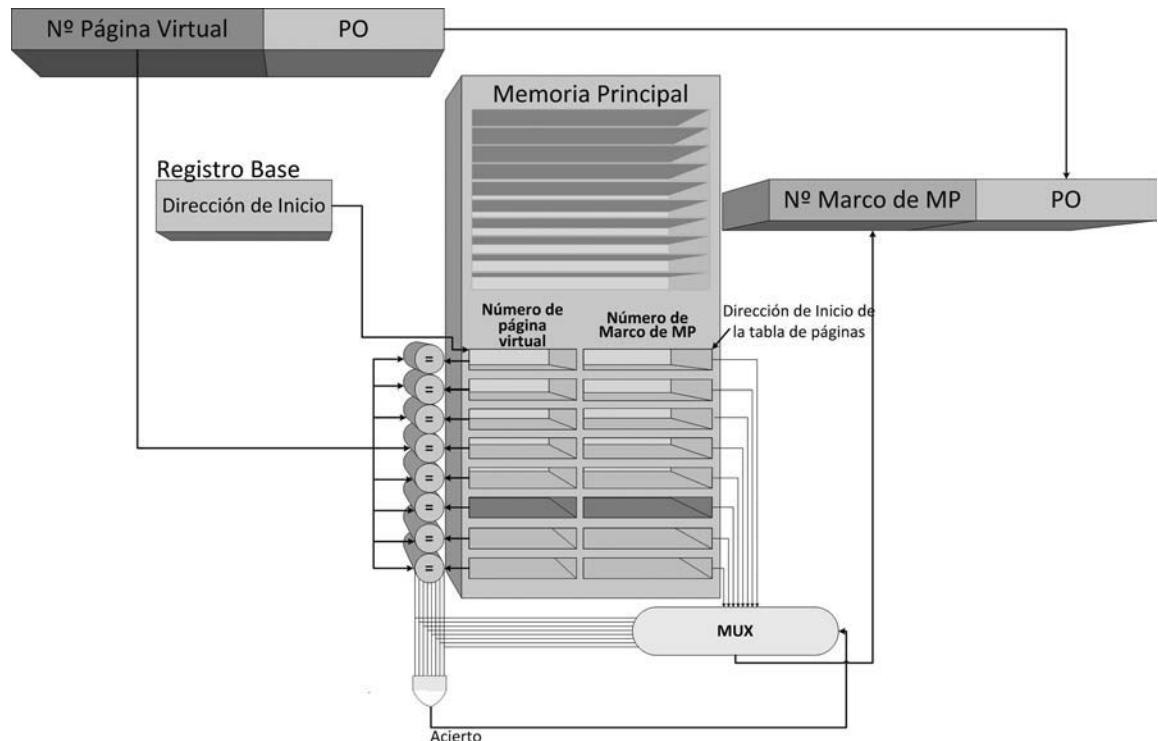


FIGURA 2.16

Traducción asociativa de DV a DF en memoria paginada.

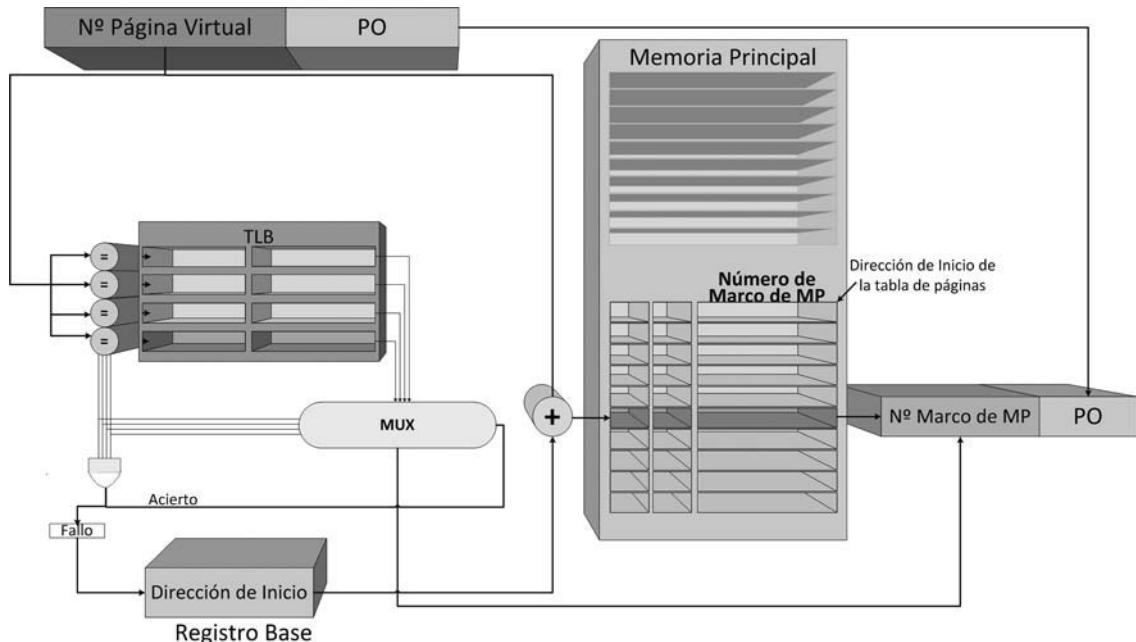


FIGURA 2.17

Traducción mixta de DV a DF en memoria paginada.

La traducción mixta reduce el número de comparaciones y las realiza mucho más rápido, ya que el TLB sólo almacena las páginas más activas y es una memoria caché. Pero sigue haciendo falta almacenar la tabla de páginas en la memoria principal. La solución suele ser la traducción a varios niveles, de manera que se divide la tabla de páginas en páginas, y no tienen por qué estar todas en la memoria principal al mismo tiempo. Lo que se almacena siempre en esta memoria es una tabla de directorios que alberga las direcciones de comienzo de las páginas de la tabla de páginas. La dirección virtual se interpreta ahora de otra manera, como se observa en la figura 2.18.

Esta decisión de diseño afecta enormemente al rendimiento de los accesos a memoria, por lo que volverá a tratarse este tema en el capítulo 4.

Cuando la memoria virtual es segmentada la interpretación de la dirección virtual y física es diferente que en una memoria paginada, ya que la memoria principal no está particionada a priori y un segmento puede comenzar y terminar en cualquier posición.

En este caso sólo hay una alternativa, la traducción directa. Cada proceso tiene asociada una tabla de segmentos en memoria principal, en la que se almacenan la dirección de comienzo y la longitud de cada segmento en memoria principal. La tabla de segmentos está apuntada por su dirección base, que se almacena en un registro (figura 2.19). En este caso el desplazamiento que se incluye tanto en la dirección virtual como en la física suele denominarse *Offset de Segmento* (SO) en lugar de *Offset de Página* (PO).

2.5 Diseño de un sistema de E/S básico

En cualquier computadora es imprescindible un sistema de E/S que permita el intercambio de información con el mundo exterior, y sin este sistema la computadora serviría de poco.

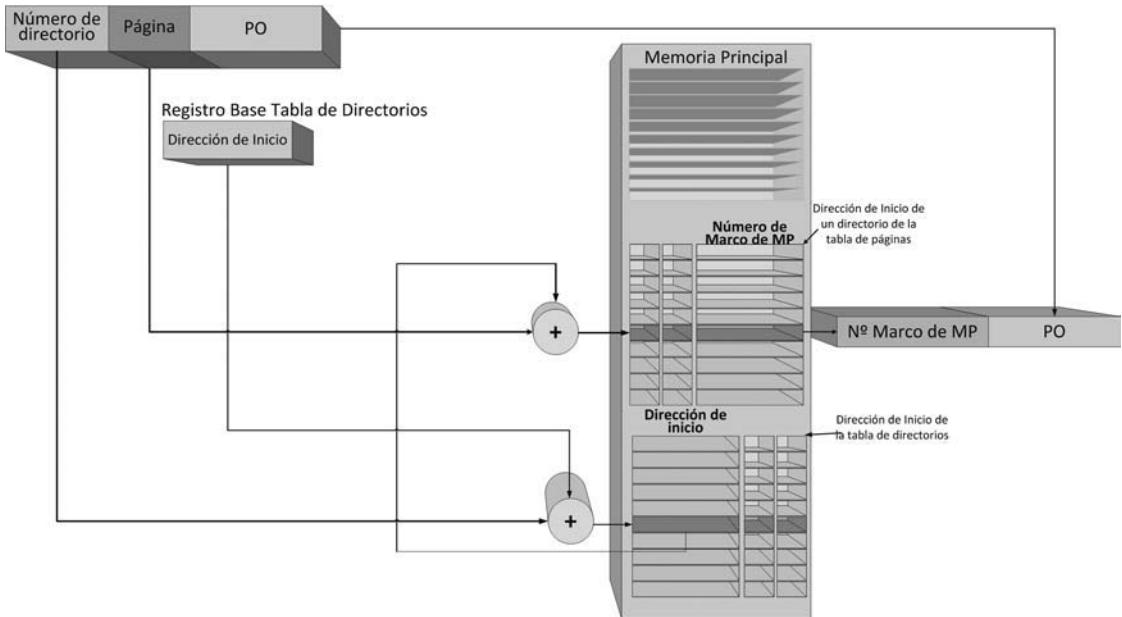


FIGURA 2.18

Traducción de DV a DF a varios niveles en memoria paginada.

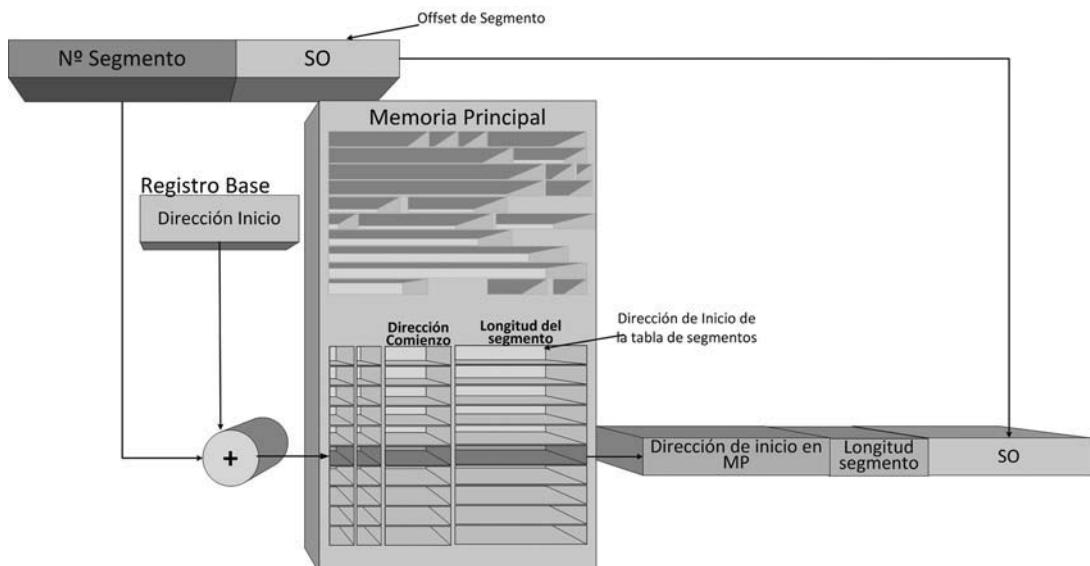


FIGURA 2.19

Traducción directa de DV a DF en memoria segmentada.

El sistema de E/S está formado por una serie de dispositivos periféricos que permiten la transferencia de información entre la computadora y el exterior, y por otros elementos imprescindibles para su conexión y control.

Este sistema ha sido siempre el gran olvidado en el campo de la arquitectura de computadoras. Sin embargo, este hecho ha terminado por afectar al rendimiento de las arquitecturas, ya que, por mucho que se optimicen el funcionamiento del procesador y de la jerarquía de memoria, el sistema de E/S puede llegar a suponer un gran cuello de botella que afecte negativamente a sus prestaciones.

Para obtener el rendimiento deseado del sistema de E/S hay que tener en cuenta, no sólo el funcionamiento del propio dispositivo de E/S, sino también la forma en la que este dispositivo se conecta a la computadora y la forma en la que se gestionan las transferencias de información.

Por eso en este libro se dividirá el estudio del sistema de E/S en tres grandes aspectos: buses incluidos en este sistema, dispositivos de E/S y gestión del sistema de E/S. Este capítulo se centra en el primero y en el tercero, ya que escapa del alcance de este libro el estudio del funcionamiento básico de los dispositivos de E/S conectados a la computadora.

2.6 Mecanismo completo de una operación de E/S

En la figura 2.20 puede observarse el camino completo que debe seguir el procesador para hacer una operación de entrada o salida con un dispositivo periférico. En este esquema de la operación de E/S sólo se muestran los aspectos hardware, ya que tanto el sistema operativo como los drivers de los dispositivos tienen también un papel importante en las transferencias de información.

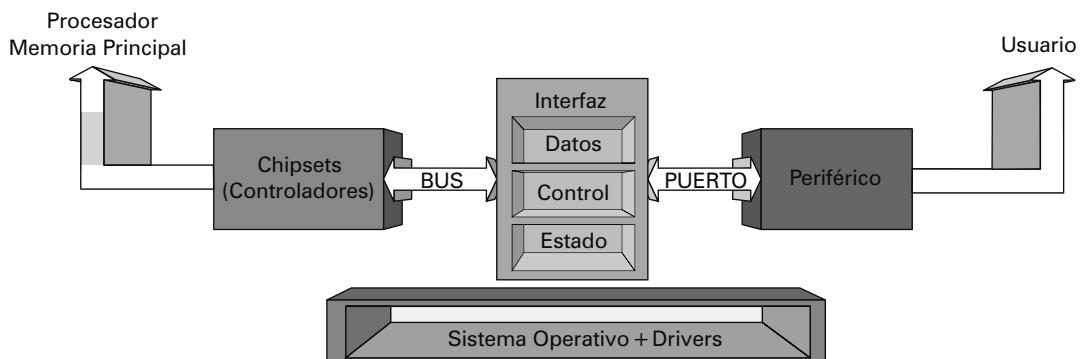


FIGURA 2.20

Mecanismo completo de E/S.

Existe una gran variedad de dispositivos periféricos que se pueden conectar a una computadora hoy en día: de E/S básica (teclado, ratón, monitor), de almacenamiento (disco duro, CD, DVD), de impresión (impresora, escáner), de comunicaciones (modem, acceso a red LAN), multimedia (micrófono, altavoces, capturadora, cámara de video), de automatización y control (sensores, alarmas, adquisición de datos), etc.

Sea cual sea el tipo de dispositivo, todos se conectan al sistema a través de un interfaz cuyas funciones principales son:

- Interpretar las órdenes que recibe del procesador y transmitirlas al periférico.
- Controlar las transferencias de datos entre el procesador y el periférico (convertir formatos, adaptar velocidades).
- Informar al procesador del estado del periférico.

Este interfaz es una unidad hardware/software (la parte software es la que está ligada a los drivers y al sistema operativo) que suele estar ubicada en una tarjeta o adaptador. Los interfaces suelen clasificarse como serie o paralelo según cómo manejen la información o en interfaces generales o específicos según estén diseñados para dar soporte a un gran conjunto de dispositivos o a un tipo de dispositivo concreto. La unidad física que permite que el dispositivo se conecte a su interfaz suele denominarse puerto.

El interfaz se conecta mediante un bus de E/S con un controlador que suele encontrarse en un chipset o hub y que gestiona la transferencia de información con el procesador o con la memoria según los mecanismos de gestión de E/S disponibles en la arquitectura.

Con esta estructura, el sistema de E/S debe ser capaz de cumplir tres funciones que son imprescindibles para la realización de una operación de entrada o salida:

- **Direccionamiento.** El procesador debe ser capaz de seleccionar de alguna manera el dispositivo con el que quiere realizar la transferencia de E/S o de identificar al dispositivo que le hace algún tipo de solicitud. Existen dos alternativas:
 - **E/S mapeada en memoria.** El sistema de E/S y la memoria principal comparten el mismo espacio de direcciones de manera que las operaciones de E/S se realizan con instrucciones de load y store. Se asigna a los puertos de E/S una porción del espacio de direcciones físicas de memoria.
 - **E/S aislada.** El sistema de E/S y la memoria utilizan un espacio de direcciones diferente. Existen instrucciones específicas para E/S (tipo in y out). Hay una línea en el bus de direcciones que especifica si se realiza un direccionamiento de memoria o de E/S ya que puede ocurrir que un puerto tenga la misma dirección que una posición válida de memoria.

La E/S mapeada en memoria reduce el espacio de memoria principal que se puede emplear para almacenar instrucciones y datos. Pero como las direcciones de los puertos se manejan igual que las de memoria, se pueden realizar todo tipo de operaciones sobre ellas (aritméticas, de manipulación de bits, etc). Y además se simplifica el repertorio de instrucciones del procesador. Hoy en día todavía se pueden encontrar las dos alternativas, ya que no se ha comprobado que una se comporte de manera sustancialmente mejor que la otra.

- **Transferencia de información.** Que podrá ser de dos tipos, lectura o escritura.
- **Gestión de la transferencia.** En el sistema de E/S es necesario un mecanismo de sincronización y control de las transferencias de información porque el procesador debe saber si el periférico está preparado para enviar o recibir datos. También debe estar informado, por ejemplo, de si el periférico ha terminado una transferencia y está en disposición de iniciar una nueva. En secciones posteriores se estudiarán las alternativas que existen para realizar esta gestión de E/S.

2.7 Evaluación de prestaciones del sistema de E/S

En el caso de los buses de E/S la ecuación de prestaciones está relacionada con el ancho de banda del bus (BW de BandWidth), es decir, con la cantidad de información que es capaz de transferir por unidad de tiempo:

$$BW = \text{ancho de datos} \cdot f \cdot n.^{\circ} \text{ de transferencias / ciclo}$$

Donde el *ancho de datos* es el número de líneas de datos del bus, *f* es la frecuencia de funcionamiento de estas líneas y *n.^º de transferencias/ciclo* indica cuántas transferencias de información es capaz de realizar el bus por cada ciclo de reloj.

Este ancho de banda suele verse limitado por:

- El ancho de datos, ya que aumentarlo implica aumentar el número de líneas de datos del bus y esto ocupa área y consume potencia. Es decir, existe una limitación de coste asociada a este factor. Además, cuantas más líneas de datos se coloquen en paralelo en un bus, más difíciles de gestionar serán todos los problemas relacionados con el ruido provocado por las interferencias que se producen entre ellas.
- En cuanto a la frecuencia, las limitaciones vuelven a ser el consumo de potencia y el ruido, ya que cuanto mayor sea la frecuencia de operación del bus, más potencia se disipará en sus líneas y más interferencias se producirán entre ellas.
- Por último, el número de transferencias por ciclo suele estar limitado por el modo de operación del bus, es decir, por los protocolos de transferencia, sincronización y arbitraje que utilice.

De hecho, aunque la ventaja principal de utilizar buses estándar para conectar dispositivos de E/S es su bajo coste, ya que permiten la compartición de líneas, y su versatilidad (facilidad de conexión de nuevos dispositivos, creación de estándares cuando se necesitan, etc); la principal desventaja está en estas limitaciones del ancho de banda que pueden llegar a convertir a los buses en el cuello de botella del sistema de E/S.

En el capítulo 4 de este libro se presentan técnicas de aumento de prestaciones que permiten aumentar el ancho de banda de los buses sin sufrir los inconvenientes ya mencionados.

En cuanto a la evaluación de prestaciones de los dispositivos, las métricas más habituales son las mismas que se utilizan para cuantificar el rendimiento de un nivel de la jerarquía de memoria, el ancho de banda del dispositivo (que en este caso es la cantidad de información que puede transferirse desde/hacia el dispositivo de E/S por unidad de tiempo) y la latencia (tiempo que tarda en completarse una transacción de E/S). Además existen otras métricas como la fiabilidad, la disponibilidad o la tolerancia a fallos que pueden ser muy importantes en algunos casos. Y cada tipo de dispositivo puede tener sus propias métricas de rendimiento dependiendo de su aplicación.

Por último, para evaluar el rendimiento de un mecanismo de gestión de E/S se suele utilizar como métrica el tiempo invertido por el procesador para realizar la transacción de E/S completa ($t_{E/S}$).

2.8 Buses de E/S

Un bus es un medio de transmisión compartido que interconecta dos o más dispositivos y permite que se establezca entre ellos una correcta comunicación.

Normalmente sólo un dispositivo puede controlar el estado de las líneas del bus en cada momento. Si varios dispositivos transmiten información por el mismo bus al mismo tiempo, las señales se solapan y se produce un error de contención. Por lo tanto en cada comunicación hay siempre dos elementos implicados, el maestro, que inicia y dirige la transferencia y el esclavo, que obedece las órdenes del maestro.

Un bus se compone de diferentes conductores eléctricos denominados líneas. Se pueden encontrar diferentes tipos de líneas en un bus:

- **Según su función:** Líneas de datos, líneas de direcciones o líneas de control.
- **Según su uso:** Líneas dedicadas o líneas multiplexadas.
- **Según sus características eléctricas:** Líneas unidireccionales o bidireccionales.

Los buses de E/S suelen estar estandarizados para que los distintos dispositivos de diferentes fabricantes puedan conectarse correctamente en todas las arquitecturas. Los estándares especifican una serie

de requisitos eléctricos, mecánicos y de protocolos de comunicación que deben cumplir los buses y las líneas que los componen.

Los parámetros de caracterización más importantes de un bus se suelen separar en diferentes niveles para definir estos protocolos:

- **Nivel físico.** Capacidad de conexión (número máximo de dispositivos que pueden conectarse al bus), longitud máxima del bus, soporte físico, niveles de tensión, frecuencia de funcionamiento, tipo de conectores, etc.
- **Líneas.** Ancho del bus (número total de líneas), ancho de datos, líneas unidireccionales o bidireccionales, etc.
- **Modo de operación.** Protocolos de transferencia (que determinen el tipo de comunicación que se establece entre los dispositivos conectados al bus), protocolos de sincronización (que determinen el inicio y el fin de cada transferencia de información) y de arbitraje (que controlen el acceso al bus cuando más de un dispositivo puede actuar como maestro).

2.8.1. Diseño de buses de E/S

Una vez realizadas las especificaciones físicas y de líneas de un bus (parámetros más relacionados con la tecnología), es necesario diseñar el protocolo de transferencia que gobierna la operación del bus y especifica cómo se utilizan las señales de datos, control y direcciones para realizar una transferencia de información completa. Existen casi tantos protocolos de transferencia como buses de E/S.

Además es necesario decidir el tipo de protocolo que sincroniza las transferencias de información y en este caso sí que existe un conjunto muy reducido de alternativas. En los buses síncronos las transferencias están gobernadas por una única señal de reloj compartida por todos los dispositivos que se conectan al bus, de manera que cada transferencia se realiza en un número fijo de ciclos de reloj (en un ciclo en el ejemplo de la figura 2.21).

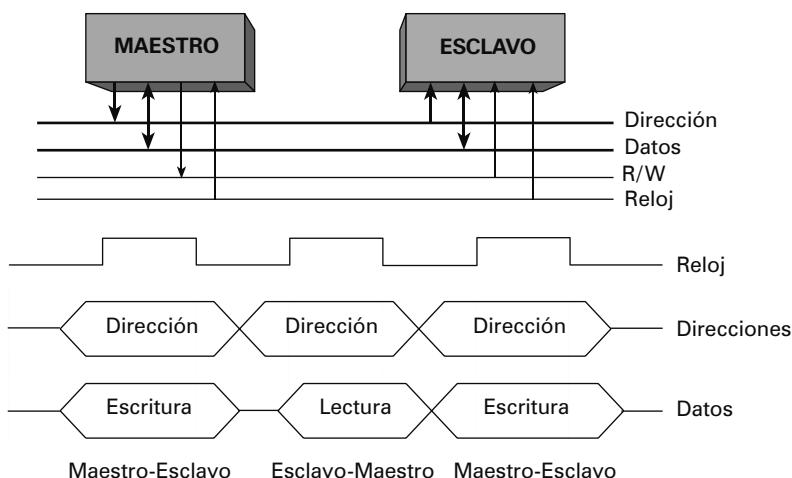


FIGURA 2.21

Bus con protocolo síncrono.

Por el contrario en los buses asíncronos no existe señal de reloj, los dispositivos implicados en la transferencia se sincronizan mediante el intercambio de señales de control (figura 2.22).

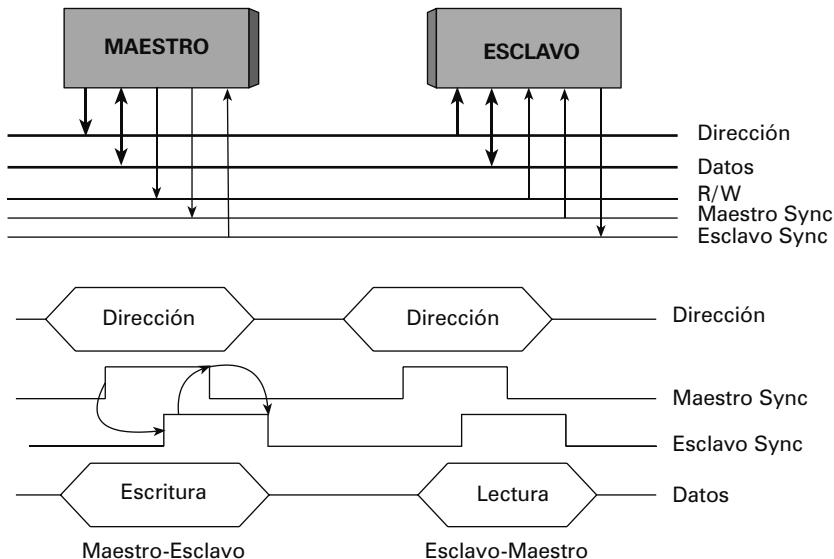


FIGURA 2.22

Bus con protocolo asíncrono.

Los protocolos síncronos son muy sencillos y sólo necesitan una señal de reloj. Pero hay que adaptar esta señal al dispositivo más lento y además es necesario distribuirla a todos los dispositivos conectados al bus. Tampoco existen confirmaciones en las transferencias, por lo que es difícil implementar mecanismos de detección y corrección de errores. Con los protocolos asíncronos no existen estos inconvenientes, pero son menos eficientes debido a la necesidad de intercambiar señales de control.

Por eso surgen los buses semisíncronos, que combinan las ventajas de los dos anteriores, se comportan como síncronos para dispositivos rápidos y como asíncronos para dispositivos lentos. Es decir, en principio se exige a todos los dispositivos conectados al bus que completen sus transferencias en un número fijo de ciclos de reloj, pero si algún dispositivo no puede hacerlo, es posible que funcione como asíncrono activando para ello las señales adecuadas.

También existen los buses de ciclo partido, en los que se libera el bus entre la petición de operación por parte del maestro y la contestación del esclavo. El problema en este caso es que todos los dispositivos tienen que ser capaces en un momento dado de actuar como maestros tomando el control del bus cuando van a proceder a dar una contestación.

En general los protocolos semisíncronos y de ciclo partido presentan importantes ventajas en comparación con los síncronos y asíncronos, pero suelen necesitar más líneas y una gestión más compleja.

Una vez determinado el protocolo de transferencia y el de sincronización, el resto de decisiones de diseño suelen estar relacionadas con el arbitraje del bus, es decir, con el protocolo que permite determinar qué dispositivo tiene el control del bus en cada momento.

Los protocolos de arbitraje pueden ser de dos tipos, centralizados o distribuidos. Sea cual sea la elección, el objetivo es evitar los errores de contención que se producen cuando dos o más dispositivos toman el control del bus al mismo tiempo, pero también evitar los interbloqueos.

En el caso del arbitraje centralizado, el control del acceso al bus es responsabilidad de un único dispositivo. Existen distintas maneras de conectar el resto de dispositivos al maestro que actúa como árbitro, pero las conexiones más habituales son en estrella y encadenadas con 2, 3 o 4 líneas (en la figura 2.23 se muestran distintos ejemplos, en todos ellos Req es la señal de solicitud del bus y Gnt es la señal de concesión).

En los protocolos distribuidos la responsabilidad se comparte entre todos los dispositivos conectados al bus, normalmente utilizando líneas o códigos de identificación (figura 2.24).

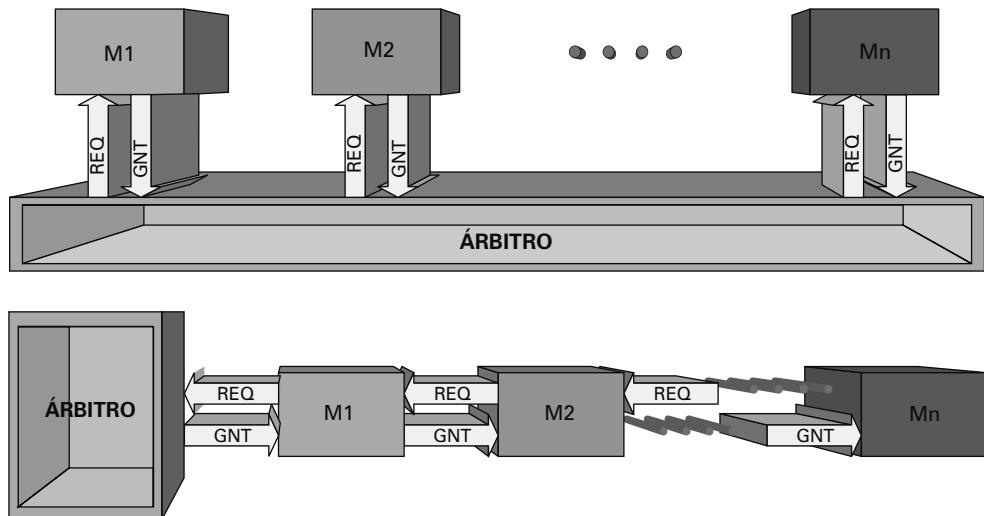


FIGURA 2.23

Protocolos de arbitraje centralizados (en estrella y encadenado de 2 líneas).

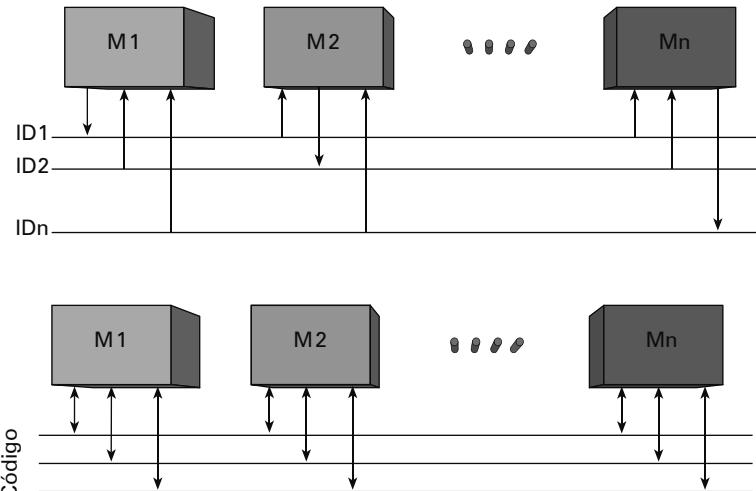


FIGURA 2.24

Protocolos de arbitraje distribuidos (líneas de identificación y códigos de identificación).

Ejemplo 2.15

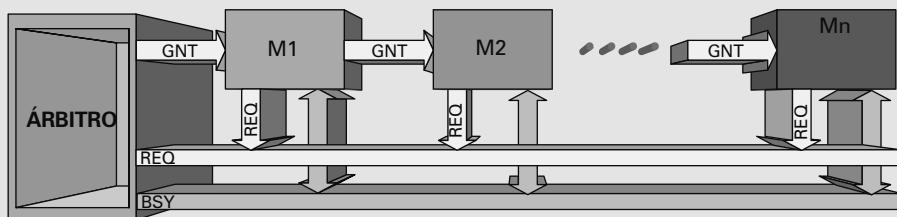
Diseño de un protocolo de arbitraje centralizado con conexión encadenada de dispositivos.

En la figura 2.23 se ha mostrado un ejemplo de protocolo centralizado de arbitraje de 2 líneas con conexión encadenada de dispositivos.

El dispositivo maestro que quiere el control del bus activa la señal de petición REQ. El resto de maestros propagan esta señal hasta el árbitro y éste activa la señal de concesión GNT. Si un maestro recibe GNT y no ha pedido el bus, la propaga al siguiente maestro. Si por el contrario un maestro recibe GNT y tenía una petición de bus pendiente, toma el control del bus. Por lo tanto las prioridades se establecen por el orden de conexión de los maestros.

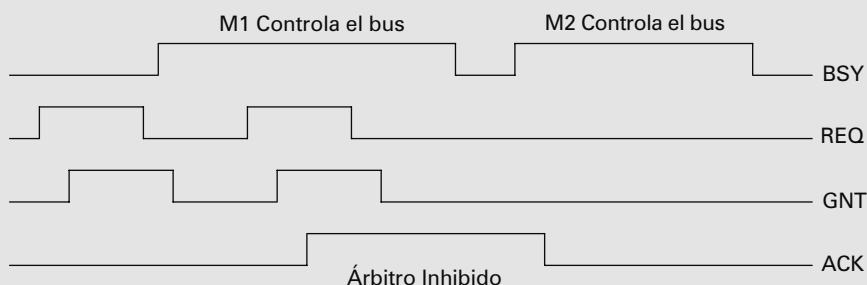
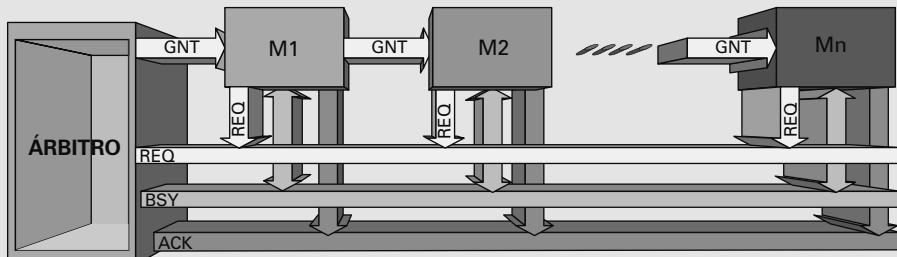
Si un maestro solicita el bus y le es concedido, puede ocurrir que otro maestro que esté más próximo al árbitro en la cadena solicite el bus y al ver GNT activada crea que puede tomar el control del bus. Para evitar estos errores en el arbitraje (que provocarían errores de contención), la señal de GNT debe funcionar por flanco y no por nivel; un maestro sólo toma el control del bus cuando detecta el flanco de subida de la señal de concesión.

Aún así, los errores de contención son posibles, y para hacer el protocolo de arbitraje más robusto, se puede añadir una tercera línea BSY que señale si el bus está ocupado. La línea BSY se activa siempre que un maestro tiene el control del bus.



De esta forma, si un dispositivo maestro solicita el control del bus activando REQ, el árbitro activa GNT cuando ve esta petición pendiente y además el bus no está ocupado (BSY sin activar). La propagación de GNT es como en el protocolo de 2 hilos, sólo que en este caso un maestro sólo toma el control del bus si tiene una petición pendiente, la línea BSY está inactiva y si detecta el flanco de subida de GNT.

Este protocolo se puede mejorar añadiendo una cuarta línea que permita solapar la transferencia del ciclo actual con el arbitraje del ciclo siguiente. Esto se hace con la señal ACK. Si un maestro solicita el bus y recibe GNT pero el bus está ocupado, activa ACK como señal de espera a que el bus quede libre para tomar el control. Mientras ACK está activa, el árbitro queda inhibido y no puede activar GNT para ningún otro dispositivo.



2.8.2. Diseño de jerarquías de buses

Si en un computador actual conectáramos todos los dispositivos a un mismo bus, tendríamos dos problemas serios. El primero, una disminución global en el rendimiento del sistema por los retardos de propagación de las señales debidos a la longitud del bus, a los tiempos de espera para conseguir la utilización del bus y a la adaptación necesaria para dar servicio a los dispositivos más lentos. Y el segundo, la incompatibilidad de los distintos dispositivos con el bus, ya que cada fabricante diseña sus propios interfaces optimizados para la función que tengan que cumplir.

La solución a todos estos problemas es utilizar una jerarquía de buses en lugar de un único bus. Para diseñar esta jerarquía hay que tener en cuenta la ubicación de los diferentes niveles de la jerarquía de memoria, la ubicación de los controladores e interfaces de E/S, la necesidad de algunos dispositivos de acceder directamente a la memoria principal, y obviamente, las diferentes necesidades en cuanto a ancho de banda de los dispositivos.

Normalmente todas las jerarquías incorporan los siguientes buses:

- **Bus del sistema y bus de memoria.** Son los que conectan el procesador con el resto del sistema y la memoria principal con el controlador de memoria respectivamente. Se trata de buses rápidos y cortos, propietarios (no estandarizados) y optimizados para arquitecturas y diseños específicos. Esta optimización es posible ya que a estos buses se conectan un número fijo de dispositivos de prestaciones conocidas.
- **Buses de expansión.** Se trata de buses más largos y lentos, abiertos (estandarizados), accesibles por el usuario y a los que se conectan un número indeterminado de dispositivos de prestaciones desconocidas y muy diferentes entre sí.

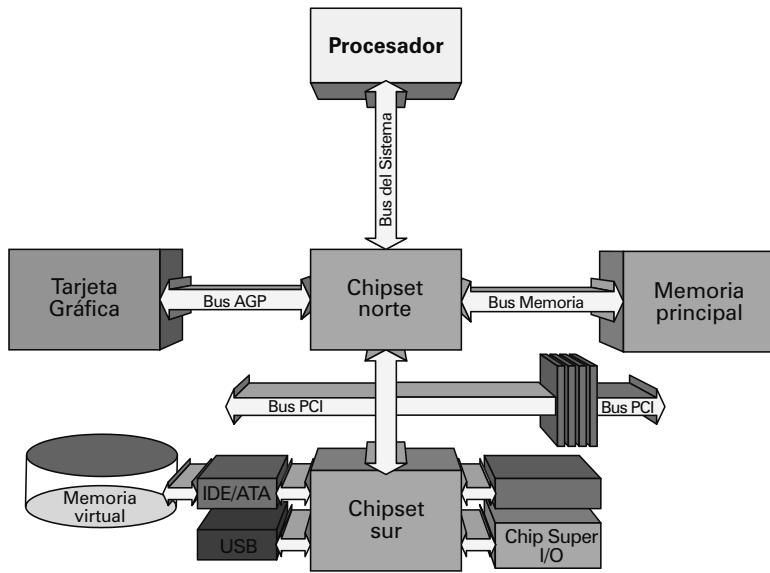
Las ventajas de este tipo de jerarquía son claras. El bus del sistema permite que la conexión entre el procesador y el mundo exterior esté optimizada para su arquitectura y necesidades específicas. Además, los distintos componentes del sistema pueden comunicarse entre sí sin interferirse unos con otros. No es necesario adaptarse al dispositivo más lento, de hecho se distinguen en la jerarquía diferentes zonas según su ancho de banda. Y por último, se elimina el problema de incompatibilidad con los buses ya que los buses de expansión suelen ser estándares.

Aunque se comenzó por utilizar un único bus de expansión en las jerarquías, con el paso del tiempo ha sido necesario mejorar todavía más la eficiencia del sistema, utilizando para ello una jerarquía con más de un bus de expansión. Así se separan los dispositivos periféricos según sus necesidades y, por norma general, un dispositivo rápido tiene la misma probabilidad de acceder al bus del sistema que todos los dispositivos lentos conjuntamente.

CASO PRÁCTICO 2.1. Evolución de la jerarquía de buses de las computadoras personales

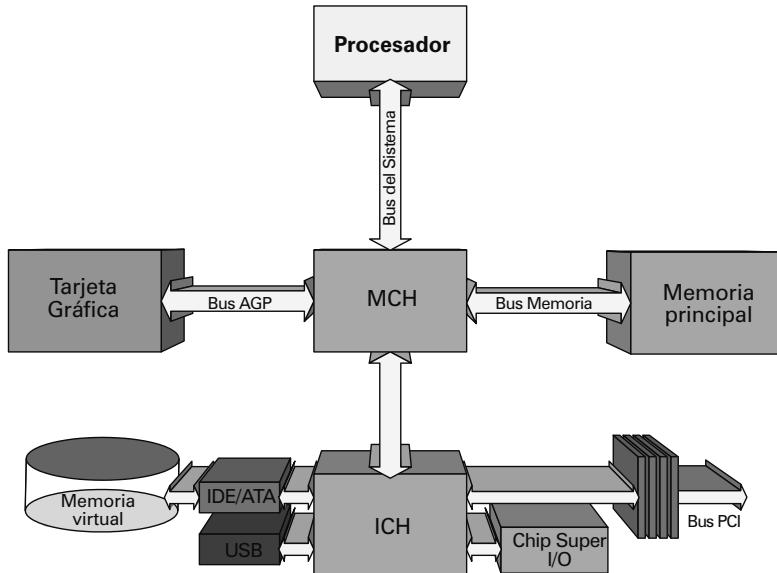
Hace unos años, la jerarquía de buses que se incluía en las placas base de las computadoras personales era la que se muestra en la primera figura de la página siguiente.

Se observa que el procesador se comunica con el controlador de memoria, integrado en el chipset norte, mediante el bus del sistema. Al chipset norte se conecta la memoria principal (mediante el bus de memoria) y también la tarjeta gráfica. La conexión de esta tarjeta se realiza mediante alguna de las versiones de AGP, un bus de expansión estándar que se utiliza sólo para la conexión de tarjetas gráficas. Este dispositivo de E/S se trata de manera diferente a los demás debido a la exigencia de altas prestaciones que se le impone y a la necesidad que tiene de acceder a la memoria principal.



El resto de dispositivos de E/S se conectan al sistema a través del bus PCI (bus de expansión tradicional) o de otros buses e interfaces más específicos, como puede ser el IDE/ATA para el disco duro. La conexión entre el chipset sur, que centraliza todos los flujos de información de E/S, y el chipset norte, se realiza mediante el bus PCI.

Este diseño de jerarquía comenzó a presentar cuellos de botella, especialmente en lo que se refiere a la conexión entre chipsets (hay que pensar que, por ejemplo, la tarjeta de red suele estar conectada en una de las ranuras PCI), por lo que se pasó a realizar la siguiente mejora:

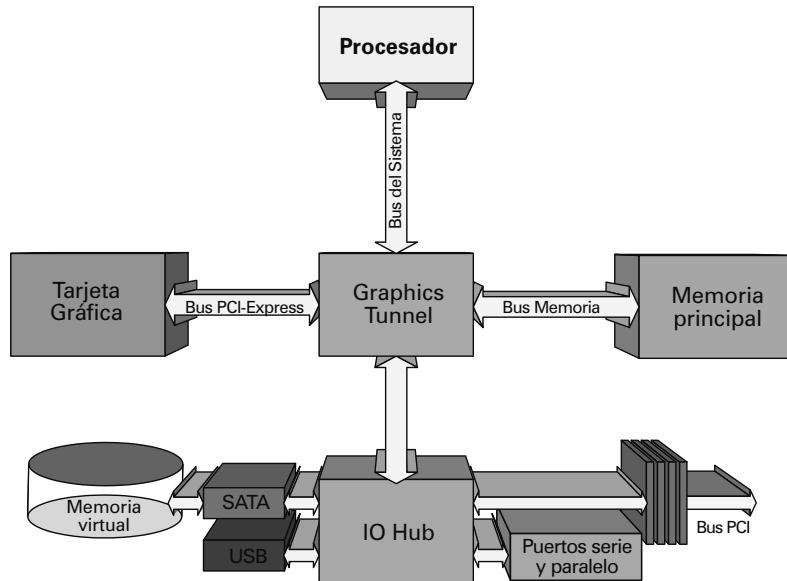


En este caso el bus PCI se conecta directamente al nuevo chipset sur, denominado ICH (IO Controller Hub). Y la conexión entre el ICH y el MCH (Memory Controller Hub que sustituye al antiguo chipset norte) se realiza mediante un bus dedicado de ancho de banda bastante mayor que PCI.

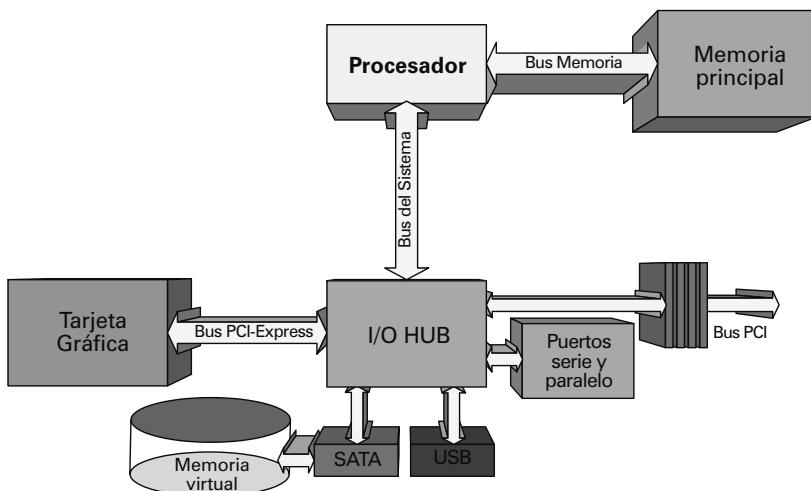
De esta manera, no se interfiere con los dispositivos conectados al bus PCI, que mejoran su rendimiento. Y también mejoran su rendimiento los dispositivos que se conectan directamente al ICH y no tienen que pasar por el bus PCI. Esto se nota especialmente en las prestaciones de los dispositivos USB e IDE/ATA.

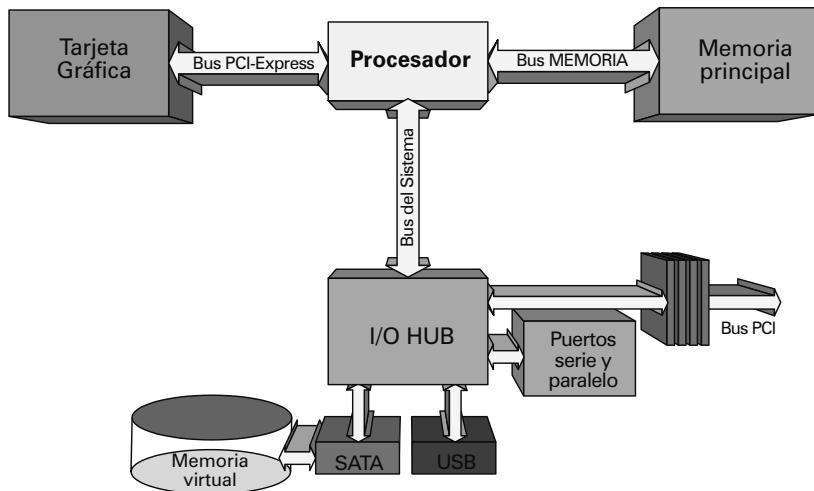
Este diseño ha terminado por modificarse de nuevo, ya que al integrar los últimos procesadores el controlador de memoria, la conexión de la memoria principal se hace directamente al procesador en lugar de al MCH. Además, el bus estándar para la conexión de tarjetas gráficas se ha modificado, siendo en los diseños más actuales PCI Express (que estudiaremos en el capítulo 4 de este libro) en lugar de AGP. Actualmente nos encontramos tres alternativas en el diseño de la jerarquía de buses del PC:

Alternativa 1



Alternativa 2



Alternativa 3

2.9 Gestión del sistema de E/S

Como se ha comentado con anterioridad, es necesario implementar un mecanismo de gestión de E/S que controle las transferencias de información entre el procesador y el dispositivo periférico. Estos mecanismos de gestión son muy variados, y es mejor utilizar unos u otros dependiendo del tipo de dispositivo que se conecte a la computadora.

2.9.1. E/S programada con espera de respuesta

Cada vez que el procesador tiene que realizar una transferencia de E/S, entra en un bucle en el que consulta una y otra vez el estado del periférico hasta que éste está preparado para realizar la transferencia (figura 2.25).

Los inconvenientes de este método son:

- El procesador no hace trabajo útil durante el bucle de espera.
- La dinámica del programa que estaba en ejecución se detiene completamente durante la operación de E/S.
- Existen dificultades para atender a varios periféricos al mismo tiempo.

Por eso es una técnica de gestión intuitiva y fácil de programar, pero que no se utiliza prácticamente nunca.

2.9.2. E/S con interrupciones

En este caso no existe ningún tipo de bucle de espera sino que es el periférico el que avisa al procesador cuando está listo para la transferencia.



FIGURA 2.25

E/S programada con espera de respuesta.

Este aviso se realiza activando una línea de petición de interrupción (IRQ). Cuando el procesador recibe esta señal de petición salta a ejecutar una Rutina de Tratamiento de Interrupción (RTI) que se encarga de atender al periférico que solicitó la interrupción y realiza la operación de E/S.

En el caso de un procesador secuencial, la secuencia típica de eventos que se produce en el tratamiento de una interrupción es la siguiente:

1. Un periférico activa la línea de petición de interrupción (IRQ).
2. Cuando el procesador termina de ejecutar la instrucción en curso, comprueba si tiene alguna petición de interrupción pendiente.
3. Si es así, el procesador almacena el contexto de ejecución (contador de programa y registros de estado) en la pila, descapacita las interrupciones y salta a la RTI, durante la cual:
 - Informa al periférico que se ha reconocido su interrupción. Esto puede hacerse por software, accediendo al registro de estado del interfaz, o por hardware, mediante una señal INTA de reconocimiento de interrupción. Entonces el periférico desactiva IRQ.
 - Salva en pila todos los registros de datos y direcciones utilizados por la RTI.

- Realiza la operación de E/S con el periférico.
 - Restaura los registros de direcciones y de datos.
 - Ejecuta la instrucción de retorno de interrupción.
4. El procesador vuelve a capacitar las interrupciones, restaura el contexto y retorna al programa que estaba ejecutando antes de pasar a atender a la interrupción. Después continúa con la ejecución normal del programa

La comprobación de peticiones de interrupción pendientes (línea IRQ activada) se hace al final de la ejecución de cada instrucción, porque así sólo es necesario guardar el contador de programa, el registro de estado y los registros accesibles por programa (registros de datos y direcciones) que modifica la RTI. Si se interrumpiese una instrucción en mitad de la ejecución sería necesario guardar el valor de todos los registros internos del procesador, es decir, el contexto de ejecución sería mucho más complicado y costoso de almacenar y restaurar.

Existen algunas excepciones, por ejemplo, en las instrucciones de muy larga duración, se comprueba si existen peticiones de interrupción pendientes cada cierto tiempo o cuando se producen interrupciones muy prioritarias, que pasan a atenderse inmediatamente.

Sin embargo, prácticamente todos los procesadores de hoy en día incorporan segmentación. Por lo tanto, se estarán ejecutando varias instrucciones al mismo tiempo en el procesador cuando se produzca la petición de interrupción y cada una de ellas estará en una etapa diferente de ejecución. ¿Cómo se tratan en este caso las interrupciones? Las interrupciones de E/S son consideradas excepciones en este tipo de procesador. Por lo tanto, las interrupciones se tratan con las técnicas de resolución de excepciones que se estudiaron en el capítulo 1 para procesadores segmentados (sección 1.5.4).

Hasta ahora se ha considerado la existencia de una única línea de petición de interrupción, pero los procesadores actuales necesitan conectarse a multitud de dispositivos periféricos diferentes, no sólo a uno. Esto obliga en muchos casos a conectar varios dispositivos a una única línea de petición de interrupción y a implementar mecanismos de identificación de la fuente de interrupción.

Cuando existen varias fuentes de interrupción en una misma línea de petición es necesario un mecanismo para identificar al periférico que ha originado la interrupción y ejecutar la RTI adecuada para atender a ese periférico en particular. Existen dos posibilidades:

- **Identificación software por encuesta (polling).** La RTI examina uno a uno los bits de estado de cada periférico hasta hallar el que tiene activado su bit de petición de interrupción. Una vez detectado, se ejecuta su RTI, durante la cual se debe desactivar el bit de petición de interrupción del periférico. El método de encuesta introduce un mecanismo de prioridades, porque en caso de peticiones simultáneas de varios dispositivos, se atiende primero al periférico al que se encuesta primero. El principal problema de esta alternativa es que desperdicia tiempo consultando a dispositivos que no han solicitado interrupción.
- **Identificación hardware por vectores.** En este caso el periférico que ha solicitado la interrupción envía un código o número de vector al procesador a partir del cual se puede calcular la dirección de comienzo de la RTI de ese periférico en la memoria principal. Normalmente, cuando el periférico recibe la señal de confirmación o reconocimiento de interrupción INTA envía su número de vector a través del bus de datos. Este mecanismo es mucho más rápido que el de encuesta y la asignación de prioridades puede realizarse de manera más flexible, el problema es que el número de dispositivos que se pueden identificar con este método depende del número de bits que se utilicen para el número de vector. Por ejemplo, con un número de vector de 4 bits podemos identificar hasta 16 dispositivos, pero no más. Una posible solución es utilizar códigos de grupo, es decir, un mismo número de vector puede utilizarse para identificar a un grupo de varios dispositivos. Cuando el procesador recibe un número de vector de grupo, la RTI debe identificar al dispositivo particular dentro de ese grupo mediante encuesta (por software).

De cualquier manera, la gestión de E/S mediante interrupciones suele implicar la necesidad de un controlador de interrupciones o PIC que se encargue de dar acceso a la única línea de petición de interrupciones que incorporan los procesadores, que se encargue de gestionar la identificación y la prioridad de peticiones simultáneas y que permita implementar esquemas de interrupciones multinivel, con enmascaramiento selectivo, anidamiento, etc. (figura 2.26).

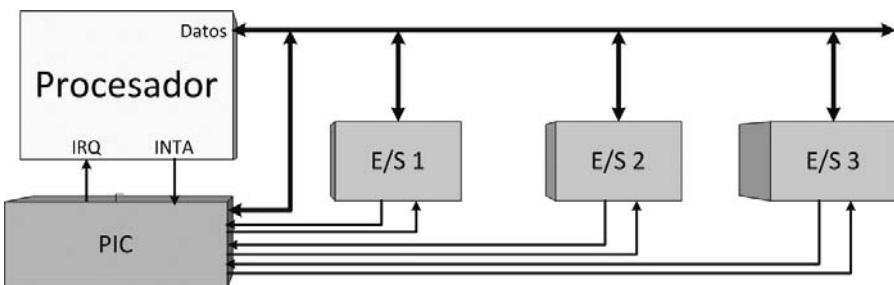


FIGURA 2.26

E/S basada en interrupciones.

Este PIC suele incluirse en los chipsets o hubs del sistema y suele tener varias líneas de petición de interrupción de entrada y una de salida que se conecta directamente a la del procesador. En muchos casos, no se utiliza un único controlador de interrupciones sino dos o más conectados en cascada.

En un sistema de interrupciones multinivel, existen varias líneas o niveles de petición de interrupción y a cada línea de interrupción se pueden conectar uno o varios dispositivos. Además, cada nivel suele tener asignada una prioridad distinta, porque hay que resolver de alguna manera los conflictos que se producen por peticiones simultáneas.

Si se producen peticiones simultáneas por la misma línea, el conflicto se resuelve con alguno de los mecanismos estudiados anteriormente: mediante encuesta (software) o mediante vectores (hardware).

Cuando se producen peticiones simultáneas por líneas distintas el problema se suele resolver mediante un codificador de prioridades dentro del PIC, de manera que se atienda siempre primero a la línea más prioritaria.

Los sistemas de interrupciones multinivel permiten enmascarar o descapacitar selectivamente las interrupciones por determinados niveles. Para ello se utiliza un registro de máscara. En este registro existe un bit para cada nivel de prioridad, de manera que si este bit vale uno el nivel de interrupción correspondiente está habilitado y si vale cero, está inhabilitado.

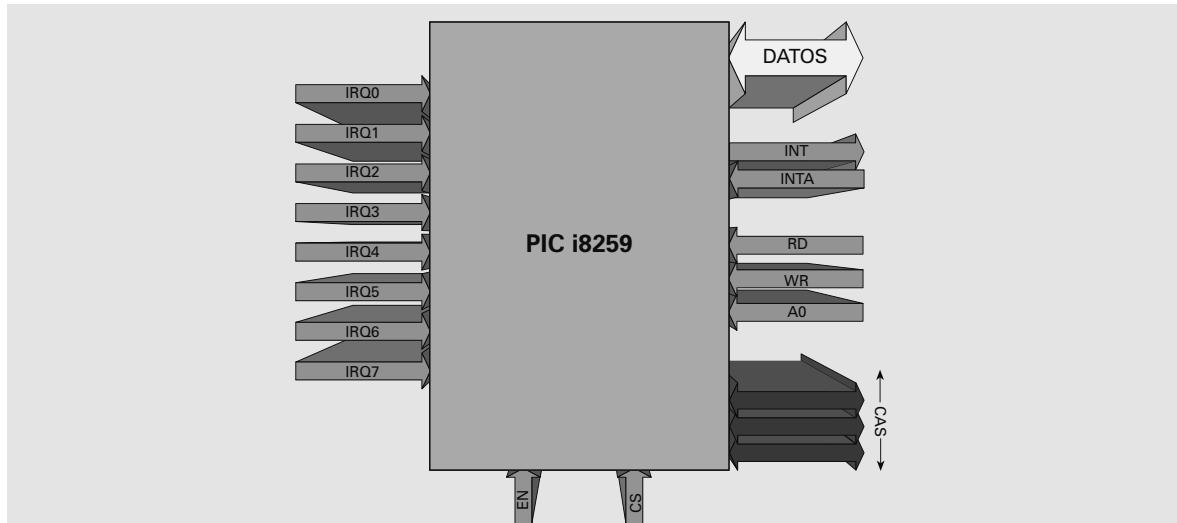
Con este tipo de enmascaramiento se suele implementar el anidamiento de interrupciones. Este anidamiento permite que mientras se ejecuta la RTI de un determinado nivel de prioridad se inhiban las interrupciones por el mismo nivel o inferiores, pero se puedan atender peticiones de interrupción de mayor prioridad.

Ejemplo 2.16

Tratamiento de interrupciones con el PIC i8259.

Supongamos que tenemos en el chipset sur o ICH del sistema dos controladores PIC i8259 conectados en cascada. El esquema de conexión de cada uno de estos controladores es el de la figura de la página siguiente, muy similar de unos diseños de PIC a otros.

Las líneas IRQ0-IRQ7 son las de petición de interrupción a las que se conectan los dispositivos de E/S, de mayor a menor prioridad. INT es la petición de interrupción que le llega al procesador, gestionada por el PIC e INTA es el reconocimiento de la interrupción por parte del procesador. La señal CS no es más que la señal de chip select para escrituras y lecturas en los registros del PIC (no tiene influencia sobre la señal de INTA).



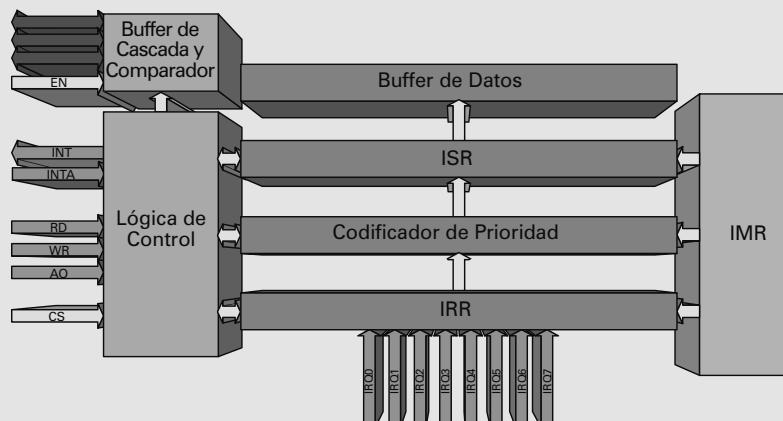
Las líneas CAS2-CAS0 son las líneas de cascada, que se utilizan cuando hay más de un PIC en el sistema. Actúan como salida en el 8259 maestro y como entrada en los 8259 esclavos. Es la señal EN la que indica si el PIC actúa como maestro o como esclavo cuando tenemos varios encadenados.

Por último, las señales RD y WR permiten leer de los registros del PIC o escribir sobre ellos, siendo la señal A0 la que selecciona los registros sobre los que el procesador realiza la operación de lectura/escritura.

Una vez comprendido el esquema de conexión, veamos cómo se diseña el controlador.

Todos los registros internos del microcontrolador PIC son de ocho bits. Los más importantes son:

- IRR (Interrupt Request Register). Cada uno de los bits de este registro está asociado con una de las líneas de petición de interrupción. Estos bits almacenan las peticiones de interrupción pendientes.
- ISR (In-Service Register). De nuevo cada uno de los bits se asocia con una línea de petición de interrupción. Pero en este caso sólo se activa el bit que corresponde a la interrupción que se está procesando en este momento.
- IMR (Interrupt Mask Register). Registro de enmascaramiento de interrupciones.



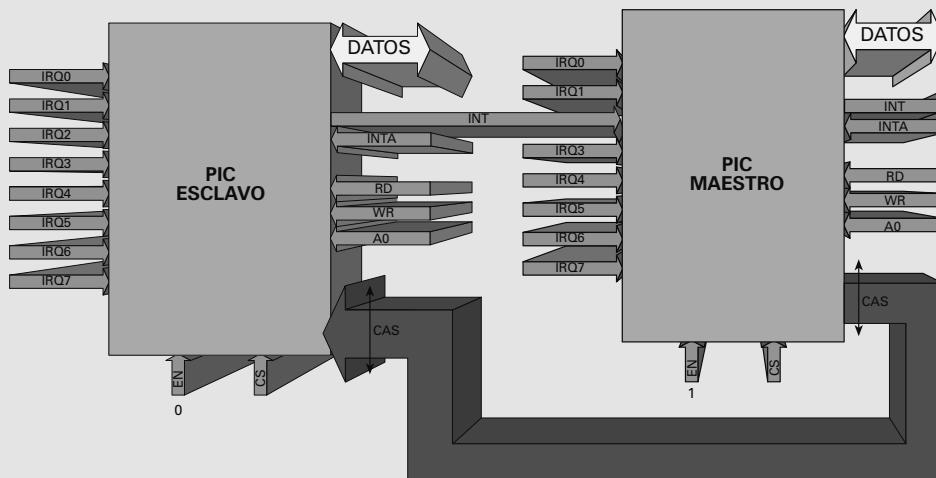
La secuencia de reconocimiento de una interrupción con un único PIC y en el modo de operación más sencillo es la siguiente:

1. Una o más líneas IRQ son activadas por los periféricos conectados al PIC y esto activa los correspondientes bits del IRR.
2. El PIC i8259 evalúa la prioridad de estas interrupciones (mediante un codificador de prioridad) y solicita una interrupción al procesador activando la línea INT.
3. Cuando el procesador reconoce la interrupción envía la señal INTA.

4. Entonces el PIC activa el bit correspondiente a la interrupción de mayor prioridad (la que va a ser procesada) en el ISR y borra ese mismo bit en el IRR. En este ciclo, el i8259 aún no controla el bus de datos.
5. Cuando el procesador envía un segundo INTA, el i8259 deposita en el bus de datos un valor de 8 bits que indica el número de vector de interrupción (type code=offset PIC + nº línea de IRQ). El procesador multiplica este valor por 4 para buscar en esa dirección de memoria la dirección de comienzo de la RTI.
6. El bit de la interrupción en el ISR permanece activo hasta que el procesador envíe el comando EOI (End of Interrupt) al final de la rutina que trata la interrupción.

Con este esquema se pueden atender peticiones de hasta ocho dispositivos. Como esto se queda corto en la mayor parte de los casos, se utilizan dos niveles de controladores, en el primero tenemos al PIC maestro, y en el segundo al PIC esclavo. Sólo la salida INT del maestro está conectada a la entrada INT de petición de interrupción del procesador.

Para conectar los PIC en configuración de cascada se conecta la salida INT del esclavo a la entrada IRQ2 del PIC maestro y sus entradas CAS a las salidas CAS del maestro. Además, la entrada EN debe ser coherente con la función de cada uno de los microcontroladores, debe valer 0 para el esclavo y 1 para el maestro.



Debemos tener en cuenta que la conexión de un PIC esclavo al PIC maestro origina una reestructuración del esquema de prioridades de los niveles de interrupción del computador: se intercalan ocho nuevos niveles entre los que ya tuviera el maestro conectados.

La secuencia de reconocimiento de una petición de interrupción en las líneas del maestro es la misma que ya hemos explicado. Si la petición de interrupción es en una de las líneas de un esclavo, entonces:

1. El PIC esclavo detecta la petición de interrupción por una de sus líneas y activa la señal INT.
2. El PIC maestro detecta la petición de interrupción por la línea a la que se conecta ese esclavo. Activa el correspondiente bit del registro IRR y además la señal INT.
3. El procesador recibe esta señal y envía el primer pulso de INTA (esta señal está conectada a todos los PIC's del sistema, tanto al maestro como a los esclavos).
4. El PIC maestro vuelve en las líneas CAS el número de identificación del PIC esclavo que debe atender la interrupción (el que la originó).
5. Tanto el maestro como el esclavo limpian el bit de la interrupción que se atiende en sus registros IRR y activan esos bits en sus registros ISR.
6. Con el segundo pulso de INTA, el esclavo vuelve en el bus de datos el número del vector de interrupción.
7. El procesador deberá enviar dos comandos de EOI, uno al maestro y otro al esclavo, cuando finalice de ejecutar la RTI. Para que esto funcione así, la RTI debe saber en cada caso si está atendiendo a una interrupción gestionada directamente por el maestro o a través de una pareja maestro-esclavo. Esto es necesario para que envíe uno o dos comandos EOI respectivamente.

Con esta configuración maestro-esclavo la flexibilidad en el tratamiento de interrupciones es elevada, pudiendo trabajar con distintas formas de anidamiento y enmascaramiento, permitiendo la rotación de prioridades, etc.

2.9.3. E/S con acceso directo a memoria

La gestión de E/S programada con espera de respuesta o por interrupciones resulta poco eficiente para dispositivos periféricos de alta velocidad, sobre todo si hay que transferir una gran cantidad de información en la operación de entrada o salida. Además, ambas técnicas requieren de tiempo de procesador que se podría invertir en realizar otras tareas durante la transferencia de información.

La técnica de Acceso Directo a Memoria (DMA) permite la transferencia de datos entre un periférico y la memoria principal sin intervención del procesador, salvo en la fase de inicialización de los parámetros de la transferencia (figura 2.27).

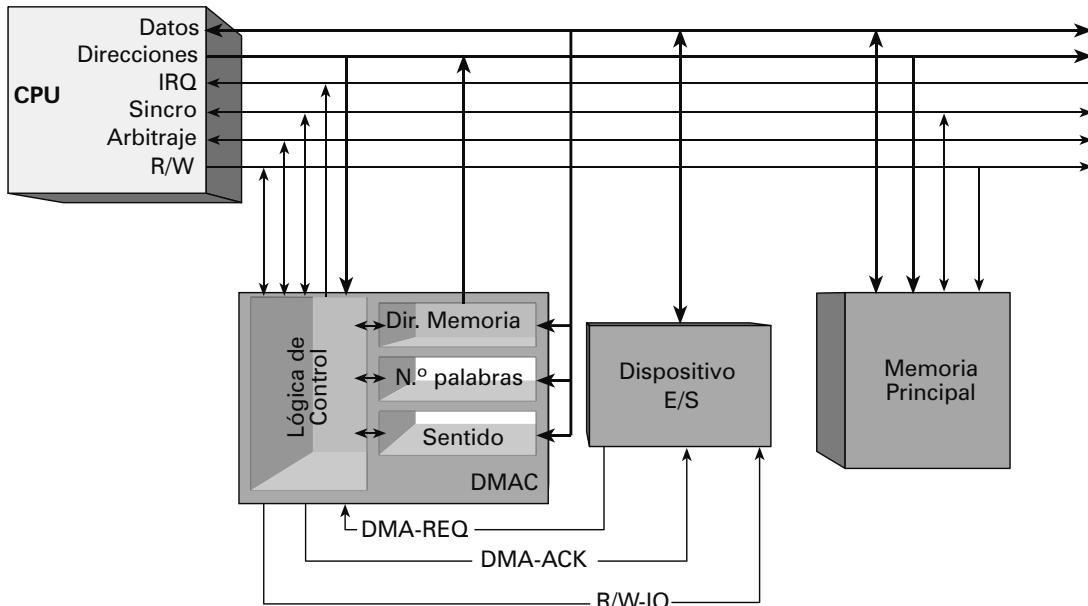


FIGURA 2.27

E/S basada en acceso directo a memoria (DMA).

En este caso es de nuevo necesario un controlador, el controlador de DMA (DMAC), que descargue de trabajo al procesador en lo que se refiere a las tareas de E/S. Este controlador también suele integrarse en los chipsets o hubs del sistema.

El controlador de DMA (DMAC) se encarga de la transferencia de datos entre un periférico y la memoria principal sin intervención del procesador. Por lo tanto debe actuar como maestro del bus de memoria durante la transferencia DMA y debe ser capaz de:

- Solicitar el uso del bus mediante las señales y la lógica de arbitraje necesarias.
- Especificar la dirección de memoria sobre la que se realiza la transferencia.
- Generar las señales de control del bus: tipo de operación (lectura/escritura), señales de sincronización, etc.

Este controlador suele incluir un registro de dirección de memoria que almacena la dirección inicial de memoria para la transferencia y que se incrementa/decremente después de transferir cada palabra. Además suele haber un registro de número de palabras para almacenar el número de palabras a transferir y que se decremente después de transferir cada palabra. Y por último, suele haber un registro de sentido que indique el sentido de la transferencia de información desde el punto de vista del procesador, lectura o escritura.

Normalmente una transferencia por DMA se divide en tres fases:

1. **Inicialización de la transferencia.** El procesador envía al interfaz del periférico los parámetros de la transferencia: n.º de bytes a transferir, tipo de transferencia (lectura/escritura) y otra información de control. Además, accede a los registros del DMAC para programar los parámetros de la transferencia: dirección inicial de memoria, número de palabras que se van a transferir y sentido de la transferencia. El procesador regresa a sus tareas y ya no se preocupa más de la evolución de la transferencia de E/S.
2. **Realización de la transferencia.** Cuando el periférico está listo para transmitir/recibir lo indica al DMAC activando la señal DMA-REQ. El DMAC solicita el control del bus de memoria mediante las líneas de arbitraje. El DMAC recibe la concesión del bus y activa la señal DMA-ACK para indicar al periférico que puede iniciar la transferencia. El DMAC debe generar y procesar las señales del bus adecuadas: dirección de memoria sobre la que se realiza la transferencia, señales de sincronización de la transferencia, señales de lectura/escritura, etc. Despues de transferir cada palabra el DMAC debe actualizar sus registros decrementando el registro de número de palabras e incrementando/decrementando el registro de direcciones de memoria.
3. **Finalización de la transferencia.** El DMAC libera el bus de memoria y solicita una interrupción para indicar al procesador la finalización de la operación de E/S solicitada.

Una operación de E/S gestionada de esta manera mediante DMA puede degradar el rendimiento del procesador si el DMAC hace uso intensivo del bus de memoria, ya que si el bus está ocupado en una transferencia DMA, el procesador no puede acceder a la memoria principal.

Este problema se reduce con el uso de la jerarquía de memoria, ya que la mayor parte del tiempo, el procesador accede a la memoria caché y no necesita utilizar la memoria principal.

Aún así, se distinguen dos modos de transferencia en DMA:

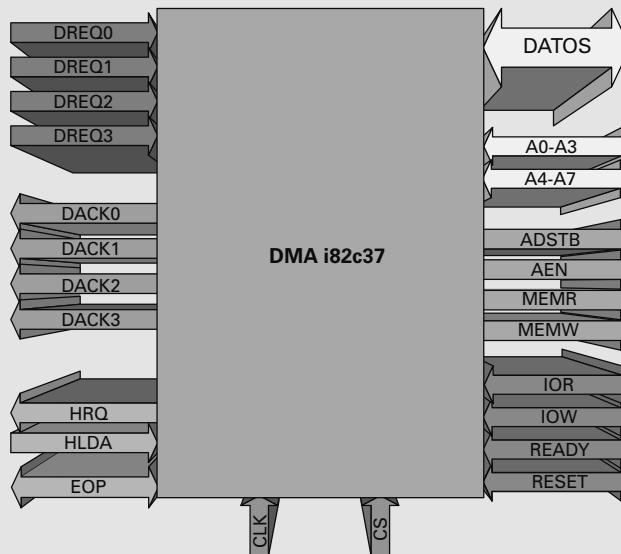
- **Transferencia DMA modo ráfaga.** El DMAC solicita el control del bus y una vez que lo obtiene no lo libera hasta haber finalizado la transferencia de todo el bloque de datos completo (cuando el registro de número de palabras llega a cero). Por lo tanto, la transferencia se realiza de forma rápida, pero durante el tiempo que dura la transferencia el procesador no puede utilizar la memoria principal, lo que puede degradar el rendimiento del sistema.
- **Transferencia DMA modo robo de ciclo.** El DMAC solicita el control del bus y cuando lo obtiene, se realiza la transferencia de una única palabra. Despues el DMAC libera el bus y vuelve a solicitar su control tantas veces como sea necesario hasta haber finalizado la transferencia del bloque completo palabra a palabra. De esta manera no se degrada el rendimiento del sistema aunque la transferencia tarda más tiempo en llevarse a cabo.

Sea cual sea el modo de transferencia escogido, es necesario trabajar con direcciones de memoria para utilizar este mecanismo de E/S. Pero, ¿con qué tipo de direcciones trabaja el DMAC, virtuales o físicas? De nuevo existen dos opciones de diseño:

- **DMA Virtual.** El DMAC utiliza direcciones virtuales que deben ser traducidas a físicas antes de realizar los accesos a memoria principal.
- **DMA Físico.** En este caso se utilizan direcciones físicas y no es necesario este proceso de traducción. Pero es necesario que el sistema operativo ubique en la memoria principal las páginas implicadas en la operación de E/S y que estén en marcos consecutivos, porque si no, al incrementar/decrementar la dirección de memoria se terminará por pasar la frontera de un marco de página y se accederá a datos incorrectos. Además estas páginas deben bloquearse, de manera que no se remplacen hasta que la transferencia DMA finalice.

Ejemplo 2.17**Acceso directo a memoria con el DMAc i82c37.**

De nuevo vamos a estudiar un ejemplo con un controlador de DMA concreto. Éste es su esquema de conexión:



Las señales DREQ0-DREQ3 son las de petición de servicio DMA de los periféricos, ya que este controlador es capaz de atender cuatro canales de acceso directo a memoria. Las señales DACK0-DACK3 son las de concesión de servicio DMA para estos cuatro canales.

La señal CS (Chip Select) es la que debe activar el procesador para acceder a los registros del controlador. CLK es la señal de reloj común al procesador, la memoria principal, el DMAc y los periféricos. Y RESET es la señal de reinicio del controlador, que pone a cero todos los registros programables por el procesador. EOP (End of Process) es una señal bidireccional con dos funciones: la utilizan los periféricos para terminar una transferencia DMA y también el DMAc para informar a los periféricos del final de una transferencia.

La señal HRQ (Hold Request) es la de petición de control del bus de memoria (BUS REQ) y la HLDA (Hold Ack) la de concesión de control del bus (BUS GNT). La señal READY es la señal que activan la memoria principal o el periférico cuando no pueden realizar la transferencia DMA en el tiempo mínimo establecido (protocolo semisíncrono).

Las señales MEMR y MEMW son activadas por el DMAc para indicar el tipo de transferencia que se va a realizar, MEMR para una lectura de memoria y MEMW para una escritura en memoria.

IOR e IOW son muy similares pero tienen dos funciones. Durante la inicialización del controlador, con estas señales indica el procesador si va a leer (IOR) o escribir (IOW) en alguno de los registros del DMAc. Durante una transferencia DMA, las activa el DMAc para indicar el tipo de transferencia desde el punto de vista del periférico, IOR para una lectura del periférico e IOW para una escritura.

El resto de señales tienen que ver con el control del direccionamiento de memoria.

Una vez comprendido el esquema de conexión del controlador, es más sencillo comprender su estructura interna, compuesta casi enteramente por registros:

- Registro base de dirección (16 bits). Almacena la dirección de memoria inicial para la transferencia DMA. Este registro es inicializado por el procesador y no se modifica durante toda la transferencia. Tenemos un registro de este tipo para cada canal, por lo que son cuatro en total.
- Registro base contador de palabras (16 bits). Almacena el número de palabras que se van a transferir (menos una) y es inicializado por el procesador (tampoco se modifican durante la transferencia). También tenemos cuatro registros de este tipo, uno para cada canal.
- Registro de dirección actual (16 bits). Cada canal tiene un registro de dirección actual que almacena la dirección de memoria de la siguiente palabra a transferir. Su contenido es incrementado/decrementado después de cada transferencia (según las direcciones de memoria sean crecientes o decrecientes).

- Registro contador de palabras actual (16 bits). Cada canal tiene un registro contador de palabras actual, que almacena el número de bytes que quedan por transferir. Tras cada transferencia se decrementa y cuando llega a cero, se finaliza la transferencia actual.
- Registro de petición (4 bits). Sirve para realizar peticiones de DMA por software. Este registro es necesario para las transferencias memoria-memoria, puesto que una memoria no es capaz de generar una petición hardware de DMA para comenzar una transferencia. Posee un bit para cada canal de DMA. Las peticiones por software no se pueden enmascarar, aunque están sujetas a la lógica de evaluación de prioridades.
- Registro temporal (8 bits). Se emplea para almacenar los bytes que se transfieren en las operaciones memoria-memoria. Tras completar el proceso de transferencia, el procesador puede averiguar la última palabra transferida leyendo este registro.
- Registro de máscara (4 bits). Cada canal tiene asociado un bit de máscara en este registro, que puede ser activado para inhibir las solicitudes de DMA a través de la línea DREQ. Este bit es automáticamente activado cada vez que se produce un EOP (al final de la transferencia). Los bits de máscara pueden modificarse por separado o todos a la vez.
- Registro de estado (8 bits). En los cuatro primeros bits almacena las peticiones de servicio DMA de los cuatro canales y en los cuatro siguientes, si se ha finalizado la transferencia en alguno de ellos (TC: Terminal Count).
- Registro de comandos (8 bits) y Registro de modo (6 bits). Registros que permiten al procesador programar el modo de operación y el tipo de transferencia.

Estos controladores permiten tres modos de transferencia: modo simple, modo bloque y modo demanda, así como multitud de configuraciones y optimizaciones diferentes, la realización de transferencias memoria-memoria, etc. Para terminar este ejemplo, vemos cómo se realizaría una transferencia sencilla en modo bloque:

1. El procesador programa al DMAC y al periférico para la transferencia completa de información.
2. El periférico solicita el servicio DMA cuando está listo, activando DREQ.
3. El DMAC solicita el control del bus activando HRQ.
4. El procesador concede el uso del bus activando HLDA.
5. El DMAC activa DACK para avisar al periférico de que puede comenzar la transferencia. Las señales DREQ y DACK se desactivan.
6. Se realiza la transferencia de una palabra. Esta transferencia dura como mínimo tres ciclos de reloj. Si el periférico o la memoria necesitan más tiempo, deben utilizar la señal READY.
7. Se actualizan los registros de dirección y cuenta de palabras actuales.
8. Se continúa hasta que el registro contador de palabras llega a cero. Entonces se activan la señal EOP y el bit TC (Terminal Count) adecuado del registro de estado.

Como el registro de cuenta de palabras puede ir desde FFFF hasta 0, podemos realizar transferencias de hasta 64 KB desde una misma dirección inicial.

También con los controladores de DMA tenemos la opción de trabajar con varios controladores en cascada si se necesitan más de cuatro canales. Suele ser habitual trabajar con dos controladores, la línea HRQ del i82c37 hijo se conecta a la DREQ0 del i82c37 padre y la HLDA a la DACK. Esto permite que las peticiones en el hijo se propaguen hasta el padre.

Teniendo en cuenta que el canal del i82c37 padre es empleado sólo para dar una determinada prioridad, el i82c37 hijo no puede emitir direcciones ni señales de control por sí mismo: esto podría causar conflictos con las salidas del canal activo en controlador hijo. Por tanto, el padre se limita en el canal al que se conecta el hijo a controlar DREQ, DACK, HLDA y HRQ, dejando inhibidas las demás señales.

Por último hay que aclarar que la utilización de una jerarquía de memoria provoca que en muchos casos existan como mínimo tres copias de la misma información en el sistema: en memoria caché, en memoria principal y en memoria virtual.

Esto implica complicaciones en el diseño del hardware y del sistema operativo cuando se introduce este tipo de gestión de E/S que permite que un periférico y la memoria principal se transfieran información directamente, ya que se debe asegurar que:

- Si se modifican bloques de la memoria principal con una operación de entrada (escritura en memoria), deberán invalidarse en la memoria caché los bloques afectados por la operación.
- Antes de una operación de salida (lectura de memoria principal), deberá actualizarse este nivel de la jerarquía desde los buffers de escritura de la memoria caché, las memorias caché de post-escritura, etc.

Resumen de decisiones de diseño de la jerarquía de memoria

MEMORIA CACHÉ

Decisión	Alternativas	Decisiones asociadas
Tamaño de la memoria caché		
Tamaño del marco/bloque		
Número de niveles de memoria caché		
Contenidos de cada nivel de memoria caché	Memorias unificadas para instrucciones y datos	
	Memorias divididas	
Política de emplazamiento	Directa	
	Asociativa	
	Asociativa por conjuntos	Número de vías (marcos por conjunto)
Política de reemplazamiento	Aleatoria	
	LRU	
	FIFO	
	Otras	
Política de escritura	Escritura Directa	
	Post-escritura	
Tratamiento de los fallos de escritura	Sin asignación en escritura	
	Con asignación en escritura	
Utilización de un buffer de escritura		
Utilización de mejoras sencillas	Palabra crítica primero	
	Rearranque rápido	
	Paralelismo entre accesos y transferencias, entre accesos a varios niveles de la jerarquía, etc.	

MEMORIA PRINCIPAL

Decisión	Alternativas	Decisiones asociadas
Tamaño de la memoria principal		
Tecnología de los módulos de memoria		
Conexión de los módulos de memoria	Paralela (tipo DDR)	
	Serie (tipo XDR)	
Técnicas de detección y corrección de errores		
Política de reemplazamiento de páginas/segmentos	LRU	
	FIFO	
	Otras	

MEMORIA VIRTUAL

Decisión	Alternativas		Decisiones asociadas
Tamaño de la memoria virtual			
Organización de la memoria virtual	Paginada		
	Segmentada		
	Paginada/segmentada		
Traducción de dirección virtual a dirección física	Paginada	Directa Asociativa Mixta Por niveles	
	Segmentada	Directa	

BUSES DE E/S

Decisión	Alternativas	Decisiones asociadas
Número máximo de dispositivos		
Longitud máxima del bus		
Niveles de tensión		
Frecuencia de funcionamiento		
Especificación de conectores		
Número de líneas		
Función de las líneas	Datos	
	Direcciones	
	Control	
Tipo de líneas	Unidireccionales	
	Bidireccionales	
Utilización de las líneas	Dedicadas	
	Multiplexadas	
Protocolo de transferencia		Número de transferencias por ciclo
Protocolo de sincronización	Síncrono	
	Asíncrono	
	Semisíncrono	
	De ciclo partido	
Protocolo de arbitraje	Centralizado	
	Distribuido	
Jerarquía de buses		

GESTIÓN E/S

Decisión	Alternativas	Decisiones asociadas
Tipo de mecanismo de gestión de E/S	Espera de respuesta	
	Interrupciones	Número, ubicación y tipo de controladores PIC Identificación de fuentes de interrupción (SW o HW) Asignación de prioridades Anidamiento
	Acceso Directo a Memoria	Número, ubicación y tipo de controladores DMA Modo de transferencia (ráfaga o robo de ciclo) Direccionamiento (virtual o físico)

BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS

- HAMACHER, C.; VRANESIC, Z. & ZAKY, S. (2001): *Computer Organization* (5.^a ed.), McGraw Hill.
- HANDY, J. (1998): *The cache memory book* (2.^a ed.), Morgan Kaufmann.
- JACOB, B.; SPENCER, NG. & WANG, D. (2007): *Memory Systems: Cache, DRAM, Disk*, Morgan Kaufmann.
- OMONDI, A. R. (1999): *The Microarchitecture of Pipelined and Superscalar Computers* (1.^a ed.), Springer.
- PARHAMI, B. (2007): *Arquitectura de Computadoras. De los microprocesadores a las supercomputadoras* (1.^a ed.), McGraw Hill.
- PATTERSON, D. A. & HENNESSY, J. L. (2008): *Computer Organization and Design: The Hardware/Software Interface* (4.^a ed.), Morgan Kaufmann.
- PRINCE B. (1996): *Semiconductor Memories: A Handbook of Design, Manufacture and Application*, Wiley.
- SHARMA A. K. (2002): *Advanced Semiconductor Memories: Architectures, Designs, and Applications*, Wiley-IEEE Press.
- SILBERSCHATZ, A.; GALVIN, P. B. & GAGNE, G. (2008): *Operating Systems Concepts* (8.^a ed.), Wiley.
- STALLINGS, W. (2007): *Organización y arquitectura de computadores* (7.^a ed.), Pearson-Prentice Hall.
- TANNENBAUM, A. S. (2005): *Structured Computer Organization* (5.^a ed.), Prentice Hall.

PROBLEMAS

- 2.1.** Discutir qué aplicaciones reales de ejecución habitual cumplen las siguientes características:
- Apenas se puede aprovechar la localidad espacial.
 - Apenas se puede aprovechar la localidad temporal.
 - Buen aprovechamiento de la localidad espacial.
 - Buen aprovechamiento de la localidad temporal.
 - Buen aprovechamiento de los dos tipos de localidad.
- 2.2.** Una computadora tiene una memoria virtual de 4 GB, una memoria principal de 1 GB y una memoria caché de 128 marcos de bloque. El tamaño de página es de 4 KB, el de bloque es de 64 palabras y la palabra son 32 bits. Con esta jerarquía de memoria:
- Explicar el formato de la dirección virtual y de la dirección física (con todas sus interpretaciones posibles) para una memoria caché directa, asociativa por conjuntos de 8 vías y asociativa.
 - Calcular en cada tipo de caché el espacio necesario para almacenar datos y para almacenar etiquetas.
- 2.3.** Una aplicación accede a las siguientes direcciones de memoria: 0,1,2,3,4,13,12,15,14,10,9,32,56, 28,0,4,10. Discutir qué fallos iniciales, de conflicto y de capacidad se producen al realizar estos accesos con las siguientes organizaciones de memoria caché:
- 4 marcos de 4 palabras cada uno, emplazamiento directo.
 - 4 marcos de 4 palabras cada uno, emplazamiento asociativo con reemplazamiento LRU.
 - 8 marcos de 2 palabras cada uno, emplazamiento asociativo por conjuntos de 2 vías con reemplazamiento LRU.
- 2.4.** Se ejecuta el siguiente código en un procesador que incluye un nivel de memoria caché con dos marcos de cuatro palabras y emplazamiento directo:
- ```
for (i=0;i++;i<8)
 temp=A(i)*B(i)
 sum=sum+temp;
```
- La dirección de inicio del vector A es dirA y la dirección de inicio del vector B es dirB. Las variables temp y sum se almacenan en registros internos del procesador.
- Evaluar los fallos de caché que se producen al ejecutar este código, discutir su dependencia con dirA y dirB, y proponer soluciones que eviten los fallos que se producen.
- 2.5.** Nos estamos planteando introducir un nivel de memoria caché en una jerarquía de memoria con las siguientes características:
- El tiempo de acceso a memoria principal es 100 veces el tiempo de acceso a memoria caché.
  - Los bloques son de 4 palabras.
  - La memoria caché tiene una tasa de fallos del 5%.
  - El 25% de los accesos a memoria son escrituras y el 75% restante son lecturas.

La memoria caché podría tener una de estas dos organizaciones, en ambos casos con asignación en escritura:

- Alternativa 1: Caché de emplazamiento directo y con escritura directa.
- Alternativa 2: Caché de emplazamiento asociativo y con post-escritura sabiendo que en media el 20% de los bloques son modificados mientras se encuentra en la memoria caché.

- a) Realizar un análisis cualitativo acerca de cuál de las dos alternativas sería más conveniente. ¿En qué casos obtendríamos mejor rendimiento con cada una de las alternativas? ¿Dónde están las principales diferencias?
- b) Realizar un análisis cuantitativo acerca de cuál de las dos alternativas sería más conveniente. ¿Con cuál de las dos alternativas se podría obtener un speedup mayor respecto de la jerarquía sin memoria caché?
- 2.6.** Se diseña una jerarquía de memoria con un único nivel de memoria caché, con instrucciones y datos separados. El procesador realiza  $5 \cdot 10^6$  referencias por segundo a datos (de las cuales el 80% son lecturas y el 20% son escrituras) y  $10^7$  referencias por segundo a instrucciones (obviamente, todas lecturas). La memoria caché de instrucciones tiene un tiempo de acceso de 1.5 ns, una tasa de fallos del 8% y el tamaño de bloque es de 32 palabras. La memoria caché de datos tiene un tiempo de acceso de 1.2 ns, una tasa de fallos del 9% y un tamaño de bloque de 16 palabras. La latencia de memoria principal es de 92 ns.
- Calcular el tiempo medio de acceso a memoria para instrucciones y para datos si la caché es de escritura directa.
  - Calcular el ancho de banda con memoria principal consumido con esta caché de escritura directa.
  - Calcular el tiempo medio de acceso a memoria para instrucciones y para datos si la caché es de post-escritura, sabiendo que el 18% de los bloques son modificados mientras se encuentran en la caché de datos.
  - Calcular el ancho de banda consumido con esta caché de post-escritura.
  - Comparar el tiempo medio de acceso y el ancho de banda consumido para las dos políticas de escritura y justificar los resultados obtenidos.
  - Si finalmente se incluye la mejor alternativa de las dos en el procesador, sabiendo que su CPI, sin tener en cuenta la memoria, es de 1 y que su frecuencia de funcionamiento es de 2.7 GHz, calcular el CPI real teniendo en cuenta la jerarquía de memoria. Para ello es necesario saber que el 28% de las instrucciones son cargas de memoria y el 7% son almacenamientos.
- 2.7.** Repetir el ejercicio anterior utilizando la técnica de palabra crítica primero. ¿Cuál es la ganancia obtenida?
- 2.8.** Se diseña una jerarquía de memoria con dos niveles de caché y las siguientes características:
- Caché de nivel 1 de emplazamiento directo y escritura directa, unificada, con un tiempo de acceso de 1 ns, una tasa de fallos del 11% y tamaño de bloque de 16 palabras.
  - Caché de nivel 2 asociativa por conjuntos de 4 vías y de post-escritura, unificada, con un tiempo de acceso de 9 ns, una tasa de fallos del 6% y un tamaño de bloque de 32 palabras.
  - Memoria principal con una latencia de acceso de 92 ns.
  - Tiempos de transferencia de información entre niveles despreciables.
- Sabiendo que el 75% de los accesos a memoria son lecturas y el 25% restante son escrituras,
- Calcular el speedup que se puede conseguir en esta jerarquía de memoria utilizando la técnica de palabra crítica primero en la caché de nivel 1.
  - Calcular el speedup que se puede conseguir en esta jerarquía de memoria utilizando la técnica de re-arranque rápido en la caché de nivel 1 (en media siempre habrá que esperar a traer hasta la memoria caché de nivel 1 la mitad del bloque que resuelve el fallo para que llegue la palabra que ha solicitado el procesador).
  - Discutir los resultados obtenidos y la conveniencia de ambas soluciones teniendo en cuenta además de las ganancias calculadas, la complejidad hardware de cada una de las soluciones.

**2.9.** En el caso de la caché de escritura directa del problema 2.7, escoger la mejor opción de entre estas dos para mejorar su rendimiento justificando la elección con valores de tiempo medio de acceso a memoria:

- a) Incluir en la jerarquía un buffer de escritura con un tiempo de acceso de 2 ns y que evita realizar las escrituras directas a memoria principal un 84% de las veces.
- b) Convertir la memoria en una caché sin asignación en escritura.

**2.10.** Se diseña una jerarquía de memoria con dos niveles de caché y las siguientes características:

- Caché de nivel 1 de instrucciones con un tiempo de acceso de 1 ns, una tasa de fallos del 5% y un tamaño de bloque de 32 palabras.
- Caché de nivel 1 de datos de escritura directa, con un tiempo de acceso de 2 ns, una tasa de fallos del 8% y un tamaño de bloque de 16 palabras.
- Caché de nivel 2 unificada con post-escritura, con un tiempo de acceso de 12 ns, una tasa de fallos del 6% y un tamaño de bloque de 64 palabras. El 19% de los bloques son modificados mientras están en este nivel de caché.
- Memoria principal con una latencia de acceso de 80 ns.
- Conexión de la memoria principal con la memoria caché de nivel 2 que permite transferir 1 palabra en 0.4 ns.
- Conexión de la memoria caché de nivel 2 con la memoria caché de nivel 1 que permite transferir 1 palabra en 0.1 ns.

Con todos estos datos y sabiendo que se utiliza siempre la técnica de palabra crítica primero:

- a) Calcular el tiempo medio de acceso a memoria para instrucciones.
- b) Calcular el tiempo medio de acceso a memoria para datos sabiendo que el 85% de los accesos son lecturas y el 15% restante son escrituras.
- c) Si esta jerarquía se incluye en un procesador con CPI ideal de 1 y frecuencia de funcionamiento de 2 GHz, calcular el CPI teniendo en cuenta los tiempos de acceso calculados en los dos apartados anteriores. Para ello tener en cuenta que el 25.5% de las instrucciones son load, y el 4.5% son store.

**2.11.** Si en la jerarquía de memoria del problema anterior se añade un buffer de escritura a la caché de datos del nivel 1 que permite realizar la escritura de una palabra en 1 ns y que evita tener que realizar la escritura directa en el segundo nivel de caché un 80% de las ocasiones, calcular de nuevo el CPI del procesador teniendo en cuenta a la jerarquía de memoria.

**2.12.** Se diseña una jerarquía de memoria con dos niveles de caché que utilizan la técnica de palabra crítica primero y con las siguientes características:

- Caché de nivel 1 de escritura directa sin asignación en escritura, unificada, con un tiempo de acceso de 1.2 ns, una tasa de fallos del 6% y tamaño de bloque de 8 palabras.
- Caché de nivel 2 de post-escritura, unificada, con un tiempo de acceso de 11 ns, una tasa de fallos del 10% y un tamaño de bloque de 32 palabras. El 30% de los bloques son modificados mientras están en este nivel de la memoria caché.
- Memoria principal con una latencia de acceso de 90 ns.
- Tiempos de transferencia de información entre niveles despreciables.

Con todos estos datos calcular el tiempo medio de acceso a memoria sabiendo que el 70% de los accesos son lecturas y el 30% restante son escrituras.

## AUTOEVALUACIÓN

1. ¿Qué tecnologías se utilizan en la actualidad para fabricar los diferentes niveles de una jerarquía de memoria? ¿Por qué?
2. ¿Cómo se calcula el tiempo medio de acceso a memoria cuando se tiene una jerarquía de memoria con un nivel de caché, una memoria principal y una memoria virtual?
3. ¿Qué alternativas existen para el emplazamiento de datos en una memoria caché? ¿Cuáles son las ventajas e inconvenientes de cada una de ellas?
4. ¿Para qué se utiliza un buffer de escritura en una memoria caché de escritura directa? ¿Y en una memoria caché de post-escritura?
5. ¿En qué consiste la técnica de palabra crítica primero?
6. ¿Cuáles son las mejoras que han introducido las memorias DDR, DDR2 y DDR 3 respecto de las SDRAM?
7. ¿Qué alternativas existen para traducir la dirección virtual a dirección física en el caso de una memoria virtual paginada?
8. ¿Cómo se calcula el ancho de banda de un bus?
9. ¿Cuáles son las jerarquías de buses habituales en las computadoras personales actuales?
10. ¿Qué diferencias hay entre la gestión de E/S mediante interrupciones y acceso directo a memoria (DMA)?

# 3

---

## Técnicas de aumento de prestaciones para procesadores

---

### Contenidos

- 3.1. Planificación dinámica de instrucciones
- 3.2. Predicción dinámica de saltos
- 3.3. Emisión múltiple de instrucciones
- 3.4. Especulación
- 3.5. Multithreading

En el capítulo 1 de este libro se ha propuesto una metodología para diseño de procesadores RISC cuya última fase o etapa consiste en optimizar el primer diseño obtenido, capaz de ejecutar el repertorio de instrucciones escogido y con las técnicas de diseño básicas estudiadas en ese primer capítulo, ajustándose a las especificaciones de área y coste, a la funcionalidad del procesador, a las limitaciones de calor y potencia, etc.

El objetivo de este capítulo es analizar todas las técnicas y herramientas disponibles en la actualidad para llevar a cabo esta optimización. El capítulo comienza estudiando técnicas dinámicas para la resolución de riesgos de datos y control en procesadores segmentados, la planificación dinámica y la predicción dinámica de saltos respectivamente. Estas técnicas son esenciales para evitar las paradas en rutas de datos multifuncionales o bifurcadas.

El objetivo de estas técnicas es conseguir el CPI ideal de 1 que se persigue con la segmentación resolviendo dentro del propio procesador los riesgos causados por los diferentes tipos de dependencias de datos y de control, si es necesario, ejecutando las instrucciones fuera del orden en el que aparecen en el código. Pero se puede intentar reducir el CPI por debajo de 1 empleando para ello técnicas de emisión múltiple de instrucciones, es decir, técnicas que permitan en cada ciclo comenzar, ejecutar y terminar más de una instrucción.

En este libro se introducen dos alternativas para realizar esta emisión múltiple: los procesadores VLIW y los procesadores superescalares. En el primer caso, es esencial el trabajo realizado por el compilador para conseguir el rendimiento deseado. En el segundo caso, se suele combinar la emisión múltiple con especulación basada en hardware. Es decir, si se pretende aumentar el rendimiento del procesador emitiendo más de una instrucción por ciclo, es necesario que el hardware tenga la capacidad de especular, de avanzar en la ejecución de instrucciones que todavía no es seguro que se deban ejecutar ya que dependen de predicciones anteriores, para que pueda aprovechar así al máximo todos los recursos de ejecución disponibles.

Todas estas técnicas se basan en explotar el paralelismo a nivel de instrucción (ILP), limitado siempre por las características de los códigos que se ejecutan. Por ello, para terminar el capítulo, se presenta el multithreading, que intenta superar estas limitaciones explotando el paralelismo a nivel de thread.

## 3.1 Planificación dinámica de instrucciones

En el capítulo 1 se ha visto cómo los riesgos de datos RAW pueden empeorar significativamente el rendimiento de los procesadores segmentados multifuncionales y cómo el adelantamiento, la única técnica hardware estudiada hasta el momento para solucionar estos riesgos de datos, no puede evitar la mayor parte de las paradas provocadas por estos riesgos. De hecho, ni siquiera en procesadores unifuncionales muy sencillos esta técnica nos garantiza que no se produzcan paradas debidas a riesgos de datos.

Resumiendo, con lo estudiado hasta ahora:

- Si al decodificar una instrucción alguno de sus operandos depende de una instrucción que está actualmente en ejecución en el procesador y esta dependencia provoca un riesgo, se intenta solucionar este riesgo con un cortocircuito o adelantamiento.
- Si esto no es posible, se para el avance de la instrucción hasta que los operandos que se necesitan estén disponibles.
- Además el compilador y/o el desarrollador pueden hacer una planificación estática que ordene las instrucciones de manera que se eviten los riesgos originados por las dependencias.

La planificación dinámica de instrucciones intenta reducir al máximo el número de paradas del procesador modificando el orden de las instrucciones en tiempo de ejecución para aprovechar al máximo el

paralelismo a nivel de instrucción (ILP) y que los recursos del procesador estén desocupados el menor tiempo posible.

Cuando se utilizan técnicas de planificación dinámica la etapa F de búsqueda de instrucción suele complicarse ligeramente, ya que se buscan instrucciones en la memoria caché de instrucciones a la mayor velocidad posible y se almacenan, normalmente, predecodificadas, en una cola de instrucciones de donde se pueden ir retirando para planificar cuáles se deben ejecutar en el procesador en cada momento. La predecodificación de instrucciones suele realizarse cuando las instrucciones pasan de la caché de nivel 2 a la caché de nivel 1 para que luego la decodificación sea más rápida. Esta predecodificación suele añadir entre 3 y 8 bits de información a cada instrucción.

La parte del procesador que se encarga de las tareas asociadas a la etapa F suele denominarse Front-End, mientras que el núcleo de planificación, ejecución y escritura de resultados suele denominarse Back-End.

La etapa D se suele dividir en dos fases, en la primera (que normalmente se sigue denominando D de decodificación) se realiza la decodificación de las instrucciones y se comprueban los riesgos, mientras que en la segunda se realiza la lectura de operandos (LO).

Tras la etapa de decodificación las instrucciones se suelen enviar a algún tipo de estructura hardware (normalmente denominada ventana de instrucciones) donde esperan a poder ejecutarse. La lectura de operandos puede realizarse antes de enviar las instrucciones a esta estructura, encargada de realizar la planificación, o ya en la propia ventana de instrucciones.

De hecho la planificación dinámica se basa en buscar y decodificar las instrucciones en el orden secuencial en el que aparecen en el código, pero en permitir que la lectura de operandos y las etapas posteriores se realicen fuera de orden. Es decir, la emisión de instrucciones (E), que no es más que comprobar que éstas tienen disponibles sus operandos y las unidades funcionales que necesitan para ejecutarse, puede desordenarse para que los riesgos de datos no obliguen a parar el procesador.

La ejecución de instrucciones fuera de orden conlleva la aparición de riesgos asociados a dependencias WAR y WAW incluso en el caso del procesador segmentado unifuncional, por lo que estos riesgos, que hasta ahora no habían aparecido para este tipo de procesador cuando todas las instrucciones pasaban por las mismas etapas, deberán resolverse también de alguna manera.

Hay dos tipos de técnicas de planificación dinámica, las centralizadas y las distribuidas, según se encargue de realizar el reordenamiento una unidad hardware especializada o se distribuya la responsabilidad de la planificación entre diferentes módulos. Es decir, se distinguen dos tipos de planificación según la ventana de instrucciones esté centralizada o distribuida. En cualquier caso, la planificación de instrucciones aritmético-lógicas suele realizarse independientemente de la planificación de instrucciones de acceso a memoria, ya que para estas últimas, los algoritmos de planificación tienen en cuenta que la jerarquía de memoria no es ideal, que la traducción de dirección virtual a física puede ser una etapa costosa, etc.

### 3.1.1. Planificación de instrucciones de acceso a memoria

Las instrucciones de acceso a memoria siempre implican tres etapas una vez que se han decodificado y leído sus operandos:

- Cálculo de la dirección de acceso a memoria.
- Traducción de la dirección virtual obtenida a dirección física.
- Acceso a la memoria de datos.

El cálculo de la dirección de acceso a memoria suele implicar la realización de una suma de enteros ya que el modo de direccionamiento más común es el indirecto con desplazamiento.

En cuanto a la traducción de dirección virtual a dirección física, se han estudiado las diferentes alternativas que existen en el capítulo 2. Y en este mismo capítulo también se han introducido todos los conceptos necesarios para comprender cómo se realiza el acceso a la memoria de datos, que en el mejor

de los casos puede ser un acierto en la caché de datos de nivel 1 y en el peor de los casos puede implicar un acceso a memoria principal (el acceso a memoria virtual ya implica un cambio de contexto).

Por lo tanto, de la misma forma que se bifurcan las instrucciones de coma flotante hacia unidades funcionales diferentes durante la etapa de ejecución en los procesadores segmentados multifuncionales, las instrucciones de acceso a memoria se pueden bifurcar también por otro camino siempre y cuando se añada una nueva ALU o sumador de enteros a la ruta de datos para que puedan realizar el cálculo de su dirección efectiva (figura 3.1).

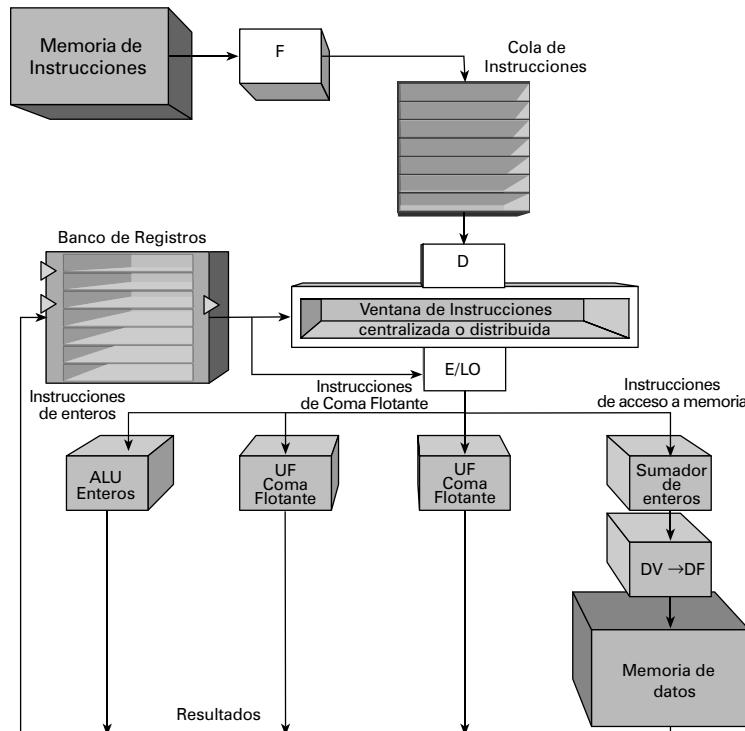


FIGURA 3.1

Ruta de datos de un procesador segmentado multifuncional con planificación dinámica de instrucciones.

La etapa X de estas instrucciones durará más o menos ciclos dependiendo de los aciertos y fallos que se produzcan en la traducción de dirección virtual a física y en el acceso a la memoria de datos.

A la hora de decidir si es posible planificar las instrucciones de acceso a memoria, desordenando el orden secuencial en el que aparecen en los programas, lo habitual es no permitirlo. Es decir, casi todas las arquitecturas con planificación dinámica obligan a mantener el orden de los accesos a memoria y de esta manera garantizan un modelo de consistencia de memoria fuerte. Esta decisión es fácil de justificar con la siguiente secuencia de instrucciones:

|     |            |
|-----|------------|
| S.D | F5,0(R10)  |
| L.D | F1,400(R7) |

Si se permite que la instrucción de carga se ejecute antes que la de almacenamiento, por ejemplo, porque el valor de F5 no está todavía disponible para que se pueda ejecutar el almacenamiento, puede ocurrir que al calcular la dirección de acceso a memoria de este almacenamiento coincida con la de la carga, y haya un riesgo RAW no resuelto correctamente. Al contrario que en el capítulo 1, la situación en

la que  $0+[R10]=400+[R7]$  puede generar un riesgo RAW a través de la memoria (ya que en un procesador más complejo no todas las instrucciones pasan por las mismas etapas y de forma ordenada), por ello se suele obligar a que las instrucciones de acceso a memoria se ejecuten en orden.

Por otro lado, si las direcciones de acceso a memoria son diferentes y este riesgo no existe, poder avanzar en la ejecución de la lectura de memoria aunque el almacenamiento esté parado, por ejemplo, por un fallo del TLB en la traducción de dirección virtual a física o por un fallo de la caché de datos, mejoraría bastante las prestaciones del procesador.

La ejecución de los accesos a memoria en desorden sólo suele permitirse cuando el procesador incorpora mecanismos para deshacer lo que se ha ejecutado incorrectamente. Como se verá un poco más adelante, esto ocurre en procesadores que permiten la especulación, pero no en los que solamente incorporan técnicas de planificación dinámica.

### 3.1.2. Planificación dinámica de instrucciones centralizada

En este tipo de técnica de planificación se incluye en el procesador una estructura hardware dedicada específicamente a controlar la planificación dinámica de instrucciones. Esta unidad suele denominarse ventana de instrucciones o pizarra (scoreboard).

La ventana de instrucciones o pizarra permite que las instrucciones se ejecuten fuera de orden siempre que tengan sus operandos disponibles y recursos suficientes para su ejecución. Se encarga de planificar dinámicamente el orden en el que deben ejecutarse las instrucciones resolviendo todos los riesgos (RAW, WAW y WAR) e intentando minimizar el número de paradas en el procesador.

En el nanoMIPS se utiliza una estructura denominada Pizarra y todas las instrucciones pasan por cuatro etapas una vez que se retiran de la cola de instrucciones (figura 3.2):

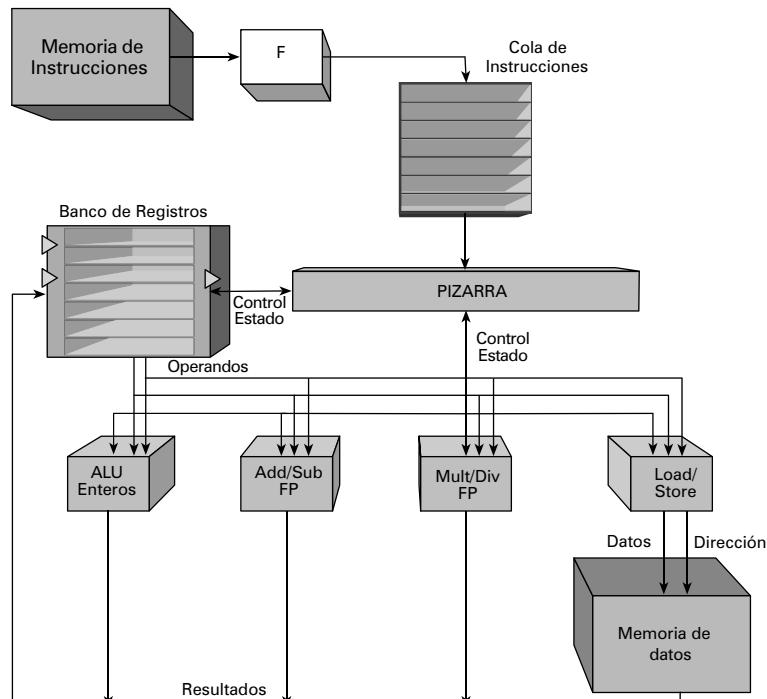


FIGURA 3.2

Ruta de datos del nanoMIPS segmentado multifuncional con planificación basada en Pizarra.

1. **Decodificación/Emisión (E).** Esta es la última etapa que debe ejecutarse en orden. La instrucción se decodifica y se emite si:

- Hay al menos una entrada libre en la Pizarra.
- La unidad funcional que necesita la instrucción para ejecutarse está libre.
- Ninguna instrucción activa tiene como registro destino el mismo que ella (riesgo WAW).

De esta manera se resuelven dinámicamente los riesgos WAW y queda reservada la unidad funcional que la instrucción va a utilizar.

2. **Lectura de operandos (LO).** La Pizarra comprueba la disponibilidad de los operandos que necesita la instrucción teniendo en cuenta que un operando está disponible si ninguna instrucción que se haya emitido anteriormente tiene que escribir este operando. Si todos los operandos están disponibles, la Pizarra ordena a la unidad funcional que lea los operandos y que comience la ejecución. Por lo tanto los riesgos RAW se resuelven dinámicamente en esta fase.
3. **Ejecución (X).** En la unidad funcional adecuada para la instrucción y en tantos ciclos como sea necesario para que la operación se complete. La unidad notifica a la Pizarra que tiene el resultado listo cuando termina.
4. **Escritura de resultados (W).** La Pizarra comprueba si existen riesgos WAR. Si no existen, deja que se escriba directamente el resultado producido por la unidad funcional en el registro adecuado.

Cuando se utiliza planificación dinámica en el nanoMIPS, se dejan de utilizar algunas de las mejoras que se introdujeron en el diseño de la ruta de datos para dar soporte a la segmentación y que ya no son tan necesarias o complicarían demasiado el diseño del planificador. Por ejemplo, en cada ciclo de reloj se puede leer o escribir en el banco de registros, pero ya no se establecen turnos, y los adelantamientos no se utilizan, todos los riesgos se resuelven mediante paradas.

La comprobación de riesgos RAW y WAR se hace en las etapas de lectura de operandos y escritura de resultados respectivamente. La resolución de estos riesgos se realiza mediante paradas en la propia instrucción, pero estas paradas no tienen por qué afectar a otras instrucciones. El peor caso relacionado con estos riesgos sería que una instrucción parada en la etapa LO porque alguno de sus operandos no está disponible hubiera reservado en su emisión una unidad funcional que pudiera estar siendo utilizada por otra instrucción posterior que sí tuviera disponibles sus operandos.

Por otro lado, la comprobación de riesgos WAW se realiza en la etapa de emisión de instrucciones y este tipo de riesgo también se resuelve mediante paradas. Pero las paradas que se realizan para resolver este tipo de riesgo sí que afectan a las instrucciones posteriores, ya que la emisión de instrucciones se debe realizar en orden.

Para implementar dinámicamente esta planificación, el nanoMIPS incluye en la pizarra tres estructuras internas de datos (figura 3.3):

- **Estado de las instrucciones.** Cada instrucción mantiene una entrada actualizada en esta estructura con información acerca de la operación que debe realizar, la unidad funcional que ha reservado y la última etapa de ejecución que ha completado.
- **Estado de las unidades funcionales.** Cada unidad funcional mantiene una entrada actualizada en esta estructura. Estas entradas indican si la unidad está ocupada, la operación que se está realizando y cuáles son los registros fuente (o la unidad funcional que va a escribir en ellos) y registro destino.
- **Estado del banco de registros.** Cada registro mantiene una entrada actualizada en la que almacena la unidad funcional que tiene que escribir su resultado en él.

### 3. TÉCNICAS DE AUMENTO DE PRESTACIONES PARA PROCESADORES

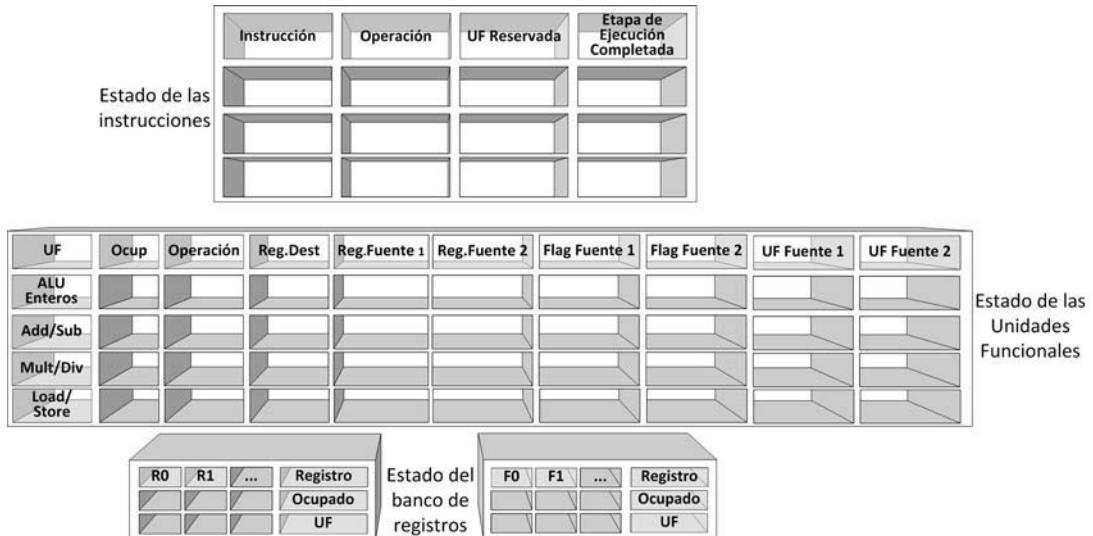


FIGURA 3.3

Implementación de la Pizarra en el nanoMIPS.

#### Ejemplo 3.1

Planificación dinámica de instrucciones con el algoritmo de la Pizarra en el nanoMIPS (sin riesgos WAR o WAW).

Supongamos que se ejecuta el siguiente fragmento de código en el nanoMIPS que utiliza planificación dinámica basada en la Pizarra con tres ALUs de enteros con latencia 1 ciclo, tres sumadores/restadores en coma flotante con latencia 4 ciclos y 2 multiplicadores/divisores en coma flotante con latencia de 8 ciclos para la multiplicación y 10 para la división. Ninguna de las unidades en coma flotante está segmentada.

Además, se supone que las instrucciones de carga y almacenamiento siguen el mismo camino que las instrucciones de enteros, utilizando las mismas ALUs y con 1 ciclo de latencia.

| Instrucción       | D/E              | LO                | X     | W                |
|-------------------|------------------|-------------------|-------|------------------|
| L.D F1,0(R0)      | 1                | 2                 | 3     | 4                |
| L.D F2,100(R0)    | 2                | 3                 | 4     | 5                |
| ADD.D F3,F1,F2    | 3                | 6 <sup>RAW</sup>  | 7-10  | 11               |
| SUB.D F4,F1,F2    | 4                | 7 <sup>RE</sup>   | 8-11  | 12               |
| MUL.D F5,F3,F4    | 5                | 13 <sup>RAW</sup> | 14-21 | 22               |
| DIV.D F6,F10,F1   | 6                | 8 <sup>RE</sup>   | 9-18  | 19               |
| S.D F5,200(R0)    | 7                | 23 <sup>RAW</sup> | 24    | 25               |
| ADD.D F8,F6,F12   | 8                | 20 <sup>RAW</sup> | 21-24 | 26 <sup>RE</sup> |
| MUL.D F13,F14,F15 | 19 <sup>RE</sup> | 21 <sup>RE</sup>  | 22-29 | 30               |
| S.D F8,400(R0)    | 20               | 27 <sup>RAW</sup> | 28    | 29               |

Este sería el diagrama temporal de ejecución de las instrucciones sin tener en cuenta la etapa de búsqueda de instrucción que las deja almacenadas en la cola de instrucciones. Se puede observar que:

- La emisión de instrucciones siempre se realiza en orden, el resto de etapas ya pueden desordenarse.
- En este primer ejemplo sólo se han producido riesgos de datos RAW y riesgos estructurales (RE), o bien por el banco de registros (en las etapas LO y W) o bien por las unidades funcionales (en el ciclo 9 la multiplicación no se puede emitir porque no tiene libre la unidad funcional que necesita, se emite en el ciclo 19 que es el primer ciclo en el que está disponible un multiplicador/divisor en coma flotante).
- La parada en la emisión de una instrucción afecta a todas las instrucciones posteriores porque la emisión debe mantenerse en orden.
- Todos los riesgos se han resuelto con paradas.

Y si por ejemplo, nos preguntamos por el estado de las estructuras internas de la Pizarra al comenzar el ciclo 19, tendríamos:

| Instrucción |            | Operación      | UF reservada | Etapa de ejecución completada |
|-------------|------------|----------------|--------------|-------------------------------|
| MUL.D       | F5,F3,F4   | Mult           | Mult/Div 1   | LO                            |
| DIV.D       | F6,F10,F1  | Div            | Mult/Div 2   | X                             |
| S.D         | F5,200(R0) | Almacenamiento | Ent 1        | D/E                           |
| ADD.D       | F8,F6,F12  | Add            | Add/Sub 3    | D/E                           |

| UF         | Ocup | Op   | Rd | Rs1 | Rs2 | Flgs1 | Flgs2 | UFs1       | UFs2 |
|------------|------|------|----|-----|-----|-------|-------|------------|------|
| Ent 1      | Sí   | Alm  | —  | F5  | R0  | NO    | Sí    | Mult/Div 1 | —    |
| Ent 2      |      |      |    |     |     |       |       |            |      |
| Ent 3      |      |      |    |     |     |       |       |            |      |
| Add/Sub 1  |      |      |    |     |     |       |       |            |      |
| Add/Sub 2  |      |      |    |     |     |       |       |            |      |
| Add/Sub 3  | Sí   | Add  | F8 | F6  | F12 | NO    | Sí    | Mult/Div 2 | —    |
| Mult/Div 1 | Sí   | Mult | F5 | F3  | F4  | Sí    | Sí    | —          | —    |
| Mult/Div 2 | Sí   | Div  | F6 | F10 | F1  | Sí    | Sí    | —          | —    |

|     |            |            |    |           |     |  |  |  |      |
|-----|------------|------------|----|-----------|-----|--|--|--|------|
| ... | F5         | F6         | F7 | F8        | ... |  |  |  | Reg  |
|     | Sí         | Sí         |    | Sí        |     |  |  |  | Ocup |
|     | Mult/Div 1 | Mult/Div 2 |    | Add/Sub 3 |     |  |  |  | UF   |

**Ejemplo 3.2**

Planificación dinámica de instrucciones con el algoritmo de la Pizarra en el nanoMIPS  
(con todo tipo de riesgos).

Supongamos que se ejecuta el siguiente fragmento de código en el mismo nanoMIPS del ejemplo anterior.

| Instrucción    | D/E               | LO                | X     | W                 |
|----------------|-------------------|-------------------|-------|-------------------|
| L.D F1,0(R0)   | 1                 | 2                 | 3     | 4                 |
| L.D F2,100(R0) | 2                 | 3                 | 4     | 5                 |
| DIV.D F3,F1,F2 | 3                 | 6 <sup>RAW</sup>  | 7-16  | 17                |
| ADD.D F4,F1,F3 | 4                 | 18 <sup>RAW</sup> | 19-22 | 23                |
| SUB.D F1,F1,F2 | 5                 | 7 <sup>RE</sup>   | 8-11  | 19 <sup>WAR</sup> |
| MUL.D F1,F6,F7 | 20 <sup>WAW</sup> | 21                | 22-29 | 30                |
| ADD.D F8,F4,F3 | 21                | 24 <sup>RAW</sup> | 25-28 | 29                |
| DIV.D F3,F6,F1 | 22                | 31 <sup>RAW</sup> | 32-41 | 42                |
| S.D F3,200(R0) | 23                | 43 <sup>RAW</sup> | 44    | 45                |
| S.D F8,64(R0)  | 24                | 30 <sup>RAW</sup> | 31    | 32                |

Este sería el diagrama temporal de ejecución de las instrucciones sin tener en cuenta la etapa de búsqueda de instrucción que las deja almacenadas en la cola de instrucciones. Se puede observar que:

- La emisión de instrucciones siempre se realiza en orden, el resto de etapas ya pueden desordenarse.
- En este ejemplo se producen todo tipo de riesgos, todos ellos se resuelven con paradas. Se produce un riesgo WAR por F1 entre las instrucciones ADD.D y SUB.D. Por eso la instrucción de resta no puede escribir su resultado en el registro F1 hasta el ciclo 19, una vez que la instrucción de suma haya leído sus operandos en el ciclo 18. En el caso del riesgo WAW que se produce por F1 entre la instrucción SUB.D y la MUL.D, no puede emitirse la segunda instrucción hasta que la primera no ha finalizado y ha escrito su resultado en el registro F1. Como esto ocurre en el ciclo 19 la multiplicación no puede emitirse hasta el ciclo 20.
- La parada en la emisión de una instrucción afecta a todas las instrucciones posteriores porque la emisión debe mantenerse en orden.

En general, este tipo de técnicas de planificación se ven muy limitadas por los siguientes factores:

- La necesidad de resolver los riesgos WAR y WAW mediante paradas.
- El número y el tipo de unidades funcionales disponibles en la ruta de datos.
- El número de entradas en la ventana de instrucciones o pizarra, que determina cuántas instrucciones se pueden evaluar para encontrar las secuencias adecuadas de instrucciones independientes que se puedan planificar y ejecutar aprovechando los recursos disponibles en el procesador.

Por eso la mayor parte de los procesadores actuales utilizan técnicas distribuidas como las que se presentan en la sección siguiente.

### 3.1.3. Planificación dinámica de instrucciones distribuida

En este tipo de planificación no existe una unidad hardware centralizada que se encargue de la ordenación dinámica de instrucciones. La ventana de instrucciones se distribuye entre varias estructuras hardware denominadas consignas o estaciones de reserva.

Cada unidad funcional puede tener su propia estación de reserva o se puede diseñar la ruta de datos de manera que varias estaciones de reserva comparten la misma unidad funcional, organizando así las estaciones de reserva según el tipo de instrucción que pueden ejecutar: enteros, suma/resta de coma flotante, multiplicación/división de coma flotante, etc.

En este caso las instrucciones se almacenan decodificadas en la cola de instrucciones de manera que las etapas F y D son etapas preliminares a la planificación. La última etapa que se realiza en orden es, de nuevo, la que se denomina de emisión (E), en la que se retiran las instrucciones en orden de la cola y se envían a la estación de reserva adecuada, si hay una disponible.

Si la única diferencia entre la planificación dinámica de instrucciones centralizada y distribuida fuera la ubicación de la ventana de instrucciones, el rendimiento de ambos tipos de planificación sería muy similar.

Sin embargo:

- Las estaciones de reserva suelen incluir espacio para almacenar los operandos que están disponibles cuando se emite la instrucción, esto evita los riesgos WAR y las paradas asociadas a ellos.
- Las estaciones de reserva pueden conectarse directamente al bus que recoge los resultados de las unidades funcionales, normalmente denominado CDB o Common Data Bus. Esto implementa adelantamiento para resolver los riesgos de datos RAW, evitando así algunas paradas.
- El Common Data Bus lleva asociado un sencillo hardware de control que ordena las escrituras que se realizan definitivamente en el banco de registros, por lo que evita los riesgos WAW y las paradas asociadas a ellos.

Y aquí en estos tres factores se encuentra la mejora sustancial respecto a los algoritmos centralizados, la implementación por defecto del adelantamiento para resolver los riesgos RAW y el uso de renombramiento dinámico de registros para evitar los riesgos WAW y WAR.

Hay que tener en cuenta que los riesgos WAR y WAW no son dependencias de datos reales. Si los registros del procesador nunca se reutilizan para almacenar diferentes variables nunca se producirían estas dependencias de datos falsas. Como sí existe reutilización, la manera de evitar estos riesgos es utilizar renombramiento de registros.

Si el renombramiento se realiza de manera estática, es el compilador el que se encarga de gestionar la utilización y reutilización de los registros para evitar los riesgos. Pero en el caso del renombramiento dinámico, se suelen utilizar registros adicionales en el procesador, no visibles para el programador, que puedan servir como almacenamiento temporal. En muchos casos estos registros se agrupan en un buffer de renombramiento, pero cuando se realiza planificación dinámica con estaciones de reserva, estos registros adicionales están ubicados en las propias estaciones de reserva. De esta manera las estaciones de reserva buscan y almacenan los operandos que necesitan las diferentes instrucciones de manera que no sea necesario leerlos directamente de los registros. Esta técnica junto con el control de escrituras, consigue evitar los riesgos WAR y WAW.

Cuando el nanoMIPS implementa planificación dinámica de instrucciones distribuida lo hace con el algoritmo de Tomasulo. En este algoritmo las instrucciones pasan por tres etapas una vez que se retiran de la cola de instrucciones (figura 3.4):

1. **Emisión (E).** Se retira la instrucción que está en la parte superior de la cola de instrucciones. Si hay alguna estación de reserva libre adecuada para esta instrucción, se envía la instrucción junto con el valor de los operandos que ya estén disponibles. Si no hay estaciones de reserva vacías hay que parar la emisión de instrucciones hasta que quede libre alguna, ya que la emisión es todavía en orden. Por el contrario, si algún operando no está disponible, la instrucción se emite y se queda a la espera en su estación de reserva. Para ello se registra en la estación de reserva qué instrucción es la que va a producir este operando como resultado. Este registro se realiza asociando etiquetas a las diferentes estaciones de reserva.

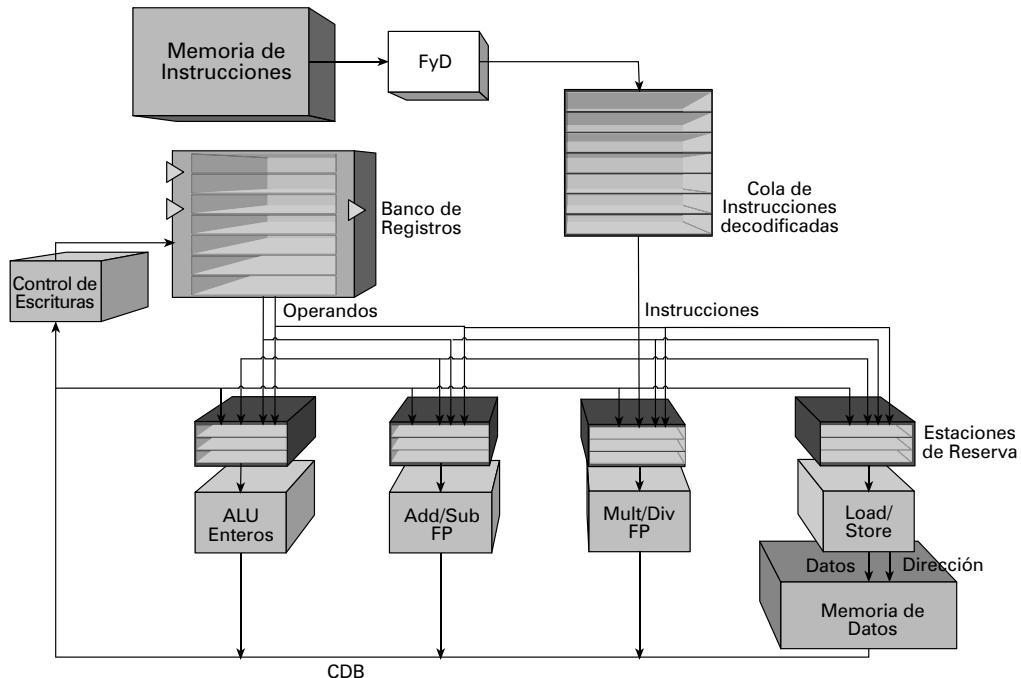


FIGURA 3.4

Ruta de datos del nanoMIPS segmentado multifuncional con planificación Tomasulo  
(varias estaciones de reserva comparten una única unidad funcional).

2. **Ejecución (X).** Una instrucción se envía de la estación de reserva a la unidad funcional cuando ésta está libre y la instrucción ya tiene en la estación de reserva sus operandos. Las instrucciones monitorizan el CDB hasta que todos sus operandos estén disponibles. Cuando se cumpla esta condición, se puede pasar a ejecutar la instrucción, de esta manera se solucionan los riesgos RAW. Si no está libre la unidad funcional que se requiere, habrá que esperar hasta que se resuelva el riesgo estructural. Si más de una instrucción puede enviarse a la unidad funcional en el mismo ciclo, se envía siempre la que lleva más tiempo esperando en la estación de reserva.
3. **Escritura de resultado (W).** Se escriben los resultados de las operaciones junto con las etiquetas de las estaciones de reserva origen de las instrucciones en el CDB para que todas las estaciones de reserva que estaban esperando por ese resultado lo puedan leer. La estructura de control de escrituras determina cuáles de los resultados tienen que escribirse finalmente en los registros.

Por lo tanto con este algoritmo se mantienen dos tipos de estructuras de planificación (figura 3.5):

- **Estaciones de reserva.** Cada instrucción mantiene una entrada actualizada en una estación de reserva con información acerca de sus operandos fuente (la etiqueta asociada si todavía no están disponibles) y sus valores.
- **Control de escrituras.** Se mantiene información actualizada sobre la etiqueta de la instrucción que va a producir el resultado que debe escribirse en cada registro. Aunque en este libro se muestra como una estructura independiente, en muchos casos se implementa dentro del propio banco de registros.

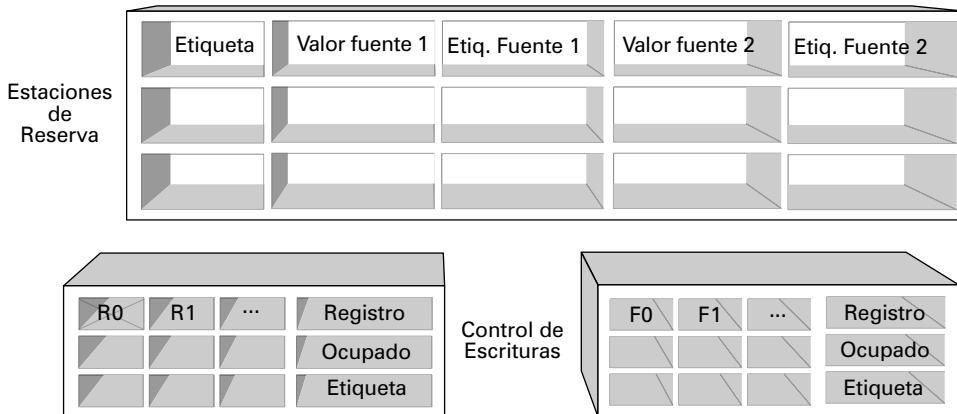


FIGURA 3.5

Implementación de las estaciones de reserva y el control de escrituras en el nanoMIPS.

Si se comparan las dos alternativas de planificación en el nanoMIPS, las diferencias fundamentales entre el método de la Pizarra y el algoritmo de Tomasulo son principalmente:

- La lógica de detección y solución de riesgos está centralizada en la Pizarra en el primer caso, mientras que en el segundo está distribuida en las estaciones de reserva y por lo tanto, es especializada. Esto hace que la Pizarra consiga un mejor aprovechamiento global de todos los recursos disponibles.
- Con el método de la Pizarra se resuelven los riesgos WAR y WAW mediante paradas. Con Tomasulo se resuelven almacenando directamente los valores de los operandos en las estaciones de reserva y utilizando un hardware de control de escrituras (renombramiento dinámico de registros).
- En el caso de Tomasulo, el CDB permite que las instrucciones que están esperando por resultados de otras instrucciones monitoricen el bus y los puedan leer en cuanto estén disponibles. Es decir, se implementa directamente el adelantamiento. En el caso de la Pizarra, los operandos se tienen que leer de los registros o implementar el adelantamiento explícitamente.
- En la Pizarra la escritura de resultados se hace directamente sobre el banco de registros mientras que con Tomasulo se hace primero en el CDB y luego se determina qué escrituras deben hacerse efectivas en el banco de registros.

### Ejemplo 3.3

Planificación dinámica de instrucciones con el algoritmo de Tomasulo en el nanoMIPS con una unidad funcional para cada estación de reserva.

Supongamos que se ejecuta el mismo fragmento de código que en el ejemplo 3.2 en el nanoMIPS que utiliza Tomasulo con tres ALUs de enteros con latencia 1 ciclo, tres sumadores/restadores en coma flotante con latencia 4 ciclos y 2 multiplicadores/divisores en coma flotante con latencia de 8 ciclos para la multiplicación y 10 para la división. Ninguna de las unidades en coma flotante está segmentada y se tiene una unidad funcional para cada estación de reserva. Las estaciones de reserva se etiquetan como 1, 2 y 3 las de enteros, 4, 5 y 6 las de suma/resta en coma flotante y 7 y 8 las de multiplicación y división en coma flotante.

Además, se supone de nuevo que las instrucciones de carga y almacenamiento siguen el mismo camino que las instrucciones de enteros, utilizando las mismas ALUs y con 1 ciclo de latencia.

| Instrucción    | E                | X                     | W                |
|----------------|------------------|-----------------------|------------------|
| L.D F1,0(R0)   | 1                | 2                     | 3                |
| L.D F2,100(R0) | 2                | 3                     | 4                |
| DIV.D F3,F1,F2 | 3                | 5 <sup>RAW</sup> -14  | 15               |
| ADD.D F4,F1,F3 | 4                | 16 <sup>RAW</sup> -19 | 20               |
| SUB.D F1,F1,F2 | 5                | 6-9                   | 10               |
| MUL.D F1,F6,F7 | 6                | 7-14                  | 16 <sup>RE</sup> |
| ADD.D F8,F4,F3 | 7                | 21 <sup>RAW</sup> -24 | 25               |
| DIV.D F3,F6,F1 | 16 <sup>RE</sup> | 17-26                 | 27               |
| S.D F3,200(R0) | 17               | 28 <sup>RAW</sup>     | 29               |
| S.D F8,64(R0)  | 18               | 26 <sup>RAW</sup>     | 28 <sup>RE</sup> |

Este sería el diagrama temporal de ejecución de las instrucciones sin tener en cuenta la etapa de búsqueda que las deja almacenadas en la cola de instrucciones. Se puede observar que:

- La emisión de instrucciones siempre se realiza en orden, el resto de etapas ya pueden desordenarse.
- Aunque en este ejemplo se producen todo tipo de riesgos, sólo los estructurales y los RAW se resuelven con paradas. Se produce un riesgo WAR por F1 entre las instrucciones ADD.D y SUB.D, pero queda resuelto gracias al almacenamiento de los valores de los operandos en las estaciones de reserva. Cuando la instrucción de suma se emite en el ciclo 4, el valor que en ese momento hay almacenado en F1 se guarda junto con la instrucción en su estación de reserva. Por lo tanto, no hay ningún problema porque una instrucción posterior escriba en este registro, la instrucción pasará a ejecutarse siempre con el valor adecuado de su operando. En cuanto al riesgo WAW entre las instrucciones de resta y multiplicación, tampoco se resuelve mediante paradas. Se deja que la instrucción de multiplicación avance y escriba su resultado en el CDB. En este caso, los resultados de las instrucciones se escriben en el orden correcto, pero si no hubiera sido así, el control de escrituras se hubiera encargado de que la escritura que quedara en F1 fuera el resultado de la multiplicación, por lo que no es necesario introducir ninguna parada.
- La parada en la emisión de la instrucción de división hasta el ciclo 16 por falta de estación de reserva afecta a todas las instrucciones posteriores porque la emisión debe mantenerse en orden.

Y si por ejemplo, nos preguntamos por el estado de las estaciones de reserva y del control de escrituras al comenzar el ciclo 19, tendríamos:

| Etiqueta | Valor s1 | Etiqueta s1 | Valor s2       | Etiqueta s2 |
|----------|----------|-------------|----------------|-------------|
| 1        | —        | 7           | [R0]           |             |
| 2        | —        | 6           | [R0]           |             |
| 3        |          |             |                |             |
| 4        | [F1]     |             | [CDB] ciclo 15 |             |
| 5        |          |             |                |             |
| 6        | —        | 4           | [CDB] ciclo 15 |             |
| 7        | [F6]     |             | [CDB] ciclo 16 |             |
| 8        |          |             |                |             |

| F3 | F4 | ... | F8 | ... |  |  |  |  | Reg      |
|----|----|-----|----|-----|--|--|--|--|----------|
| Sí | Sí |     | Sí |     |  |  |  |  | Ocup     |
| 7  | 4  |     | 6  |     |  |  |  |  | Etiqueta |

**Ejemplo 3.4****Planificación dinámica de instrucciones con el algoritmo de la Tomasulo en el nanoMIPS con varias estaciones de reserva compartiendo una única unidad funcional.**

Supongamos que se ejecuta el mismo fragmento de código que en el ejemplo anterior pero con una única unidad funcional de cada tipo, ya que en el ejemplo anterior se puede observar que las unidades funcionales están desocupadas la mayor parte del tiempo. De esta manera se tienen tres estaciones de reserva para operaciones de enteros, tres para suma y resta en coma flotante y dos para multiplicación y división en coma flotante, pero cada grupo de estaciones comparte una única unidad funcional de cada tipo.

La principal diferencia con el ejemplo anterior es que una instrucción en una estación de reserva con sus operandos disponibles, tendrá que comprobar la disponibilidad de la unidad funcional antes de pasar a la etapa X.

El diagrama temporal de ejecución en este caso es el siguiente:

| Instrucción    | E                | X                     | W  |
|----------------|------------------|-----------------------|----|
| L.D F1,0(R0)   | 1                | 2                     | 3  |
| L.D F2,100(R0) | 2                | 3                     | 4  |
| DIV.D F3,F1,F2 | 3                | 5 <sup>RAW</sup> -14  | 15 |
| ADD.D F4,F1,F3 | 4                | 16 <sup>RAW</sup> -19 | 20 |
| SUB.D F1,F1,F2 | 5                | 6-9                   | 10 |
| MUL.D F1,F6,F7 | 6                | 15 <sup>RE</sup> -22  | 23 |
| ADD.D F8,F4,F3 | 7                | 21 <sup>RAW</sup> -24 | 25 |
| DIV.D F3,F6,F1 | 16 <sup>RE</sup> | 24 <sup>RAW</sup> -33 | 34 |
| S.D F3,200(R0) | 17               | 35 <sup>RAW</sup>     | 36 |
| S.D F8,64(R0)  | 18               | 26 <sup>RAW</sup>     | 27 |

Es decir, casi no se nota la diferencia a pesar de haber reducido tanto los recursos de ejecución en la ruta de datos, sólo ha habido que parar la instrucción de multiplicación que no puede pasar a utilizar la unidad funcional de multiplicación/división en coma flotante hasta que la primera división no termina de ejecutar y por lo tanto no comienza su ejecución hasta el ciclo 15.

Si además se segmentaran las unidades funcionales, las paradas por conflicto en la utilización de una de ellas se reducirían a un ciclo en la mayor parte de los casos. Esta segmentación es muy habitual cuando las unidades funcionales son compartidas por varias estaciones de reserva.

## 3.2 Predicción dinámica de saltos

En la sección anterior se han presentado técnicas que permiten reordenar las instrucciones en tiempo de ejecución para evitar en todo lo posible las paradas provocadas por los riesgos de datos RAW. Pero con las técnicas propuestas, ninguna instrucción puede empezar su etapa de ejecución (la primera que modifica realmente el estado del procesador) hasta que no se resuelvan todas las instrucciones de salto que le preceden. Y los riesgos de control son muy frecuentes, por lo que es necesario disponer también de técnicas dinámicas que permitan avanzar con la ejecución de instrucciones a pesar de estos riesgos de control o las técnicas de planificación estarán tremadamente limitadas.

Todas estas técnicas se basan en predecir lo que va a pasar con los saltos, pero no con predicciones estáticas como las que se presentaron en el capítulo 1, sino con predicciones dinámicas que puedan acertar

sus predicciones en un porcentaje mayor, ya que la penalización por riesgo de control se reduce cuando la predicción se acierta. Algunos estudios demuestran que con predicciones estáticas se pueden acertar entre un 70 y un 80% de las predicciones de salto, mientras que los esquemas dinámicos incrementan los aciertos hasta el rango entre el 80 y el 95%.

En lo que se refiere a la predicción dinámica de saltos, hay dos tipos de decisiones que tomar. La primera, qué tipo de estructura hardware se utiliza para realizar este tipo de predicción. La segunda, qué tipo de predictor se utiliza para realizar las predicciones, es decir, qué técnica se utiliza para predecir si un salto se va a tomar o no y con qué conjunto de ceros y unos se va a almacenar esta predicción.

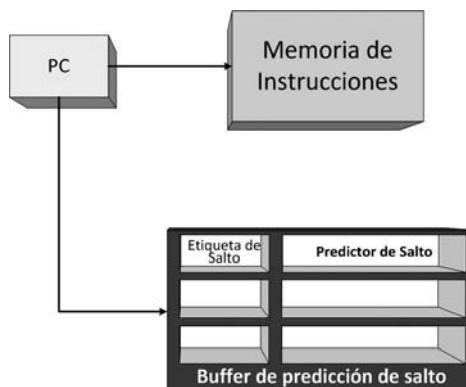
Sean cuales sean las estructuras y predictores escogidos, de nuevo como en el caso de las predicciones estáticas no se debe permitir que las instrucciones predichas modifiquen el contenido de la memoria y del banco de registros, a no ser que se incorporen en la ruta de datos mecanismos que puedan deshacer estas modificaciones si las predicciones resultan ser fallidas.

Y de la misma manera, cuanto antes se sepa que la instrucción es un salto, antes se podrá comenzar a realizar su tratamiento y evitar así el mayor número de paradas. Gracias a la predecodificación, se puede saber que una instrucción es un salto en la etapa F o en la cola de instrucciones, normalmente no es necesario esperar hasta la decodificación en D.

### 3.2.1. Estructuras hardware para la predicción de salto

La estructura hardware más sencilla asociada a la predicción dinámica de saltos es una pequeña caché a la que se accede en la fase F al mismo tiempo que se busca la instrucción, por si ésta fuera un salto, en la que se almacena un bloque de bits que permiten predecir el comportamiento del salto en su próxima ejecución (tomado o no tomado).

Esta memoria se denomina buffer de predicción o tabla de historia de saltos y contiene una serie de bits que resumen el comportamiento del salto en ejecuciones anteriores y que permiten predecir cuál va a ser su comportamiento la próxima vez que se ejecute (figura 3.6).



**FIGURA 3.6**

Buffer de predicción de salto.

Se indexa con la parte baja de la dirección de las instrucciones de salto, es decir, es una memoria de emplazamiento directo en la que a cada salto le corresponde siempre la misma entrada dependiendo de su dirección en la memoria de instrucciones. Como sólo se utiliza la parte baja de las direcciones cabe la posibilidad de que varios saltos se indexen en el buffer con la misma etiqueta, de manera que van sobrescribiendo sus predicciones. Este fenómeno suele denominarse aliasing y hace que un salto utilice una predicción que no es la suya.

De hecho, los fallos en las predicciones de salto pueden deberse, principalmente, a dos factores: a que algunos saltos son impredecibles (porque no se tiene información acerca de su historia reciente, porque los datos involucrados en la evaluación de la condición son intrínsecamente aleatorios, etc) y a los problemas de aliasing.

Para que el aliasing ocurra el menor número de veces posible, el tamaño del buffer de predicción debe ser el adecuado. Actualmente, con un buffer de 4096 entradas prácticamente nunca coinciden dos saltos en la misma entrada, evitando así los fallos que se deben a la utilización de la predicción de otro salto que tiene la misma terminación en su dirección. De esta manera los fallos de la predicción de saltos se deberán a que algunos saltos son difíciles de predecir, pero no al tamaño de la estructura de predicción.

Cuando se decodifica una instrucción y resulta ser un salto, se utilizan los bits de predicción que se leyeron del buffer durante la etapa F para realizar la predicción del salto. Si se predice como no tomado, se continúa con la búsqueda secuencial de instrucciones y si se predice como tomado, en cuanto se conoce la dirección destino de salto se pasa a buscar esa instrucción.

Como ocurría con las predicciones estáticas estudiadas en el capítulo 1, si se acierta la predicción, se reduce la penalización por riesgo de control. Sin embargo, si se falla se debe corregir el predictor en el buffer para la siguiente ejecución del salto y se sufre una penalización por riesgo de control igual o superior a la que se habría sufrido si no se hubiera realizado la predicción.

La mejora global que se obtendrá gracias al buffer de predicción dependerá de la frecuencia con la que aparezcan las instrucciones de salto, de la tasa de fallos de las predicciones y de la penalización que suponga cada fallo de predicción.

Sobre el primer factor no se puede influir desde el diseño del procesador, pero sí en los otros dos. Para reducir la tasa de fallos en todo lo posible se debe aumentar el tamaño del buffer de predicción (hasta las 4096 entradas, por encima de este valor no se notarán mejoras significativas) y utilizar los predictores de salto más adecuados, es decir, que fallen lo menos posible para los códigos y aplicaciones que se ejecutan sobre la arquitectura. En cuanto a la penalización que supone cada fallo de predicción, normalmente depende del diseño de la ruta de datos del procesador, pero también se puede reducir incorporando al buffer los suficientes puertos de lectura y escritura para poder corregir las predicciones que se han fallado sin necesidad de parar la ejecución de instrucciones en el procesador. Hay que tener en cuenta que si el buffer sólo tiene un puerto de acceso, en el ciclo en el que haya que hacer una corrección de un predictor que ha fallado su última predicción no se podrá comenzar una nueva instrucción ya que habrá riesgo estructural en la etapa F.

### Ejemplo 3.5

**Utilización de un buffer de predicción de saltos en un procesador segmentado de 6 etapas y comparación con el mejor esquema de predicción estática.**

Supongamos que tenemos un procesador segmentado en 6 etapas: F, D, IS, X, M y W. En este procesador las instrucciones de salto condicional calculan la dirección destino de salto en la etapa D y evalúan la condición y cargan el nuevo PC en la etapa X. Vamos a comparar el rendimiento de la predicción estática de salto (ejemplo 1.7 del capítulo 1) con un esquema de predicción dinámico basado en la utilización de un buffer de predicción de 4096 entradas (por lo que es muy improbable que se produzcan casos de aliasing de saltos).

En el caso de la utilización de un buffer de predicción, supongamos que la predicción se obtiene durante la etapa F (se accede al mismo tiempo al buffer de predicción y a la memoria de instrucciones) y que las predicciones de este buffer, muy sencillas, acierran el 80% de las ocasiones.

Tenemos:

|                         |   |   |    |    |   |   |
|-------------------------|---|---|----|----|---|---|
| Salto NT, predicción NT | F | D | IS | X  | M | W |
| PC+4                    |   | F | D  | IS |   |   |
| PC+8                    |   |   | F  | D  |   |   |
| PC+12                   |   |   |    | F  |   |   |
| ↓                       |   |   |    |    |   |   |
| Salto NT, predicción T  | F | D | IS | X  | M | W |
| Destino de salto        |   | — | F  | Ð  |   |   |
| Destino+4               |   |   |    | F  |   |   |
| PC+4                    |   |   |    |    | ↓ | F |
| ↓                       |   |   |    |    |   |   |
| Salto T, predicción T   | F | D | IS | X  | M | W |
| Destino de salto        |   | — | F  | D  |   |   |
| Destino+4               |   |   |    | F  |   |   |
| Destino+8               |   |   |    |    | ↓ |   |
| ↓                       |   |   |    |    |   |   |
| Salto T, predicción NT  | F | D | IS | X  | M | W |
| PC+4                    |   | F | Ð  | IS |   |   |
| PC+8                    |   |   | F  | Ð  |   |   |
| PC+12                   |   |   |    | F  | ↓ |   |
| Destino de salto        |   |   |    |    |   | F |

En resumen, tenemos las siguientes penalizaciones:

|                   |   |
|-------------------|---|
| Salto NT, acierto | 0 |
| Salto NT, fallo   | 3 |
| Salto T, acierto  | 1 |
| Salto T, fallo    | 3 |

Por lo que si el 75% de los saltos condicionales se toman y el 25% de los saltos no, tenemos que la penalización media por salto condicional es: %fallos · 3 + %saltos tomados · %aciertos · 1 = 0.2 · 3 + 0.75 · 0.8 · 1 = 1.2 ciclos, lo que mejora ligeramente el resultado que se obtenía con la predicción estática de salto tomado (una penalización media de 1.5 ciclos).

### Ejemplo 3.6

Ejemplo de las diferencias entre un buffer de predicción de saltos con un puerto de acceso y con dos.

En el ejemplo anterior no hemos tenido en cuenta en ningún momento el tiempo que lleva corregir las predicciones equivocadas en el buffer de predicción. Lo que se ha supuesto, implícitamente, es que el buffer tiene dos puertos de acceso. Uno de lectura, que permite obtener las predicciones de cada salto en la etapa F y otro de escritura que permite, al mismo tiempo, corregir las predicciones que se han comprobado erróneas.

Pero si no existen estos dos puertos de acceso y el buffer sólo dispone de uno, la penalización de los fallos aumenta en 1 ciclo, que debe invertirse en corregir la predicción en el buffer:

|                        |   |   |    |   |   |                       |
|------------------------|---|---|----|---|---|-----------------------|
| Salto NT, predicción T | F | D | IS | X | M | W                     |
| Destino de salto       |   | — | F  | Ð |   |                       |
| Destino+4              |   |   |    | F |   |                       |
| PC+4                   |   |   |    |   | ↓ | corrección del buffer |
|                        |   |   |    |   |   | F                     |

| Salto T, predicción NT | F | D | IS | X  | M                     | W |
|------------------------|---|---|----|----|-----------------------|---|
| PC+4                   |   | F | θ  | IS |                       |   |
| PC+8                   |   |   | F  | θ  |                       |   |
| PC+12                  |   |   |    | F  |                       |   |
| Destino de salto       |   |   |    |    | corrección del buffer | F |

En los dos casos en los que el buffer puede fallar es necesario detener la búsqueda de instrucciones durante un ciclo para poder acceder al buffer sin riesgo estructural y corregir la predicción equivocada.

En resumen, tenemos las siguientes penalizaciones:

|                   |   |
|-------------------|---|
| Salto NT, acierto | 0 |
| Salto NT, fallo   | 4 |
| Salto T, acierto  | 1 |
| Salto T, fallo    | 4 |

Por lo que ahora la penalización media por fallo es  $0.2 \cdot 4 + 0.75 \cdot 0.8 \cdot 1 = 1.4$  ciclos, prácticamente la misma que con una predicción estática de salto tomado.

En el caso concreto de la arquitectura que se está tomando como ejemplo en este libro, en el nanoMIPS, el buffer de predicción de saltos no resulta muy eficiente. Esto ocurre con todas las arquitecturas en las que se conoce la dirección destino de salto al mismo tiempo que el resultado de la evaluación de la condición.

Si el buffer de predicción predice que el salto no se toma, no hay ningún retraso puesto que al ciclo siguiente se busca la instrucción siguiente. Pero si el buffer de predicción predice que el salto se va a tomar, no se puede buscar la instrucción destino de salto hasta que no se haya calculado la dirección de esta instrucción. Y una vez que se conozca esta instrucción ya se conoce si el salto se toma realmente o no, así que no tiene sentido continuar utilizando la predicción (figura 3.7).

La solución está en implementar algún mecanismo que permita predecir también la dirección destino de salto. Para esto se utiliza otra estructura hardware denominada BTB (Branch Target Buffer) o buffer de predicción destino de salto. Esta estructura permite resolver el problema de la predicción dinámica de saltos para procesadores como el nanoMIPS, en los que se conoce al mismo tiempo la dirección destino de salto que el resultado de la evaluación de la condición y por lo tanto no es posible reducir las paradas por riesgo de control realizando la predicción de salto tomado.

El BTB es una caché que almacena la dirección destino que tuvo cada salto la última vez que se tomó (figura 3.8). El acceso al BTB se hace durante la etapa F de las instrucciones, al mismo tiempo que se busca la instrucción en la memoria de instrucciones, con su dirección se accede al BTB para saber si se trata de una instrucción de salto. El BTB se indexa con las direcciones completas porque en este caso es muy importante que no haya confusiones entre los diferentes saltos. En un buffer de predicción sólo hay dos opciones, predicción de salto tomado o no tomado, por lo que si dos saltos coinciden en la misma entrada del buffer, existe un 50% de probabilidad de que aún así se acierte la

### 3. TÉCNICAS DE AUMENTO DE PRESTACIONES PARA PROCESADORES

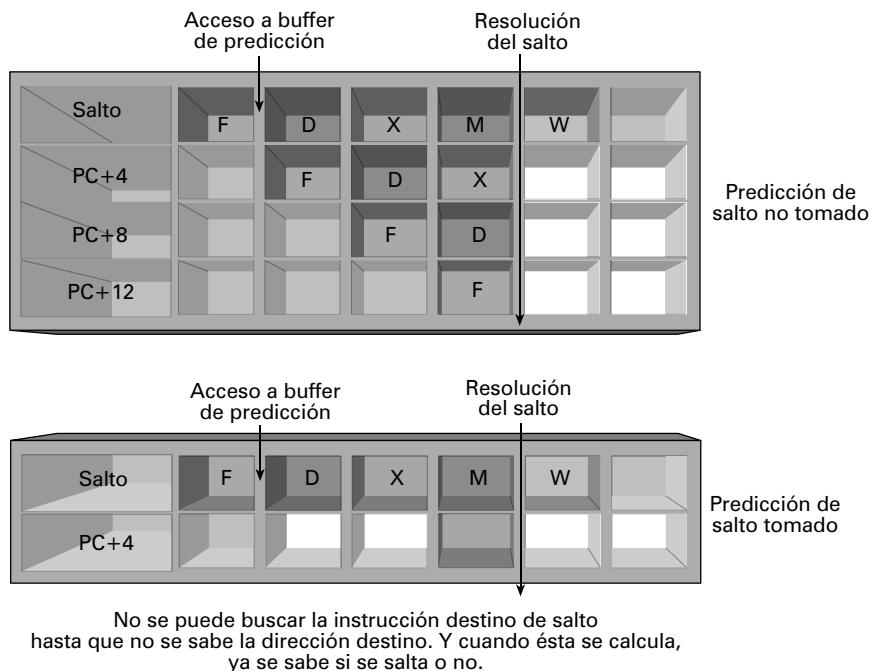


FIGURA 3.7

Utilización de un buffer de predicción de saltos en el nanoMIPS que resuelve los saltos en la etapa M.

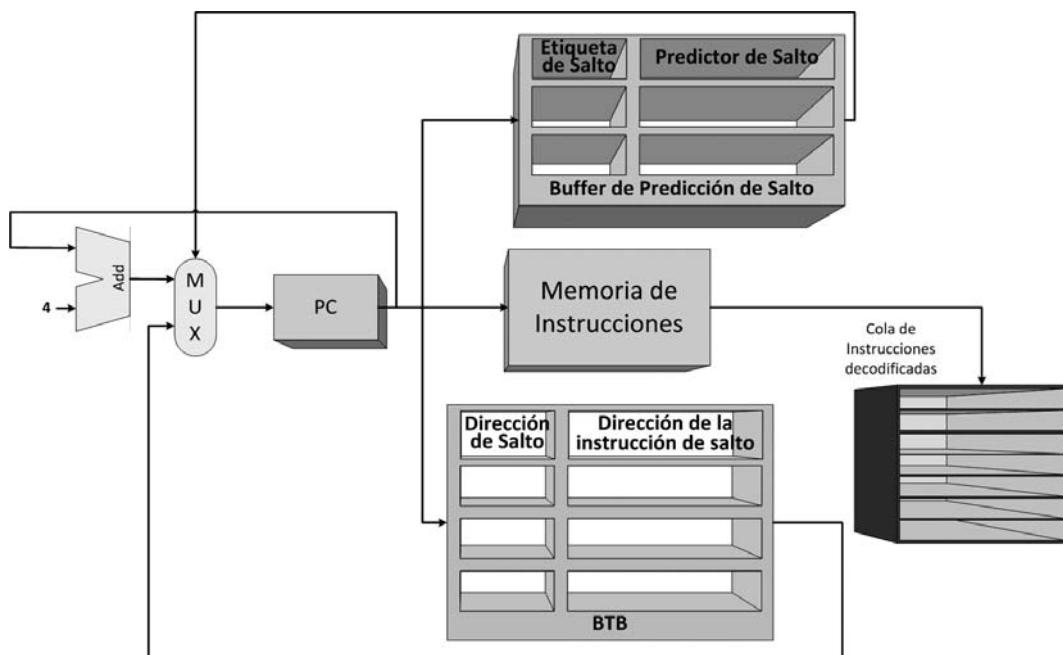


FIGURA 3.8

BTB (Branch Target Buffer).

predicción, pero la dirección destino de salto cambia con toda probabilidad de un salto a otro. Como se indexa con direcciones completas, se trata de una memoria con emplazamiento asociativo en la que hay que recorrer todas las entradas para encontrar la predicción del salto que se está buscando. Esto limita el tamaño del BTB (cuantas más entradas tenga, más comparaciones hay que realizar y más lenta es esta estructura).

Por lo tanto, para las instrucciones de salto se puede conocer:

- La predicción dinámica del comportamiento del salto en su próxima ejecución accediendo al buffer de predicción.
- La dirección de la instrucción siguiente (calculada en la etapa de F sumando 4 al PC) por si se predice que no se salta.
- La dirección de la instrucción destino de salto, que se ha obtenido en la etapa F accediendo al BTB, por si se predice que se salta.

Pero esto obliga a incluir en la ruta de datos, y a mantener actualizadas, dos estructuras diferentes de predicción.

La primera solución para evitar esto sería almacenar en el BTB sólo los saltos que se toman. Esto es lógico, porque para los saltos que no se toman no es necesario almacenar el destino, que es siempre la instrucción siguiente.

Entonces, al encontrar una instrucción de salto en el BTB, está implícita la predicción de que el salto se va a tomar. Esta predicción implícita es en realidad un predictor de 1 bit, ya que sólo hay dos alternativas posibles para la predicción. Si se salta se almacena el salto en el BTB y la próxima vez se predecirá el salto como tomado (como si el predictor valiera 1). Si no se salta no se almacena en el BTB y se predecirá el salto como no tomado (como si el predictor valiera 0).

Resumiendo, con un BTB de este tipo, los pasos que se siguen después de buscar una entrada con el PC de la instrucción actual en el BTB en la etapa F son:

- Si no se encuentra:
  - Si la instrucción que se ejecuta resulta ser un salto que se toma, se para la ejecución de la instrucción que se había buscado, se busca la instrucción adecuada y se actualiza el BTB incluyendo una nueva entrada.
  - Si no es un salto o es un salto que no se toma se continúa con la ejecución normalmente.
- Si se encuentra:
  - Si el salto se toma y con el destino predicho, se continúa la ejecución normalmente.
  - Si el salto se toma pero tiene otro destino, se para la ejecución de la instrucción que se había buscado, se busca la instrucción adecuada y se actualiza el BTB.
  - Si el salto no se toma, se para la ejecución de la instrucción que se había buscado, se busca la instrucción adecuada y se elimina la entrada del BTB

Como se verá en la sección siguiente, en muchos casos es necesario manejar predictores mucho más complejos que éste de 1 bit que se utiliza gracias a la predicción implícita del BTB. En estos casos, la alternativa para evitar mantener y actualizar dos estructuras diferentes es fundir ambas de manera que se puede almacenar el predictor de cada salto en el propio BTB. En este caso, se almacenan todos los saltos en el BTB, no sólo los que se toman (figura 3.9).

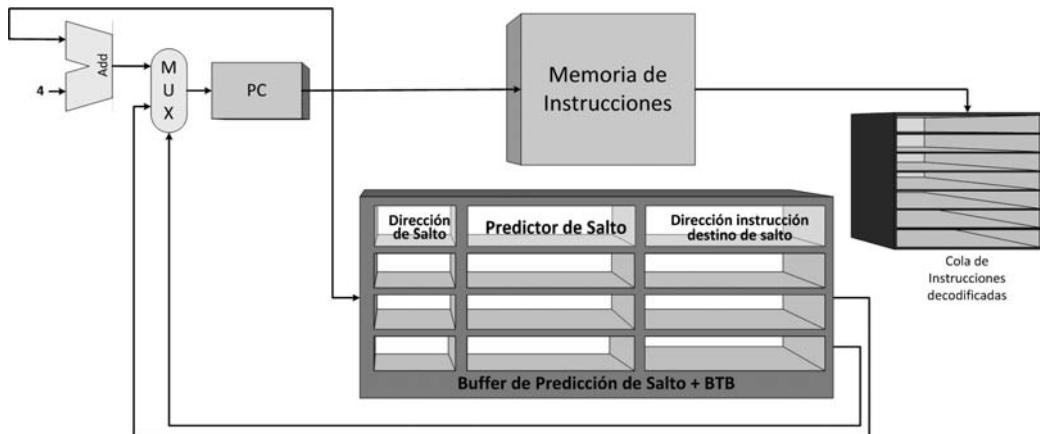


FIGURA 3.9

Buffer de predicción + BTB

**Ejemplo 3.7**

Utilización de un buffer de predicción de saltos junto con BTB en un procesador segmentado de 6 etapas

Supongamos que tenemos el mismo procesador segmentado en 6 etapas: F, D, IS, X, M y W que en el ejemplo 3.5.

En este caso la estructura de predicción es un buffer de predicción con un BTB incorporado y la predicción se obtiene durante la etapa F (se accede al mismo tiempo al buffer+BTB y a la memoria de instrucciones). Esta estructura de predicción tiene dos puertos de acceso y sus predicciones aciertan de nuevo el 80% de las ocasiones en lo que se refiere a si el salto se toma o no. Supondremos que la dirección destino de salto predicha por el BTB siempre es acertada. Tenemos:

| Salto NT, predicción NT | F | D | IS | X  | M | W |
|-------------------------|---|---|----|----|---|---|
| PC+4                    |   | F | D  | IS |   |   |
| PC+8                    |   |   | F  | D  |   |   |
| PC+12                   |   |   |    | F  |   |   |
|                         |   |   |    |    |   |   |
| Salto NT, predicción T  | F | D | IS | X  | M | W |
| Destino de salto        |   | F | D  | IS |   |   |
| Destino+4               |   |   | F  | D  |   |   |
| Destino+8               |   |   |    | F  |   |   |
| PC+4                    |   |   |    |    | F |   |
|                         |   |   |    |    |   |   |
| Salto T, predicción T   | F | D | IS | X  | M | W |
| Destino de salto        |   | F | D  | IS |   |   |
| Destino+4               |   |   | F  | D  |   |   |
| Destino+8               |   |   |    | F  |   |   |
|                         |   |   |    |    |   |   |
| Salto T, predicción NT  | F | D | IS | X  | M | W |
| PC+4                    |   | F | D  | IS |   |   |
| PC+8                    |   |   | F  | D  |   |   |
| PC+12                   |   |   |    | F  |   |   |
| Destino de salto        |   |   |    |    | F |   |

En resumen, tenemos las siguientes penalizaciones:

|                   |   |
|-------------------|---|
| Salto NT, acierto | 0 |
| Salto NT, fallo   | 3 |
| Salto T, acierto  | 0 |
| Salto T, fallo    | 3 |

Por lo que tenemos una penalización media por salto de  $0.2 \cdot 3 = 0.6$  ciclos.

### Ejemplo 3.8

#### Comparación de esquemas de predicción en el nanoMIPS con ranura de 3 ciclos.

Vamos a comparar en este ejemplo la predicción estática de salto no tomado que realiza el nanoMIPS (recordemos que la de salto tomado no tiene sentido ya que se calcula la dirección destino de salto en la misma etapa en la que el salto evalúa la condición y determina si se salta o no), con la utilización de un BTB con la predicción implícita (sólo se almacenan en el BTB los saltos que se predicen como tomados) al que se accede en la etapa D.

Hasta que se accede al BTB en esta etapa, la ruta de datos se comporta con normalidad buscando la instrucción siguiente en la memoria de instrucciones, trabajo que luego se podrá aprovechar o no dependiendo de la predicción realizada por el BTB. Además el BTB tiene dos puertos de acceso, uno de lectura y otro de escritura.

Para realizar la comparación de ambos esquemas de predicción se determina que en media, el 19% de las instrucciones son saltos condicionales. De estas instrucciones el 80% son saltos tomados y además sabemos que el BTB acierta su predicción el 88% de las veces. Supondremos que las instrucciones que no son saltos tienen un CPI igual a 1. Con la predicción estática de salto no tomado en el nanoMIPS con ranura de 3 ciclos:

|          |   |   |   |   |   |  |
|----------|---|---|---|---|---|--|
| Salto NT | F | D | X | M | W |  |
| PC+4     |   | F | D | X |   |  |
| PC+8     |   |   | F | D |   |  |
| PC+12    |   |   |   | F | ↓ |  |

|                  |   |   |   |   |   |  |
|------------------|---|---|---|---|---|--|
| Salto T          | F | D | X | M | W |  |
| PC+4             |   | F | ✗ | * |   |  |
| PC+8             |   |   | F | ✗ |   |  |
| PC+12            |   |   |   | F | ↓ |  |
| Destino de salto |   |   |   |   | F |  |

En caso de utilizar un BTB:

|                         |   |   |   |   |   |  |
|-------------------------|---|---|---|---|---|--|
| Salto NT, predicción NT | F | D | X | M | W |  |
| PC+4                    |   | F | D | X |   |  |
| PC+8                    |   |   | F | D |   |  |
| PC+12                   |   |   |   | F | ↓ |  |

|                        |   |   |   |   |   |  |
|------------------------|---|---|---|---|---|--|
| Salto NT, predicción T | F | D | X | M | W |  |
| PC+4                   |   | F | ↓ |   |   |  |
| Destino del salto      |   |   | F | D |   |  |
| Destino+4              |   |   |   | F | ↓ |  |
| PC+4                   |   |   |   |   | F |  |

|                       |   |   |   |   |   |  |
|-----------------------|---|---|---|---|---|--|
| Salto T, predicción T | F | D | X | M | W |  |
| PC+4                  |   | F | ↓ |   |   |  |
| Destino del salto     |   |   | F | D |   |  |
| Destino+4             |   |   |   | F | ↓ |  |
| Destino+8             |   |   |   |   |   |  |

|                        |   |   |   |   |   |  |
|------------------------|---|---|---|---|---|--|
| Salto T, predicción NT | F | D | X | M | W |  |
| PC+4                   |   | F | ↓ | D | * |  |
| PC+8                   |   |   | F | D |   |  |
| PC+12                  |   |   |   | F | ↓ |  |
| Destino de salto       |   |   |   |   | F |  |

Si calculamos el CPI en cada uno de los casos con los datos de este ejemplo:

- Predicción estática. Las penalizaciones son las siguientes:

|          |   |
|----------|---|
| Salto NT | 0 |
| Salto T  | 3 |

Por lo que el CPI es:

$$\text{CPI} = 1 + \% \text{saltos condicionales} \cdot \% \text{tomados} \cdot 3 = 1 + 0.19 \cdot 0.8 \cdot 3 = 1.456$$

- Predicción dinámica con BTB. Las penalizaciones son las siguientes:

|                   |   |
|-------------------|---|
| Salto NT, acierto | 0 |
| Salto NT, fallo   | 3 |
| Salto T, acierto  | 1 |
| Salto T, fallo    | 3 |

Por lo que el CPI en este caso es:

$$\begin{aligned} \text{CPI} &= 1 + \% \text{saltos condicionales} \cdot (\% \text{fallos} \cdot 3 + \% \text{aciertos} \cdot \% \text{tomados} \cdot 1) = \\ &= 1 + 0.19 \cdot (0.12 \cdot 3 + 0.88 \cdot 0.8 \cdot 1) = 1.202 \end{aligned}$$

Es decir, la mejora que obtiene el esquema dinámico respecto al estático es del 21% ( $1.456/1.202 = 1.211$ ).

Los saltos condicionales tomados son, por norma general, más frecuentes que los no tomados. Por ello, para mejorar el rendimiento de la predicción de saltos, se puede almacenar en el BTB directamente la instrucción destino de salto en lugar de la dirección en la que se encuentra esta instrucción. De esta manera si la instrucción es un salto que se toma, el destino del salto se encontrará en el BTB y no será necesario buscarlo en la memoria de instrucciones, es decir, no se realiza la etapa F para el destino del salto.

Así no será necesario que el acceso al BTB se complete en una única etapa de F, ya que la etapa F de la instrucción destino de salto no se llega a realizar. Esto permite que el acceso sea un poco más lento de lo normal, lo que se traduce en la posibilidad de tener un BTB de tamaño algo mayor.

Además este tipo de estructura que almacena directamente la instrucción destino de salto mejora mucho el rendimiento de los saltos incondicionales, permitiendo que en este tipo de salto el procesador siempre pueda ahorrarse la ejecución de las etapas de la instrucción de salto a partir de F, ya que al fin y al cabo este tipo de instrucción lo único que hace es modificar el valor del PC. Esta técnica suele denominarse branch folding (plegado de saltos) e implica que si en F se accede al BTB y una instrucción está marcada como salto incondicional (información que se obtiene fácilmente gracias al código de operación), se modifica el PC adecuadamente y la instrucción que pasa a la etapa D es directamente la instrucción destino del salto, que está almacenada en el propio BTB (figura 3.10).

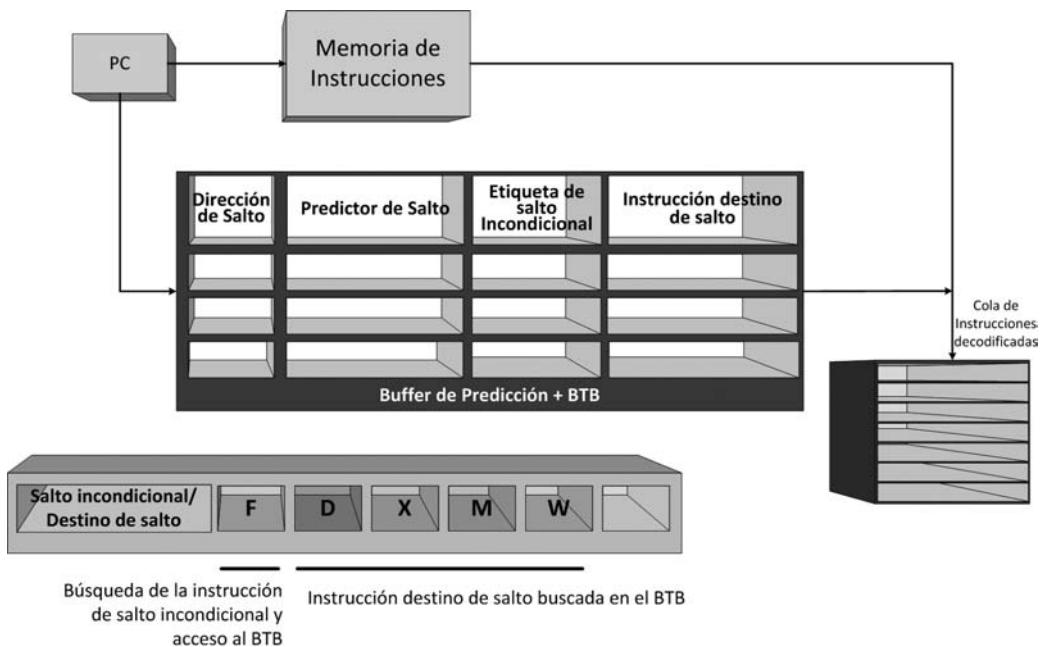


FIGURA 3.10

Branch folding.

Con este mismo objetivo, acelerar la búsqueda de la instrucción destino de salto, en algunas arquitecturas se ha propuesto no utilizar ni buffer de predicción ni BTB sino incluir directamente en la memoria caché de instrucciones el predictor de salto y un índice o puntero sucesor, que no es más que la dirección de la instrucción que debe ejecutarse tras el salto (la siguiente si la predicción es de salto no tomado o el destino de salto si la predicción es de salto tomado).

Aunque este tipo de solución mejora el rendimiento cuando se aciertan predicciones de salto tomado, el nanoMIPS no incorpora esta técnica por dos motivos fundamentales. El primero, obliga a aumentar la complejidad de la memoria caché de instrucciones, añadiendo a los accesos para la búsqueda de instrucción, otros necesarios para mantener y actualizar los predictores de salto y los índices sucesores. Y el segundo, la utilización de esta técnica no permite que se almacene más de un salto condicional por bloque de la caché de instrucciones (para almacenar así un único predictor e índice sucesor por bloque y no aumentar demasiado el tamaño de la caché).

Por último, hay que mencionar otras estructuras hardware que pueden incluirse en la ruta de datos para mejorar el rendimiento de cierto tipo de saltos muy concretos:

- **Pila de dirección de retorno.** La mayor parte de los saltos cuya dirección destino de salto se resuelve en tiempo de ejecución son retornos de procedimiento. Aunque se puede utilizar el BTB para

estos saltos, no es la técnica más adecuada, puesto que se puede llamar al mismo procedimiento desde distintas zonas de un programa y entonces las direcciones de retorno (los destinos del salto) irán variando en cada ejecución de la instrucción de retorno. Si se implementa una pila en la que se almacenan las direcciones desde las que se llama a los procedimientos, siempre y cuando sea lo suficientemente profunda, cuando se realicen los retornos se irán desapilando las direcciones adecuadas y se acertarán todas las predicciones.

- **Predictor de confianza de saltos.** La predicción de confianza de saltos no intenta predecir si un salto se va a tomar o no, intenta predecir si la predicción de salto va a acertar o no. Por lo tanto, el objetivo de esta estructura es estimar la confianza que se puede tener en la predicción de un salto y tener esta información en cuenta para generar la predicción en saltos que sean muy complicados de acertar. Normalmente se tiene en cuenta el historial de la predicción en las últimas ejecuciones del salto, almacenando un 0 si la predicción ha acertado y un 1 si la predicción ha fallado. Por ejemplo, si en el predictor de confianza de saltos se almacena para un salto concreto la siguiente secuencia de ceros y unos, 11000111, significa que para este salto, se han fallado dos predicciones, luego se han acertado tres y a continuación se han vuelto a fallar tres. Esta información puede utilizarse para mejorar la tasa de fallos de ciertos predictores complejos.

### 3.2.2. Predictores de salto

Como se ha mencionado en la sección anterior al explicar uno de los diseños más sencillos para el BTB, el predictor de saltos más fácil de implementar consiste en un único bit que vale 1 si la última vez se tomó el salto y se predice que la próxima vez se volverá a tomar o 0 si no se saltó, y por tanto se predice que la próxima vez no se tomará tampoco.

Este esquema tiene un problema que reduce su rendimiento con los saltos que tienen un comportamiento excepcional. Un buen ejemplo es el de un salto que se toma siempre, por ejemplo, la instrucción de salto que hay al final de un bucle, que se toma en todos los casos excepto en la última iteración del bucle. Por lo tanto, durante la ejecución del bucle el predictor siempre valdrá 1.

Para la última iteración se predice que se va a saltar y se falla. El bit de predicción se pone a 0. La siguiente vez que se ejecuta la instrucción de salto (primera pasada del bucle otra vez), se predice que no se va a saltar, y se falla de nuevo. Es decir, una excepción en un salto que siempre tiene el mismo comportamiento provoca dos fallos del predictor.

Para solucionar este problema se utilizan muy a menudo los esquemas de dos bits de predicción. Con un esquema como éste, una predicción debe fallar dos veces antes de que se modifique, por lo que el predictor es más estable ante excepciones en saltos con un comportamiento muy marcado. Como se observa en la figura 3.11, cuando el predictor vale 11 o 10, la predicción es que el salto se toma, y cuando el predictor vale 00 o 01, la predicción es que el salto no se toma.

Cuando el predictor vale 11 y el salto se toma, el acierto hace que no se modifiquen la predicción ni el predictor. Si por el contrario, el predictor falla porque el salto no se toma, se pasa al valor 10 para la siguiente ocasión. Pero este cambio en el predictor no implica un cambio en la predicción. Es decir, con un único comportamiento excepcional del salto, se modifica el valor del predictor pero no la predicción.

De esta manera, si de verdad era un comportamiento excepcional y la siguiente vez que se ejecuta el salto se vuelve a tomar, el predictor vuelve al valor 11 y sólo se ha fallado una vez. Si por segunda vez el salto no se toma y la predicción se falla, el predictor pasa a valer 00 de manera que en la siguiente ocasión sí se predice que el salto no se toma. Pero esto sólo ocurre después de dos ocasiones en las que se predice que se va a saltar y finalmente no se salta.

El mismo camino se puede recorrer comenzando con una predicción fuerte de no tomado en el estado 00 del predictor.

En general, se pueden utilizar esquemas de predicción de n bits con diagramas de transición de estados similares al de la figura 3.11. Para realizar la predicción hay que fijarse siempre en el bit más significativo del predictor.

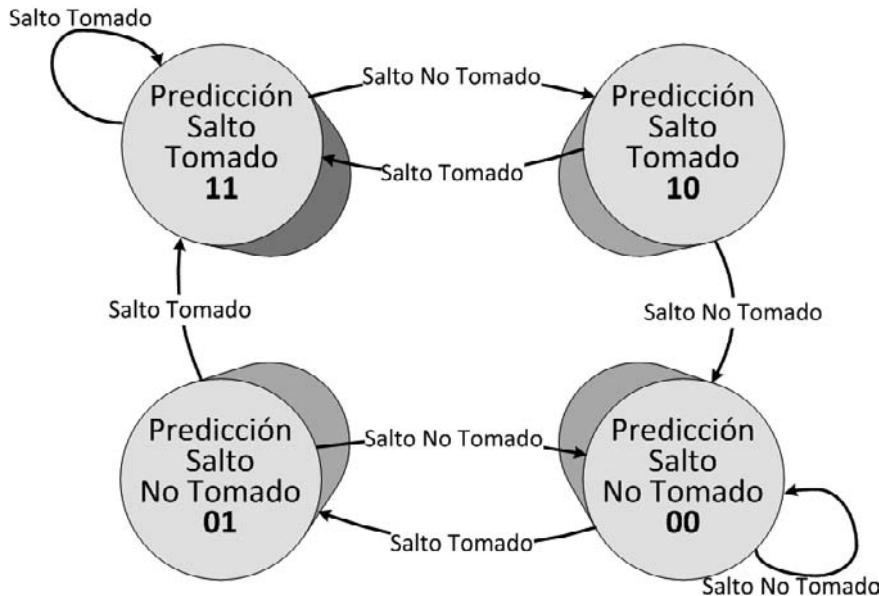


FIGURA 3.11

Estados de un predictor de dos bits.

Con estos esquemas el contador formado por los bits de predicción puede tomar valores entre 0 y  $2^{n-1}$ . Cuando el valor del contador es mayor que la mitad de su valor máximo, el salto se predice como tomado y en el resto de los casos se predice como no tomado. El contador puede mantener su valor, incrementarse o decrementarse según se acierten o se fallen las predicciones.

Estudios relacionados con los predictores de  $n$  bits han demostrado que el comportamiento de los predictores de 2 bits es suficientemente bueno, por lo que estos predictores son los más utilizados.

Sin embargo, estos predictores no siempre obtienen tasas de fallos suficientemente bajas. Esto se debe a que sólo utilizan información acerca del propio salto (local) para predecir lo que va a ocurrir en su próxima ejecución. Pero en algunos programas el comportamiento de unos saltos tiene que ver con el de otros, y con un predictor local de los estudiados hasta ahora, esto no puede tenerse en cuenta.

Los predictores multinivel o correlados tienen en cuenta información local (del salto para el que se realiza la predicción) y global (del resto de saltos del programa). El predictor más utilizado de este tipo es el más sencillo, el (1,1).

En este caso, el predictor asociado a cada salto es de dos bits, el primero es la predicción para el salto si el último salto del programa no se tomó, el segundo es la predicción para el salto si el último salto del programa se tomó. Por lo tanto este predictor se denomina (1,1) porque utiliza el comportamiento del último salto para escoger un predictor de un bit para el salto actual (figura 3.12).

Siempre que se utilicen predictores multinivel es necesario almacenar el comportamiento de los últimos saltos del programa en un registro de historia global de saltos (GBH o Global Branch History) que permita escoger el predictor que se debe utilizar en cada caso.

En general, un predictor multinivel ( $m,n$ ) utiliza el comportamiento de los  $m$  últimos saltos tomados para escoger un predictor de  $n$  bits para el salto actual. Los predictores locales que se habían estudiado hasta ahora en esta sección serían (0,1) el de 1 bit, y (0,2) el de 2 bits.

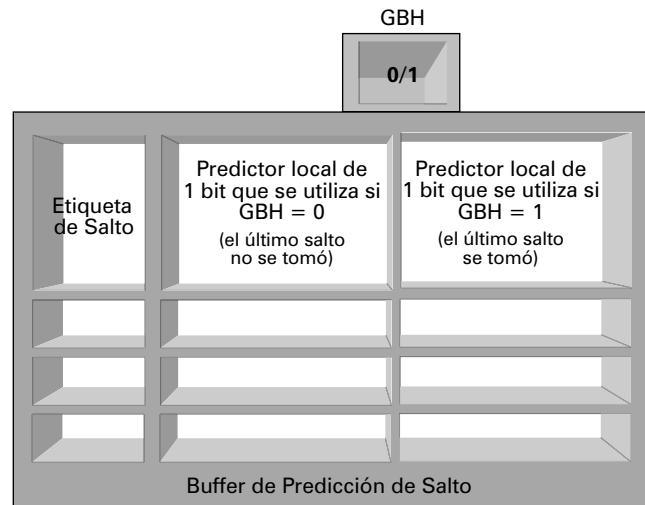


FIGURA 3.12

Predictor multinivel (1,1).

**Ejemplo 3.9****Ejemplo de utilización de un predictor multinivel (1,2).**

Vamos a comprender con un ejemplo sencillo el funcionamiento de estos predictores multinivel. Supongamos que ejecutamos el siguiente pseudocódigo en un procesador RISC:

```

ceros=0
for (i=0;i++;i<100)
 if vector(i)=0
 ceros=ceros+1;
 else
 vector(i)=vector(i)+10;

```

Si lo traducimos a ensamblador, quedaría algo como:

|        |                   |
|--------|-------------------|
| ADD    | R1,R0,R0          |
| ADD    | R10,R0,R0         |
| ADDI   | R2,R0,#400        |
| loop:  | LW R5,vector(R10) |
|        | BNEZ R5,else      |
|        | ADDI R1,R1,#1     |
|        | J final           |
| else:  | ADDI R5,R5,#10    |
|        | SW R5,vector(R10) |
| final: | ADDI R10,R10,#4   |
|        | BNEQ R10,R2,loop  |

Es decir, tenemos dos saltos condicionales y uno incondicional. Supongamos que utilizamos un predictor (1,2), esto significa que utilizamos la historia del último salto para escoger entre predictores locales de 2 bits. El valor inicial del GBH es 1, es decir, el último salto que se ha ejecutado se ha tomado. Y partimos de la siguiente situación inicial en el buffer de predicción:

| Etiqueta de salto | Predicción si GBH=0 | Predicción si GBH=1 |
|-------------------|---------------------|---------------------|
| ...               |                     |                     |
| BNEZ              | 00                  | 10                  |
| J                 | 01                  | 11                  |
| BNEQ              | 00                  | 11                  |

Si comenzamos la ejecución del código ejemplo con un vector que sólo almacena ceros:

- Al llegar a BNEZ la primera vez, GBH=1, por lo tanto se escoge para este salto la predicción de la columna 1: "10". Como es un predictor local de dos bits esto significa que se predice que el salto se toma. Pero como vector(1)=0, el salto no se toma. Por lo tanto, la predicción falla y hay que actualizar el buffer de predicción:

| Etiqueta de salto | Predicción si GBH=0 | Predicción si GBH=1 |
|-------------------|---------------------|---------------------|
| ...               |                     |                     |
| BNEZ              | 00                  | 00                  |
| J                 | 01                  | 11                  |
| BNEQ              | 00                  | 11                  |

- Al llegar al salto J, GBH=0 porque el último salto no se ha tomado. Por lo tanto se escoge la predicción de la columna 0: "01", y la predicción es que el salto no se toma. Pero es un salto incondicional que se toma, por lo tanto, hay que modificar el predictor:

| Etiqueta de salto | Predicción si GBH=0 | Predicción si GBH=1 |
|-------------------|---------------------|---------------------|
| ...               |                     |                     |
| BNEZ              | 00                  | 00                  |
| J                 | 11                  | 11                  |
| BNEQ              | 00                  | 11                  |

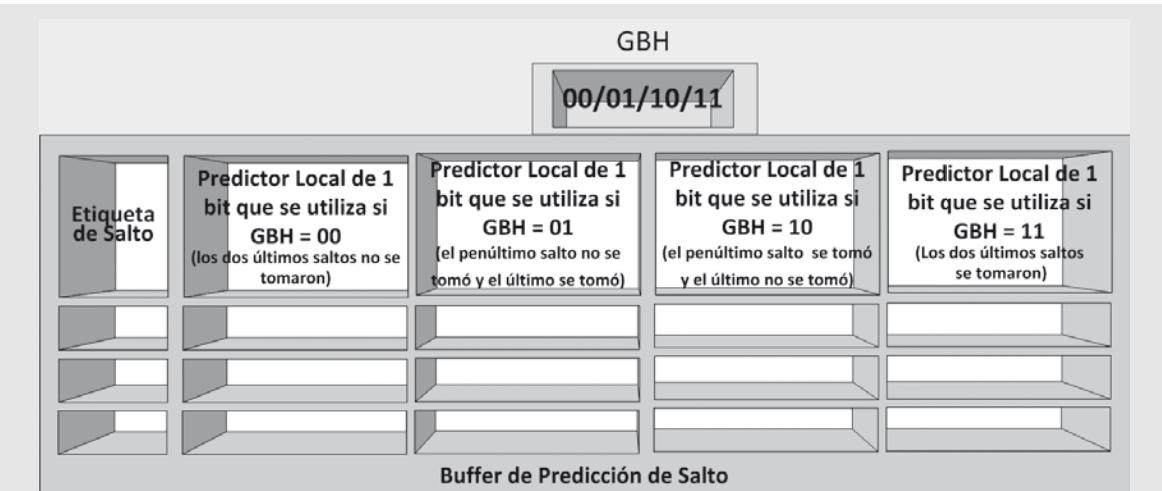
- Al llegar al salto BNEQ, GBH=1 porque el último salto se ha tomado. Así que se escoge la predicción de la columna 1: "11". Es decir, la predicción es de salto tomado y acierta, por lo que no hay que modificar el buffer de predicción.
- Al llegar de nuevo al salto BNEZ, GBH=1 porque el último salto se ha tomado, así que se escoge el predictor de la columna 1: "00". Esto implica que la predicción es que no se salta, y como vector(2)=0, no se salta y esta predicción acierta, así que no hay que corregir el buffer.

Y así sucesivamente.

### Ejemplo 3.10

Ejemplo de utilización de un predictor (2,1).

Repitamos el ejemplo anterior con un predictor (2,1), es decir, un predictor que utilice la historia de los dos últimos saltos para escoger entre predictores locales de 1 bit.



El valor inicial del GBH es 11, es decir, los dos últimos saltos que se han ejecutado se han tomado. Y partimos de la siguiente situación inicial en el buffer de predicción:

| Etiqueta de salto | Pred. si GBH=00 | Pred. si GBH=01 | Pred. si GBH=10 | Pred. si GBH=11 |
|-------------------|-----------------|-----------------|-----------------|-----------------|
| ...               |                 |                 |                 |                 |
| BNEZ              | 0               | 1               | 0               | 1               |
| J                 | 1               | 1               | 1               | 0               |
| BNEQ              | 0               | 1               | 1               | 0               |

Si comenzamos la ejecución del código ejemplo con un vector que sólo almacena ceros:

- Al llegar a BNEZ la primera vez, GBH=11, por lo tanto se escoge para este salto la predicción de la columna 11: "1". Como es un predictor local de 1 bit esto significa que se predice que el salto se toma. Pero como vector(1)=0, el salto no se toma. Por lo tanto, la predicción falla y hay que actualizar el buffer de predicción:

| Etiqueta de salto | Pred. si GBH=00 | Pred. si GBH=01 | Pred. si GBH=10 | Pred. si GBH=11 |
|-------------------|-----------------|-----------------|-----------------|-----------------|
| ...               |                 |                 |                 |                 |
| BNEZ              | 0               | 1               | 0               | 0               |
| J                 | 1               | 1               | 1               | 0               |
| BNEQ              | 0               | 1               | 1               | 0               |

- Al llegar al salto J, GBH=10 porque el penúltimo salto se tomó y el último no se ha tomado. Por lo tanto se escoge la predicción de la columna 10: "1", y la predicción es que el salto se toma por lo que tenemos un acierto.
- Al llegar al salto BNEQ, GBH=01 porque el último salto se ha tomado. Así que se escoge la predicción de la columna 01: "1". Es decir, la predicción es de salto tomado y acierta, por lo que no hay que modificar el buffer de predicción.
- Al llegar de nuevo al salto BNEZ, GBH=11 porque el último salto se ha tomado, así que se escoge el predictor de la columna 11: "0". Esto implica que la predicción es que no se salta, y como vector(2)=0, no se salta y esta predicción acierta, así que no hay que corregir el buffer.

Y así sucesivamente.

Tener en cuenta información global de todas las instrucciones de salto ejecutadas por un programa puede mejorar la tasa de fallos de un predictor local que sólo tiene en cuenta información relacionada con el salto para el que se va a hacer la predicción. Pero puede que esta mejora no sea suficiente para compensar la utilización de un hardware más complejo o incluso que no exista mejora alguna dependiendo del programa.

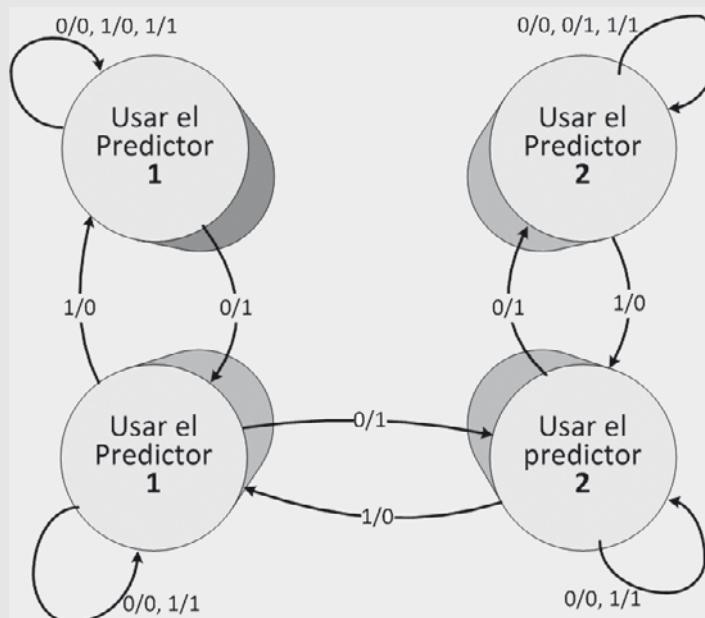
En general, los programas incluyen una mezcla de distintos tipos de saltos que pueden predecirse mejor o peor con los diferentes tipos de predictores. Por este motivo existen diferentes técnicas que permiten utilizar dos o más esquemas de predicción dinámica de saltos para realizar la predicción final más adecuada.

Existen técnicas muy complejas basadas, por ejemplo, en la información proporcionada por el predictor de confianza de saltos o en fusionar las predicciones proporcionadas por dos o más predictores. Pero también se pueden diseñar sencillos predictores adaptativos o de torneo con la capacidad de escoger entre dos predictores según cuál se vaya a comportar mejor para un determinado salto.

### Ejemplo 3.11

Diseño de un predictor adaptativo que escoja entre un predictor local y uno global.

En el ejemplo de la figura se puede escoger entre dos predictores diferentes, uno local (predictor 1) y otro global (predictor 2). Los estados del diagrama indican qué predictor se utiliza en cada uno de ellos.



En el diagrama de transición de estados se indica si cada uno de los predictores ha acertado (1) o fallado en la última predicción (0). Por ejemplo, la transición 1/0 es la que ocurre cuando el predictor 1 acierta y el predictor 2 falla.

Si se está utilizando el predictor 1, mientras éste acierte, da igual lo que ocurra con el predictor 2. En el momento en el que el predictor 1 falle y el predictor 2 acierte, se cambia de estado pero se sigue utilizando el predictor 1. Si en la siguiente predicción vuelve a ocurrir lo mismo, que el predictor 1 falla y el predictor 2 acierta, entonces sí se pasa a un estado en el que se utiliza el predictor 2.

Se puede observar que se ha seguido una política similar a la de los predictores de dos bits. Es decir, no se cambia de predictor a no ser que falle en dos predicciones consecutivas, así se eliminan los problemas que se puedan producir por comportamientos excepcionales de los saltos y de los predictores.

Este sencillo diagrama permite establecer una competición o torneo para utilizar en cada momento el predictor que acierta en más ocasiones.

### 3.3 Emisión múltiple de instrucciones

Todas las técnicas de diseño de procesadores estudiadas hasta este momento persiguen la consecución de un CPI ideal igual a 1 en procesadores segmentados, a pesar de la existencia de los distintos tipos de riesgos.

Sin embargo este valor límite para el CPI viene impuesto, simplemente, por la emisión de una única instrucción por ciclo de reloj. Si el objetivo es reducir el valor del CPI por debajo de 1, es imprescindible emitir más una instrucción por ciclo: emisión múltiple de instrucciones.

Existen distintos tipos de arquitecturas que permiten emitir más de una instrucción por ciclo. La diferencia principal entre ellas está en la manera de realizar la emisión de instrucciones, de resolver los riesgos y de planificar el orden de ejecución (tabla 3.1).

**TABLA 3.1**  
Arquitecturas que permiten la emisión múltiple de instrucciones.

| Arquitectura                           | Emisión  | Detección de riesgos | Planificación                                                                                   |
|----------------------------------------|----------|----------------------|-------------------------------------------------------------------------------------------------|
| VLIW                                   | Estática | Software             | Estática                                                                                        |
| Superescalar estática                  | Dinámica | Hardware             | Estática con ejecución en orden                                                                 |
| Superescalar dinámica                  | Dinámica | Hardware             | Dinámica, esperando a la resolución de los saltos                                               |
| Superescalar dinámica con especulación | Dinámica | Hardware             | Completamente dinámica, es posible ejecutar instrucciones que dependen de predicciones de salto |

#### 3.3.1. VLIW

Los procesadores Very Long Instruction Word (VLIW, Palabra de Instrucción Muy Larga), estuvieron unos años casi en desuso pero han vuelto a estar de actualidad porque han inspirado el diseño de arquitecturas como la de los procesadores Itanium e Itanium 2 (arquitecturas EPIC), que han resuelto la mayor parte de los problemas que presentaban los antiguos diseños VLIW.

En este tipo de arquitecturas el compilador empaqueta un conjunto de instrucciones que pueden ejecutarse en paralelo en una única instrucción muy larga. Es decir, el compilador especifica explícitamente el paralelismo en cada instrucción, no es una responsabilidad del procesador.

El compilador debe tener en cuenta que no existan riesgos de datos ni de control entre las subinstrucciones que se empaquetan juntas y que haya recursos suficientes en el procesador para ejecutarlas todas en paralelo. Si se supone un procesador como el de la figura 3.13, con cuatro tipos diferentes de unidades funcionales y recursos de ejecución (por ejemplo, una ALU de enteros, un sumador/restador en coma flotante, un multiplicador/divisor en coma flotante y las unidades de carga y almacenamiento), se deberán empaquetar siempre cuatro subinstrucciones juntas, una de cada tipo, en cada palabra de instrucción muy larga. Lo habitual es que estas subinstrucciones avancen de manera sincronizada por la ruta de datos, de manera que si una se detiene se detengan también las demás, comenzando así cada etapa de ejecución todas al mismo tiempo.

Como en este tipo de arquitectura no son necesarias estructuras para la planificación dinámica de instrucciones (ni ventana de instrucciones ni estaciones de reserva, la planificación la realiza el compilador al empaquetar las instrucciones), se pueden incluir más unidades funcionales para realizar opera-

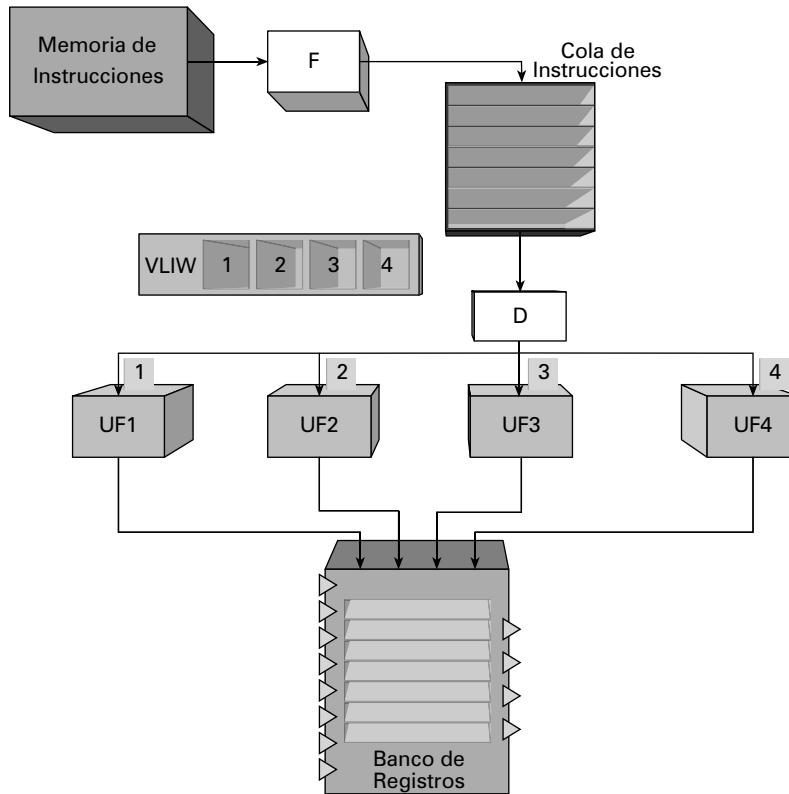


FIGURA 3.13

Ruta de datos de un procesador segmentado multifuncional VLIW.

ciones, de manera que el diseño es el de un procesador segmentado sencillo pero con gran cantidad de recursos de ejecución. Además es necesario incluir puertos adicionales de lectura y escritura en el banco de registros, para que todas las subinstrucciones puedan leer sus operandos y escribir sus resultados en la etapa adecuada.

Para mantener todas estas unidades funcionales del procesador ocupadas, el paralelismo inherente al código tiene que ser alto. Es habitual utilizar técnicas que lo aumenten como el desenrollado de bucles y la reordenación del código. En algunos casos también se modifica el diseño del procesador para que ayude al compilador a aumentar el paralelismo, por ejemplo, es habitual incluir instrucciones que permitan la utilización de predicados en los repertorios de este tipo de procesadores.

El principal problema de este tipo de arquitectura ha estado tradicionalmente en el tamaño de los códigos, ya que en cada instrucción suelen quedar huecos al ser imposible empaquetar siempre subinstrucciones que llenen palabras de instrucción muy largas por completo debido a los riesgos. Estos huecos ocupan espacio en la memoria innecesariamente.

Además, existen problemas de compatibilidad entre diferentes arquitecturas. Incluso entre diferentes implementaciones de la misma arquitectura (una nueva unidad funcional o una latencia diferente para alguna de las unidades, por ejemplo), sería necesario recomilar el código para planificar adecuadamente las subinstrucciones.

Los procesadores VLIW han sido una buena opción para el diseño de sistemas empotrados, ya que consiguen la emisión múltiple de instrucciones con menor consumo de potencia que otro tipo de diseños en los que la responsabilidad de la planificación recae sobre el hardware. Pero no se han utilizado mucho

en otro tipo de sistemas, en los que se ha comprobado que cuando un procesador VLIW puede conseguir un rendimiento alto, también puede hacerlo casi siempre un procesador vectorial, con un diseño del compilador y del propio procesador bastante más sencillo.

En este caso el diseño persigue aprovechar de manera eficiente el paralelismo de datos realizando el procesamiento sobre variables agrupadas en vectores, de manera que los operandos y los resultados sean siempre vectores. Este concepto se ha incorporado en casi todas las arquitecturas actuales que no son VLIW ni vectoriales pero que incluyen extensiones multimedia en los repertorios de instrucciones, diseñadas para aprovechar el paralelismo SIMD (Single Instruction Multiple Data) trabajando con vectores.

### 3.3.2. Superescalar estática

En la figura 3.14 se muestra la evolución seguida hasta el momento en este libro. Se comenzó por un procesador secuencial en el capítulo 1, en este mismo capítulo se explicó cómo convertirlo en un procesador segmentado y por último en un procesador segmentado multifuncional (en la figura se muestran las cuatro bifurcaciones típicas que se han utilizado previamente: ALU de enteros, sumador/restador en coma flotante segmentado, multiplicador/divisor en coma flotante segmentado y ALU/traducción de dirección virtual a física y memoria de datos para las instrucciones de carga/almacenamiento). En la figura 3.15 se muestra cómo convertir este último procesador en uno superescalar estático capaz de emitir dos instrucciones por ciclo.

En un procesador superescalar típico se emiten entre 1 y 8 instrucciones por ciclo. Además, si la planificación es estática, la emisión de instrucciones debe realizarse en orden y los riesgos se detectan y resuelven antes de la emisión. Por lo tanto, las instrucciones que se emiten simultáneamente deben

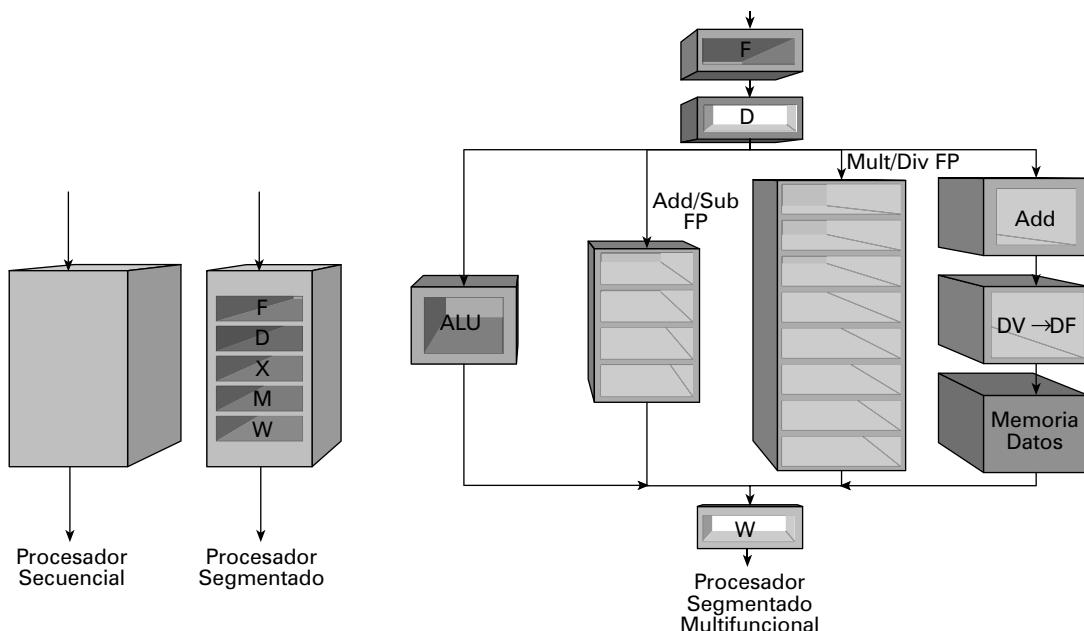


FIGURA 3.14

Evolución en las técnicas de diseño del procesador: procesador secuencial, procesador segmentado y procesador segmentado multifuncional.

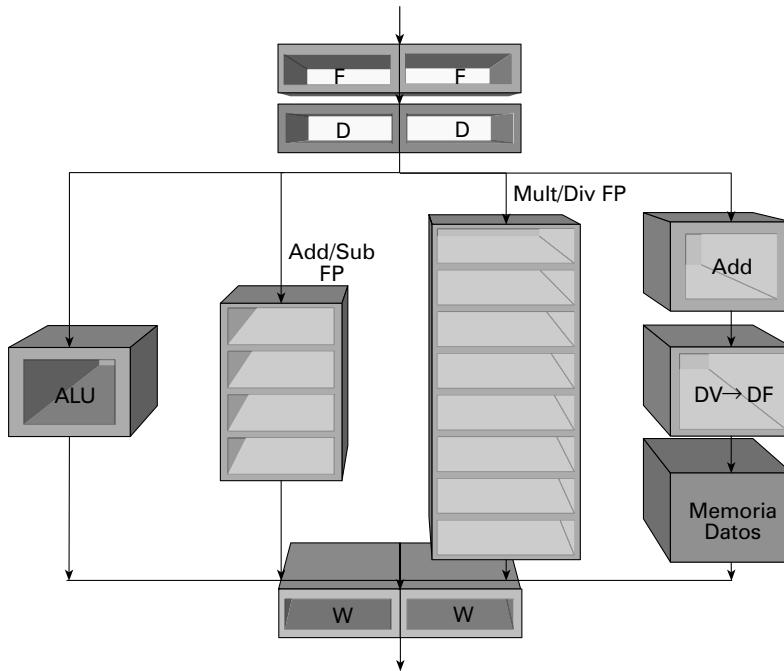


FIGURA 3.15

Procesador segmentado multifuncional superescalar estático con emisión de dos instrucciones por ciclo.

ser independientes y satisfacer una serie de restricciones. Por ejemplo, no se suele emitir en un ciclo más de una instrucción que necesite hacer una referencia a memoria para evitar riesgos estructurales.

Como la emisión debe realizarse en orden, si en un bloque de instrucciones se encuentran dependencias, sólo se podrán emitir las instrucciones anteriores a la primera que presenta una dependencia. La ordenación que permite aprovechar el paralelismo entre instrucciones debe ser realizada por el compilador, ya que el procesador emite y ejecuta las instrucciones en orden, simplemente comprueba que las instrucciones que se van a emitir en paralelo pueden ejecutarse en paralelo porque no hay riesgos que lo impidan.

En general los diseños superescalares complican las etapas F y D puesto que es necesaria la búsqueda y decodificación de más de una instrucción por ciclo. Estas dos etapas deben realizar las siguientes tareas:

- Búsqueda de instrucciones en memoria y almacenamiento temporal.
- Predecodificación de instrucciones, como mínimo, para determinar el tipo de cada instrucción e identificar el camino que se va a seguir por la ruta de datos, los recursos que se van a utilizar, etc. Este paso permite además realizar una identificación de saltos temprana.
- Predicción dinámica de saltos.
- Decodificación completa de las instrucciones.
- Detección de riesgos.

Esta última tarea es necesaria para que se envíen a la ruta de datos bloques de instrucciones en los que ya se ha comprobado que es posible la emisión múltiple porque se cumplen todos los requisitos.

En el próximo capítulo se estudiarán técnicas que mejoren el rendimiento de la memoria para que sea posible buscar más de una instrucción por ciclo. Además de estas técnicas, será necesario incluir en la ruta de datos nuevos comparadores que permitan realizar la detección de riesgos entre todas las instrucciones.

En cuanto a la emisión de instrucciones, la diferencia fundamental con los procesadores segmentados multifuncionales que utilizan planificación dinámica pero sólo emiten una instrucción por ciclo está en que en el caso de los procesadores superescalares serán necesarios múltiples puertos de lectura y escritura en el banco de registros para que las instrucciones que se emitan en el mismo ciclo puedan leer sus operandos y escribir sus resultados al mismo tiempo si así lo necesitan.

La emisión en el nanoMIPS superescalar estático es de dos instrucciones por ciclo. Hay dos alternativas para parear instrucciones:

- Emitir una instrucción entera (cargas, almacenamientos, saltos y aritmético-lógicas con enteros) y una en coma flotante al mismo tiempo.
- Emitir una instrucción y estudiar si las dependencias permiten emitir también la siguiente.

Casi siempre se utiliza la primera solución porque es mucho más sencilla. No complica apenas el hardware de detección de riesgos, porque la única posibilidad de riesgo de datos entre una instrucción entera y una de coma flotante aparece con las instrucciones de carga y almacenamiento de operandos de coma flotante (que son instrucciones enteras).

Por lo tanto, pareando instrucciones enteras y de punto flotante, la detección de riesgos entre las instrucciones del paquete de emisión es muy sencilla, ya que sólo hay que comprobar estas instrucciones. Pero como las dos instrucciones que se emiten al mismo tiempo deben avanzar en paralelo por la ruta de datos, se desaprovecha el hardware para operaciones con enteros enormemente, ya que las operaciones en coma flotante tardan mucho más en completar su etapa de ejecución (figura 3.16).

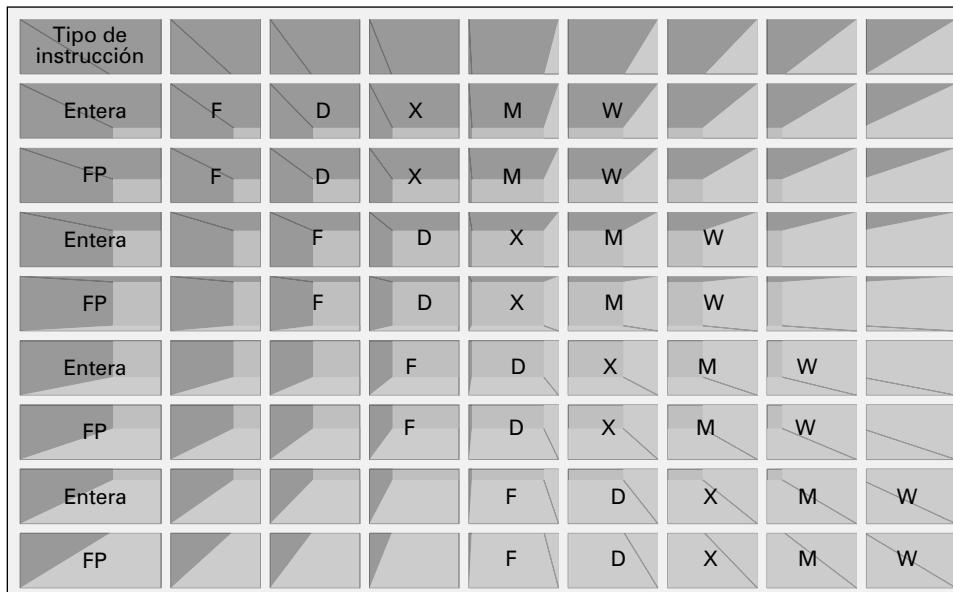


FIGURA 3.16

Diagrama de ejecución del nanoMIPS superescalar estático con emisión de dos instrucciones por ciclo, una de enteros y una de coma flotante.

Resumiendo, las modificaciones más importantes que se deben realizar en el nanoMIPS para obtener un diseño superescalar a partir del segmentado son:

- Modificar la unidad de búsqueda de instrucciones para que puedan buscarse dos instrucciones en cada ciclo.
- Segmentar las unidades funcionales de FP y replicar las que sea necesario. Si no se hace esto, el cauce de FP se convierte rápidamente en el cuello de botella (debido a la gran latencia de este tipo de operaciones).
- Añadir los caminos para los adelantamientos necesarios para solucionar los nuevos riesgos de datos.
- Sería deseable añadir un puerto extra de lectura/escritura en el banco de registros de FP para poder emitir simultáneamente las instrucciones de carga y almacenamiento de valores en FP y una instrucción de FP.
- Por último, destacar que el tratamiento preciso de excepciones puede complicarse en el caso del nanoMIPS superescalar porque hay el doble de instrucciones ejecutándose simultáneamente en la ruta de datos, tratándose además de una mezcla de instrucciones enteras y de FP. Sin embargo, las técnicas utilizadas para implementar el modo preciso son muy similares a las del nanoMIPS clásico que se explicaron en el capítulo 1.

El problema es que, aunque se hagan estas modificaciones, sólo se consigue el rendimiento máximo si existe una proporción determinada de instrucciones de enteros y coma flotante en los códigos: el 50% de cada tipo (y aún así, se desaprovecha el hardware de enteros). En cuanto no se cumpla esta proporción ya no será posible emitir dos instrucciones en todos los ciclos y algunas unidades funcionales del procesador estarán infrautilizadas.

Además, al tener tantas instrucciones ejecutándose simultáneamente en la ruta de datos, las paradas por riesgos afectan a un número mucho mayor de instrucciones y empeoran todavía más el rendimiento que en el nanoMIPS que emite una única instrucción. Por eso se hace necesaria la utilización de técnicas de planificación dinámica que resuelvan los riesgos en tiempo de ejecución y que además permitan al propio procesador reordenar las instrucciones de la manera más adecuada.

### 3.3.3. Superescalar dinámica

La planificación dinámica es la técnica que permite obtener el máximo rendimiento en un procesador con emisión múltiple de instrucciones. La principal ventaja que ofrece es la resolución de riesgos de datos sin necesidad de un gran número de ciclos de parada.

La idea sería extender los algoritmos de planificación que se han estudiado al principio de este capítulo, especialmente los algoritmos distribuidos como Tomasulo (que son los que menos paradas introducen en la ruta de datos) para que permitan emitir dos o más instrucciones arbitrarias por ciclo (figura 3.17).

La principal limitación está en que no se pueden emitir instrucciones que dependan de saltos antes de que estos se resuelvan. Y recordemos que tampoco pueden desordenar la ejecución de las instrucciones de acceso a memoria.

Por eso los procesadores superescalares con planificación dinámica siempre permiten combinar esta planificación dinámica con la predicción de saltos y la ejecución desordenada de los accesos a memoria, de manera que se pueda avanzar trabajo de las instrucciones predichas mientras se resuelven los saltos y los cálculos de las direcciones de acceso a memoria. Esta técnica se conoce como especulación.

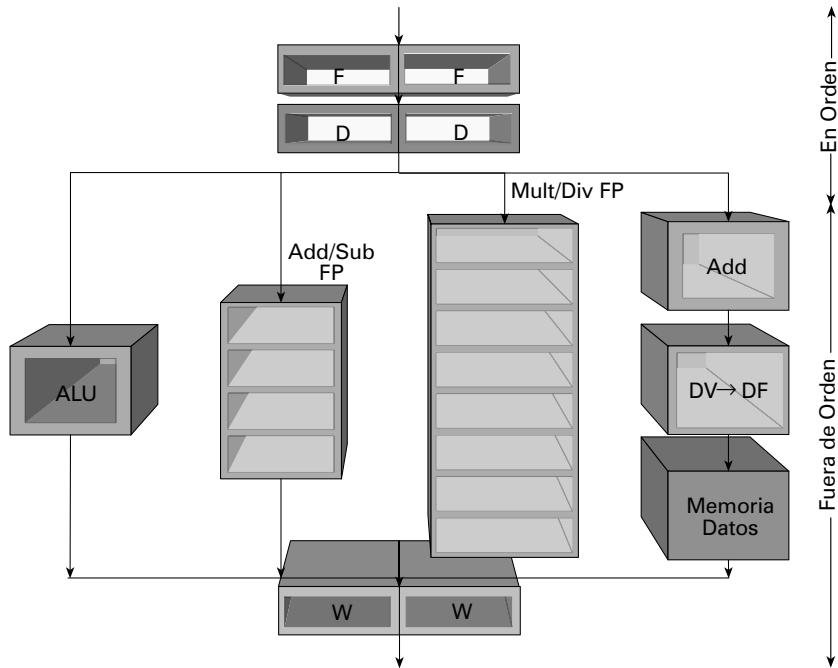


FIGURA 3.17

Procesador segmentado multifuncional con planificación dinámica y superescalar dinámico con emisión de dos instrucciones por ciclo.

## 3.4 Especulación

La especulación permite ejecutar instrucciones que dependen de las predicciones de los saltos o de los cálculos necesarios para obtener las direcciones de acceso a memoria. Se supone que estas predicciones son correctas y se continúa la ejecución normalmente. Por lo tanto esta técnica puede aprovechar mucho más que las estudiadas hasta ahora el hardware incluido en el procesador, permitiendo que la planificación dinámica pueda escoger entre todas las instrucciones, incluidos los saltos y los accesos a memoria, y sin las limitaciones que antes imponían estas instrucciones (figura 3.18). Para ello es necesario añadir una nueva etapa a la ejecución de instrucciones, denominada de terminación o commit, en la que se comprueba si las instrucciones tenían que ejecutarse realmente o no y si por tanto, deben escribir sus resultados en los registros y en memoria o no.

Obviamente, esta técnica implica un coste, que puede ser simplemente el haber ejecutado instrucciones innecesariamente si alguna de las predicciones de las que dependían ha resultado ser incorrecta (esto afecta especialmente al consumo de potencia del procesador), o incluso el tener que deshacer o compensar las modificaciones que estas instrucciones han realizado cuando luego se comprueba que no deberían haberse ejecutado o que no se ha resuelto un riesgo RAW entre una instrucción de almacenamiento y una de carga adecuadamente (aunque esto se evita siempre que es posible porque implica generar y ejecutar nuevo código). Además, como se verá un poco más adelante, la especulación obliga a modificar el tratamiento de excepciones dentro del procesador.

En todos los procesadores que se basan en la especulación se utiliza planificación dinámica distribuida, es decir, basada en estaciones de reserva y en el renombramiento de registros, ya que es el tipo de

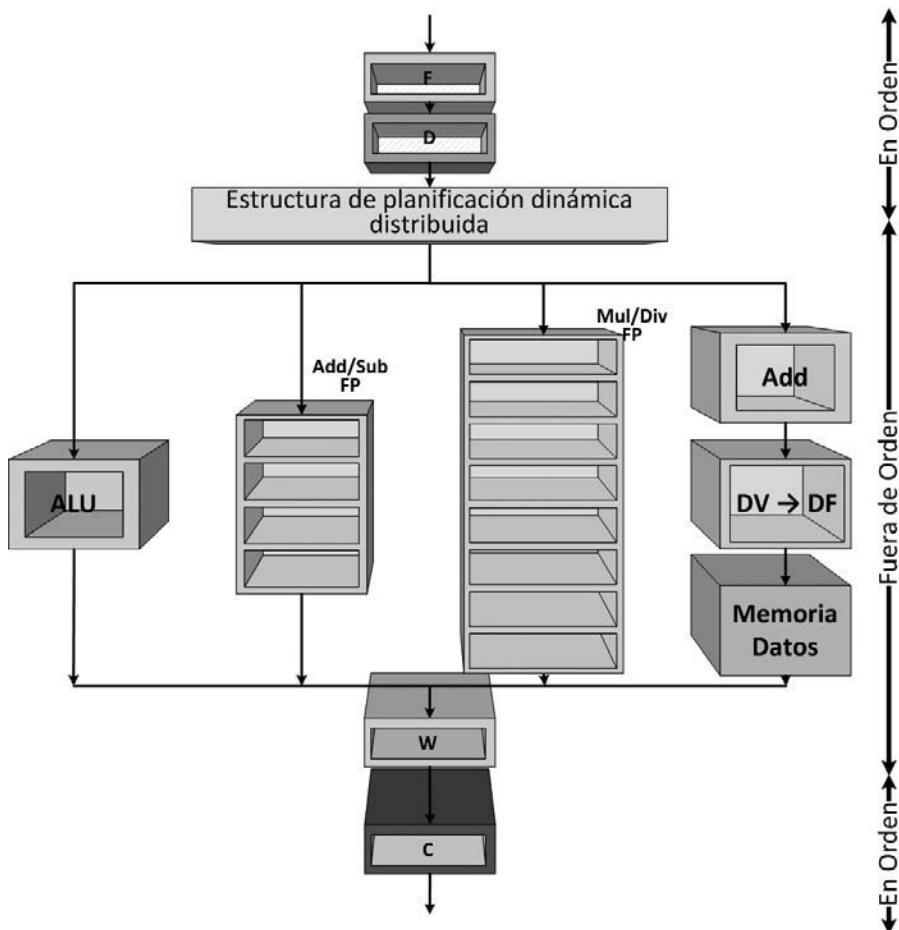


FIGURA 3.18

Procesador segmentado multifuncional con planificación dinámica y especulación.

planificación que más reduce las paradas por riesgos de datos, lo que es especialmente importante cuando se diseña un procesador con emisión múltiple en el que se van a estar ejecutando tantas instrucciones al mismo tiempo. Pero para soportar la especulación hay que hacer una serie de modificaciones en el hardware que implementa este tipo de planificación.

La primera modificación afecta al CDB (Common Data Bus). Es necesario separar el hardware que permite el adelantamiento de operandos entre instrucciones del que se utiliza para escribir los resultados definitivos en el banco de registros.

La idea fundamental es que las instrucciones especuladas no puedan modificar el estado del procesador. Para conseguir esto, se permite ejecución fuera de orden pero se obliga a que la terminación de las instrucciones sea en orden (es en esta terminación en la que se permitirá escribir en el banco de registros o en memoria). Esta terminación o commit se produce cuando se resuelven los saltos previos y se conoce a ciencia cierta la secuencia de código que debe ejecutarse.

El hardware que permite estas modificaciones se denomina Buffer de Reordenamiento (ROB). Se trata de una estructura que se encarga de almacenar el resultado de las instrucciones especuladas desde que finalizan su ejecución hasta que se les permite terminar. De esta manera, el ROB se encarga de es-

cribir los resultados en el banco de registros y en memoria cuando se sabe que son definitivos porque las instrucciones han llegado a la etapa de terminación.

El tamaño de la ventana de instrucciones ya no está sólo limitado por el número de estaciones de reserva, sino también por el número de entradas en el ROB, ya que no se puede emitir una instrucción si no se le puede asociar una entrada en este buffer.

En el nanoMIPS con especulación las etapas por las que pasan las instrucciones son cuatro, la primera y la cuarta deben realizarse siempre en orden. Es como si se añadiera a la planificación con el algoritmo de Tomasulo un ROB y una etapa de commit (figura 3.19). Primero se supondrá que se emite una instrucción por ciclo:

- Emisión (E).** Se retira la instrucción que está en la parte superior de la cola de instrucciones. Si hay alguna estación de reserva adecuada para esta instrucción libre y existe una entrada libre en el ROB, se envía la instrucción junto con el valor de los operandos que ya estén disponibles. Si se produce riesgo estructural por alguno de los dos recursos, la estación de reserva o el ROB, hay que parar la emisión de instrucciones hasta que se resuelva. Por el contrario, si algún operando no está disponible, la instrucción se emite y se queda a la espera en su estación de reserva. Para ello se registra en la estación de reserva qué instrucción es la que va a producir este operando como resultado. Este registro se realiza con la entrada en la que la instrucción que produce el operando está almacenada en el ROB.

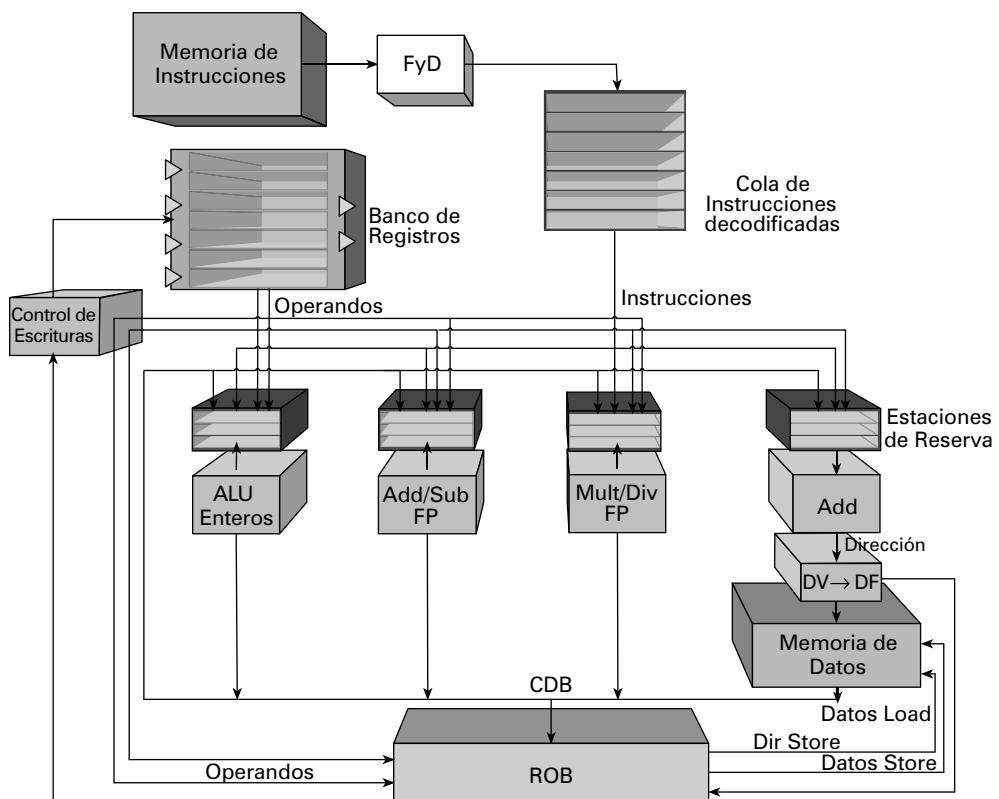


FIGURA 3.19

Ruta de datos del nanoMIPS segmentado multifuncional con planificación Tomasulo y especulación.

2. **Ejecución (X).** Una instrucción se envía de la estación de reserva a la unidad funcional cuando ésta está libre y la instrucción ya tiene en la estación de reserva sus operandos. Las instrucciones monitorizan el CDB hasta que todos sus operandos estén disponibles. Cuando se cumpla esta condición, se puede pasar a ejecutar la instrucción, de esta manera se solucionan los riesgos RAW. Si no está libre la unidad funcional que se requiere, habrá que esperar hasta que se resuelva el riesgo estructural. Si más de una instrucción puede enviarse a la unidad funcional en el mismo ciclo, se envía siempre la que lleva más tiempo esperando en la estación de reserva.
3. **Escritura de resultado (W).** Se escribe el resultado de la operación (junto con la etiqueta que indica la entrada correspondiente a la instrucción en el ROB) en el CDB. El ROB almacena este resultado temporalmente y cualquier instrucción que estuviera esperando por este resultado en una estación de reserva lo puede leer del CDB.
4. **Terminación o Commit (C).** En esta etapa, que debe realizarse en orden, pueden ocurrir tres cosas diferentes:
  - a) Que la instrucción se haya especulado correctamente y haya que actualizar el valor que estaba almacenado en el ROB en el banco de registros. Una vez hecha esta actualización, se puede eliminar la instrucción del ROB.
  - b) Que la instrucción se haya especulado correctamente y sea un almacenamiento. Entonces lo que hay que actualizar es la memoria en lugar del banco de registros.
  - c) Que se haya realizado alguna predicción incorrecta y la instrucción no se ha debido ejecutar. En este caso se elimina directamente del ROB sin realizar ninguna actualización. Normalmente después de una instrucción de salto cuya predicción se ha fallado, se eliminan todas las entradas del ROB y se comienza de nuevo buscando la instrucción correcta. Si el problema es que no se ha resuelto un riesgo RAW entre un almacenamiento en memoria y una carga, si la especulación afectaba sólo a la lectura de memoria, ésta se vuelve a ejecutar. Si afectaba a otras operaciones que utilizaban el operando leído de memoria, habrá que repetirlas.

Cuando se utiliza especulación se mantienen los mismos dos tipos de estructuras de planificación que con la planificación de Tomasulo, las estaciones de reserva y el control de escrituras (figura 3.5), pero además es necesario mantener la estructura de ROB (figura 3.20) con un puntero que indique la primera entrada libre que hay en esta estructura (como las instrucciones hacen el commit en orden, es sencillo gestionar este puntero).

En algunos casos, se añaden nuevas colas de instrucciones al procesador, de manera que cada vez que se realiza una predicción de salto, las instrucciones predichas se van volcando a una cola de instrucciones diferente. Si la predicción del salto falla, con esta separación de instrucciones por colas es más fácil recuperar el fallo y desechar las instrucciones dependientes de esta predicción equivocada que se han buscado y que finalmente no se van a ejecutar.

Si además de utilizar especulación, el procesador es superescalar, las instrucciones pasan por estas mismas etapas pero se emite más de una instrucción por ciclo. Como es de esperar que termine más de una instrucción por ciclo, se suele modificar el CDB para que permita más de una escritura simultánea de resultados en este caso.

En cualquiera de los dos casos, la resolución de la especulación asociada a las predicciones de salto es relativamente sencilla, pero para resolver la especulación de los accesos a memoria es necesario utilizar instrucciones centinela.

La idea es colocar un centinela de este tipo en la posición en la que se encuentra la lectura de memoria especulativa que se permite que se ejecute antes que algún almacenamiento. Cuando se ejecuta el load especulativo, se almacena temporalmente la dirección de memoria a la que se accede, de manera que se pueda comparar con la que utilizan los almacenamientos. Si no coinciden, la especulación es correcta.

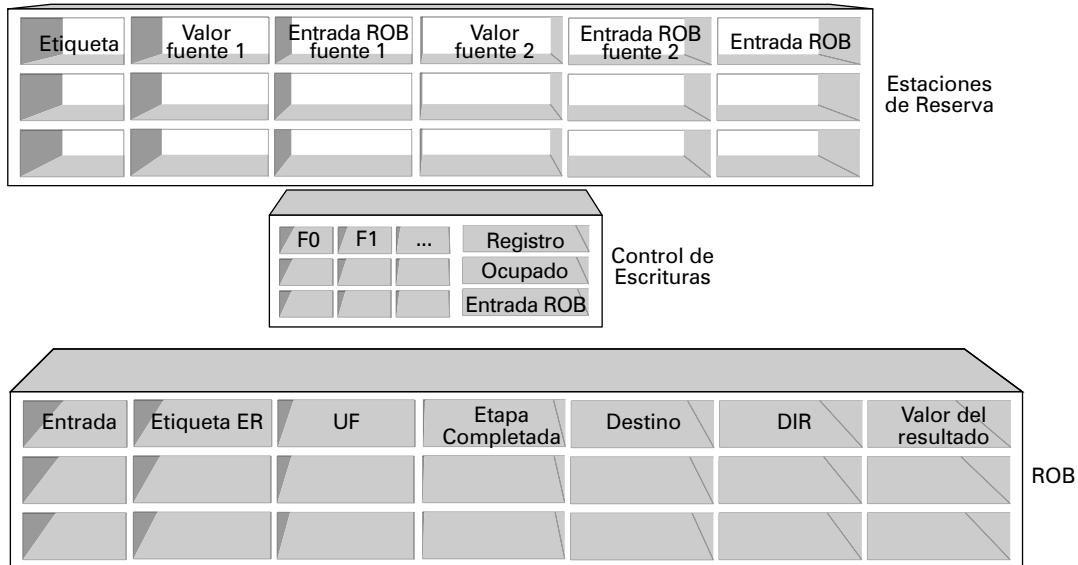


FIGURA 3.20

Implementación de las estaciones de reserva, el control de escrituras y el ROB en el nanoMIPS especulativo.

Pero si alguna dirección de un store que aparece antes en el código coincide con la del load, significa que la especulación ha fallado y que el load (y las instrucciones que dependen de esta lectura de memoria) deben repetirse. ¿Cuándo? Cuando se llegue a la instrucción centinela.

### Ejemplo 3.12

Planificación dinámica de instrucciones con el algoritmo de Tomasulo especulativo en el nanoMIPS con una unidad funcional para cada estación de reserva y emisión de una instrucción por ciclo.

Supongamos que se ejecuta el siguiente fragmento de código en el nanoMIPS (ejemplo 1.11 del capítulo 1):

```

1 bucle: L.D F1,0(R1)
2 L.D F2,0(R2)
3 MUL.D F10,F1,F2
4 ADD.D F4,F6,F10
5 ADDI R1,R1,#8
6 ADDI R2,R2,#8
7 S.D F4,0(R1)
8 BNE R1,R10,bucle

```

Inicialmente R1=R2=0 y R10=800.

Este nanoMIPS tiene tres ALUs de enteros con latencia 1 ciclo, dos sumadores/restadores en coma flotante con latencia 4 ciclos y 2 multiplicadores/divisores en coma flotante con latencia de 8 ciclos para la multiplicación y 10 para la división.

Ninguna de las unidades en coma flotante está segmentada y se tiene una unidad funcional para cada estación de reserva. Las estaciones de reserva se etiquetan como 1, 2 y 3 las de enteros, 4 y 5 las de suma/resta en coma flotante y 6 y 7 las de multiplicación y división en coma flotante.

Por simplicidad, se supone de nuevo que las instrucciones de carga y almacenamiento siguen el mismo camino que las instrucciones de enteros, utilizando las mismas ALUs y con 1 ciclo de latencia.

En esta arquitectura se utiliza una estructura de tipo buffer de predicción con BTB para realizar la predicción de los saltos, y en el caso de este código, la predicción es que el salto se toma en todas las ocasiones. Además el ROB tiene 10 entradas. Este sería el diagrama de ejecución hasta que se resuelva la primera especulación realizada:

| Instrucción      | E                | X                     | W                | C  |
|------------------|------------------|-----------------------|------------------|----|
| L.D F1,0(R1)     | 1                | 2                     | 3                | 4  |
| L.D F2,0(R2)     | 2                | 3                     | 4                | 5  |
| MUL.D F10,F1,F2  | 3                | 5 <sup>RAW</sup> -12  | 13               | 14 |
| ADD.D F4,F6,F10  | 4                | 14 <sup>RAW</sup> -17 | 18               | 19 |
| ADDI R1,R1,#8    | 5                | 6                     | 7                | 20 |
| ADDI R2,R2,#8    | 6                | 7                     | 8                | 21 |
| S.D F4,0(R1)     | 7                | 19 <sup>RAW</sup>     | 20               | 22 |
| BNE R1,R10,bucle | 8                | 9                     | 10               | 23 |
| L.D F1,0(R1)     | 9                | 10                    | 11               |    |
| L.D F2,0(R2)     | 11 <sup>RE</sup> | 12                    | 14 <sup>RE</sup> |    |
| MUL.D F10,F1,F2  | 12               | 15 <sup>RAW</sup> -22 |                  |    |
| ADD.D F4,F6,F10  | 13               |                       |                  |    |
| ADDI R1,R1,#8    | 15 <sup>RE</sup> | 16                    | 17               |    |
| ADDI R2,R2,#8    | 20 <sup>RE</sup> | 21                    |                  |    |
| S.D F4,0(R1)     | 21               |                       |                  |    |
| BNE R1,R10,bucle | 22               |                       |                  |    |

Este sería el diagrama temporal de ejecución de las instrucciones sin tener en cuenta la etapa de búsqueda de instrucción que las deja almacenadas en la cola de instrucciones. Se puede observar que:

- La planificación se realiza exactamente igual que con el algoritmo de Tomasulo pero cuando se llega a una instrucción de salto, se puede tener en cuenta la predicción para ese salto y seguir ejecutando instrucciones siempre y cuando los riesgos estructurales lo permitan.
- La etapa de terminación o commit debe hacerse en orden. Por ejemplo, la instrucción ADDI R1,R1,#8 escribe sus resultados en el CDB en el ciclo 7 pero no puede terminar hasta el ciclo 20 (que es cuando libera su entrada en el ROB).
- La instrucción L.D F2,0(R2) no puede emitirse en el ciclo 10 porque no tiene libre una estación de reserva para enteros (sólo hay tres y las tres están ocupadas en ese ciclo).
- La instrucción ADDI no se puede emitir en el ciclo 14 porque hay un riesgo estructural por el ROB, ya no quedan entradas libres. Por lo tanto, hay que esperar hasta que queda una libre en el ciclo 15 para poder continuar con la emisión. Lo mismo ocurre con la siguiente instrucción ADDI, que tiene que esperar a que quede una entrada libre en el ROB en el ciclo 20 para poder emitirse.
- Las entradas sombreadas del diagrama de ejecución corresponden a instrucciones especuladas, que se planifican sin saber seguro si deben ejecutarse o no. La frontera para la especulación está en el ciclo 23,

cuando la primera instrucción de salto termina y se sabe realmente si las instrucciones que se han ejecutado según la predicción para este salto deben seguir avanzando o por el contrario, si deben desecharse porque la predicción estaba equivocada.

- El efecto de utilizar especulación con este bucle es casi el mismo que el de hacer un desenrollado software, sólo que en este caso, es el propio procesador el que se encarga de realizar toda la gestión desde el hardware. Se puede encontrar en un determinado ciclo la misma instrucción de distintas iteraciones del bucle ejecutándose en el procesador.

Y si por ejemplo, nos preguntamos por el estado de las estaciones de reserva, el ROB y del control de escrituras al comenzar el ciclo 19, tendríamos:

| Etiqueta | Valor s1      | Entrada ROB s1 | Valor s2      | Entrada ROB s2 | Entrada ROB |
|----------|---------------|----------------|---------------|----------------|-------------|
| 1        |               |                |               |                |             |
| 2        |               |                |               |                |             |
| 3        | [CDB]ciclo 18 |                | [R1]          |                | 7           |
| 4        |               |                |               |                |             |
| 5        | [F6]          |                | —             | 1              | 2           |
| 6        |               |                |               |                |             |
| 7        | [F1]          |                | [CDB]ciclo 14 |                | 1           |

| F0 | F1 | F2 | F3 | F4 | .. | F10 |  |  | Reg         |
|----|----|----|----|----|----|-----|--|--|-------------|
|    | SÍ | SÍ |    | SÍ |    | SÍ  |  |  | Ocup        |
|    | 9  | 10 |    | 2  |    | 1   |  |  | Entrada ROB |

| Entrada | Etiqueta estación de reserva | UF       | Etapa completada | Destino | DIR | Valor del resultado |
|---------|------------------------------|----------|------------------|---------|-----|---------------------|
| 1       | 7                            | Mult/Div | E                | F10     | —   | —                   |
| 2       | 5                            | Add/Sub  | E                | F4      | —   | —                   |
| 3       | —                            | —        | W                | R1      | —   | ##                  |
| 4       | —                            | —        | W                | F4      | —   | ##                  |
| 5       | —                            | —        | W                | R1      | —   | ##                  |
| 6       | —                            | —        | W                | R2      | —   | ##                  |
| 7       | 3                            | ALU      | E                | —       | @@  | —                   |
| 8       | —                            | —        | W                | —       | —   | ##                  |
| 9       | —                            | —        | W                | F1      | @@  | ##                  |
| 10      | —                            | —        | W                | F2      | @@  | ##                  |

Donde @@ y ## expresan valores numéricos concretos que son las direcciones de acceso a memoria y los resultados de operaciones aritmético-lógicas respectivamente.

**Ejemplo 3.13**

Planificación dinámica de instrucciones con el algoritmo de Tomasulo especulativo en el nanoMIPS con una unidad funcional para cada estación de reserva y emisión de dos instrucciones por ciclo (implementación superescalar).

Repitamos la ejecución del código del ejemplo anterior en un diseño más eficiente que pueda emitir dos instrucciones por ciclo, siempre y cuando no existan riesgos entre ellas, y que además comparta la misma unidad funcional para varias estaciones de reserva. Hay que tener en cuenta que en este diseño, además de poder emitir dos instrucciones por ciclo, el CDB admitirá dos escrituras por ciclo, el ROB podrá terminar dos instrucciones por ciclo, etc. Este sería el diagrama de ejecución en este caso:

| Instrucción      | E                | X                     | W  | C  |
|------------------|------------------|-----------------------|----|----|
| L.D F1,0(R1)     | 1                | 2                     | 3  | 4  |
| L.D F2,0(R2)     | 2                | 3                     | 4  | 5  |
| MUL.D F10,F1,F2  | 3                | 5 <sup>RAW</sup> -12  | 13 | 14 |
| ADD.D F4,F6,F10  | 4                | 14 <sup>RAW</sup> -17 | 18 | 19 |
| ADDI R1,R1,#8    | 4                | 5                     | 6  | 19 |
| ADDI R2,R2,#8    | 5                | 6                     | 7  | 20 |
| S.D F4,0(R1)     | 6                | 19 <sup>RAW</sup>     | 20 | 21 |
| BNE R1,R10,bucle | 7                | 8                     | 9  | 22 |
| L.D F1,0(R1)     | 8                | 9                     | 10 |    |
| L.D F2,0(R2)     | 9                | 10                    | 11 |    |
| MUL.D F10,F1,F2  | 10               | 13 <sup>RE</sup> -20  | 21 |    |
| ADD.D F4,F6,F10  | 11               |                       |    |    |
| ADDI R1,R1,#8    | 15 <sup>RE</sup> | 16                    | 17 |    |
| ADDI R2,R2,#8    | 20 <sup>RE</sup> | 21                    |    |    |
| S.D F4,0(R1)     | 21               |                       |    |    |

Como se puede observar, el hecho de compartir las unidades funcionales sólo provoca una parada de un ciclo por riesgo estructural entre las dos multiplicaciones en coma flotante (que se podría evitar segmentando las unidades funcionales). Por lo demás, y a pesar de reducir tanto los recursos hardware disponibles, el rendimiento de la ejecución no ha sufrido ningún cambio.

En cuanto a la emisión de dos instrucciones por ciclo, los riesgos estructurales y de datos entre las diferentes instrucciones, han impedido que se pueda aprovechar en la mayor parte de los ciclos. Este es un problema muy común; aunque el hardware incluido en el procesador sea capaz de emitir más de una instrucción por ciclo, las dependencias entre las instrucciones limitan mucho la explotación del paralelismo a nivel de instrucción.

Por último, es necesario estudiar detenidamente la posibilidad de hacer un tratamiento de excepciones preciso en un procesador que utilice especulación.

En general, hay que evitar que las instrucciones especuladas produzcan excepciones que afecten a los resultados del programa y a su comportamiento, ya que mientras están especuladas, no se sabe seguro si tienen que ejecutarse o no.

Existen diferentes alternativas para el tratamiento de excepciones en procesadores especulativos, estos son sólo dos ejemplos:

- Las excepciones que provocan las instrucciones especuladas se comprueban mediante código explícito (normalmente instrucciones de check o test que comprueban las causas típicas de excepción para cada tipo de instrucción). Si estas excepciones provocan la finalización del programa, se ignoran, en caso contrario, se tratan.
- Se añaden bits de veneno (poison bits) a los registros del banco de registros, de manera que si una instrucción especulada provoca una excepción, ésta se ignora pero se pone a 1 el poison bit del registro en el que escribe su resultado. Si otra instrucción tiene como operando fuente este registro, se trata la excepción antes de que el valor envenenado se utilice.

En el caso del nanoMIPS el tratamiento de excepciones se realiza gracias al buffer de reordenamiento, utilizando una técnica similar a la del vector de estados que se explicó en el capítulo 1 y aprovechando que las instrucciones están obligadas a realizar el commit en orden. Para ello se añade nueva información a cada entrada del ROB, que indica si la instrucción ha provocado una excepción en alguna de sus etapas, de la misma manera que se almacenaba esta información en el vector de estados que acompañaba a la instrucción por la ruta de datos.

## 3.5 Multithreading

Las técnicas de diseño estudiadas hasta el momento se limitan a buscar el paralelismo entre instrucciones en un único hilo o thread de ejecución. Pero si no es posible explotar el paralelismo a nivel de instrucción (ILP) dentro de este thread, la ganancia obtenida utilizando estas técnicas será muy pobre, a pesar de su gran consumo en recursos hardware y en potencia.

Una posible solución es el multithreading, que permite explotar el paralelismo entre instrucciones que pertenecen a diferentes hilos de ejecución. Es decir, se va un paso más allá intentando aprovechar el paralelismo a nivel de hilo (Thread level parallelism, TLP) además del ILP. Hay que recordar que un thread o hilo de ejecución tiene su propia pila y contador de programa pero comparte el código, las variables globales y los archivos abiertos con el resto de threads del mismo proceso. Por eso los cambios de contexto entre threads son más sencillos. En general, los threads son más ligeros que los procesos, y se pueden crear y destruir más rápidamente.

Esta técnica de multithreading suele combinarse con todas las estudiadas anteriormente, mejorando así el rendimiento de los procesadores superescalares con especulación. La principal diferencia es que para utilizar esta técnica se involucra directamente al sistema operativo y al compilador.

El multihtreading consiste en ejecutar al mismo tiempo dos o más threads de un programa, permitiendo que cada uno de estos threads sea planificado de la manera más conveniente en el procesador, es decir, aprovechando al máximo todos los recursos disponibles. Es equivalente a tener dos o más procesadores lógicos o virtuales en lugar de uno sólo (figura 3.21).

De momento la mayor parte de los diseños que ha incorporado esta técnica han permitido ejecutar simultáneamente instrucciones de 2 threads diferentes. El SMT (Symmetric MultiThreading) o Hyper-threading consiste en permitir que se emitan en el mismo ciclo instrucciones que pertenecen a dos hilos de ejecución o threads diferentes. Por el contrario, el CMT (Coarse MultiThreading) no permite que se lancen instrucciones de threads diferentes en el mismo ciclo. En cada ciclo o son de un thread o son de otro. Es decir, aunque se ejecuten varios threads en paralelo, lo hacen turnándose en la emisión de ins-

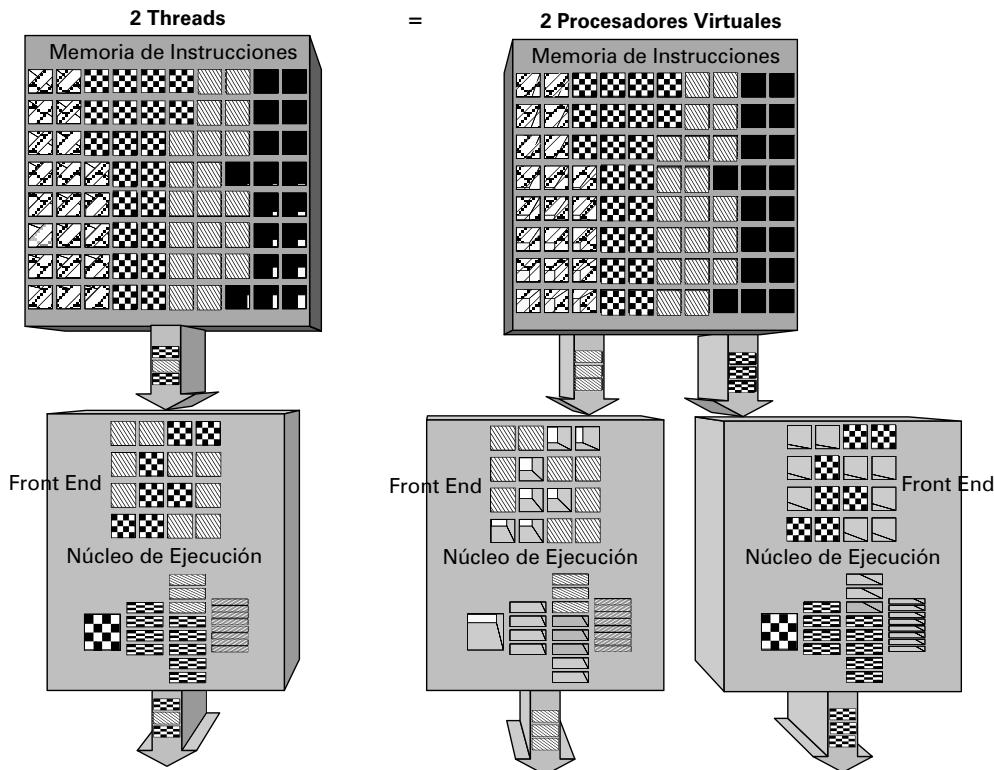


FIGURA 3.21

Multithreading.

trucciones en cada ciclo. Esta técnica evita tener que recompilar las aplicaciones cuando se ejecutan en procesadores diferentes (figura 3.22).

En cualquier caso, ¿cómo comparten los diferentes threads los recursos del procesador? Existen tres tipos de recursos:

- **Recurso replicado.** Cada thread tiene el suyo propio, es el caso del contador de programa, los registros de control o la pila.
- **Recurso repartido.** Existe un único recurso que se divide de manera estática entre todos los threads, es decir, inicialmente se establece un reparto que no se puede modificar en tiempo de ejecución. Suele ser el caso del buffer de reordenamiento, la cola de planificación de instrucciones, etc, en los que se reparten las entradas para cada thread.
- **Recurso compartido.** Existe un único recurso que se comparte de manera dinámica entre todos los threads según la planificación que se haga en tiempo de ejecución y sus necesidades. Es el caso de las unidades de ejecución, los registros de propósito general y las memorias caché.

La existencia de los recursos compartidos es la que mejora el rendimiento de los procesadores superscalares tradicionales ya que permite aprovechar recursos que de otra manera no se utilizarían. Aunque por otro lado limita la ganancia obtenida ya que realmente sólo se tiene un procesador. Por ejemplo, con 2 threads (2 procesadores virtuales), se puede incrementar el rendimiento del procesador en un 25 o 30%, pero no en un 100%.

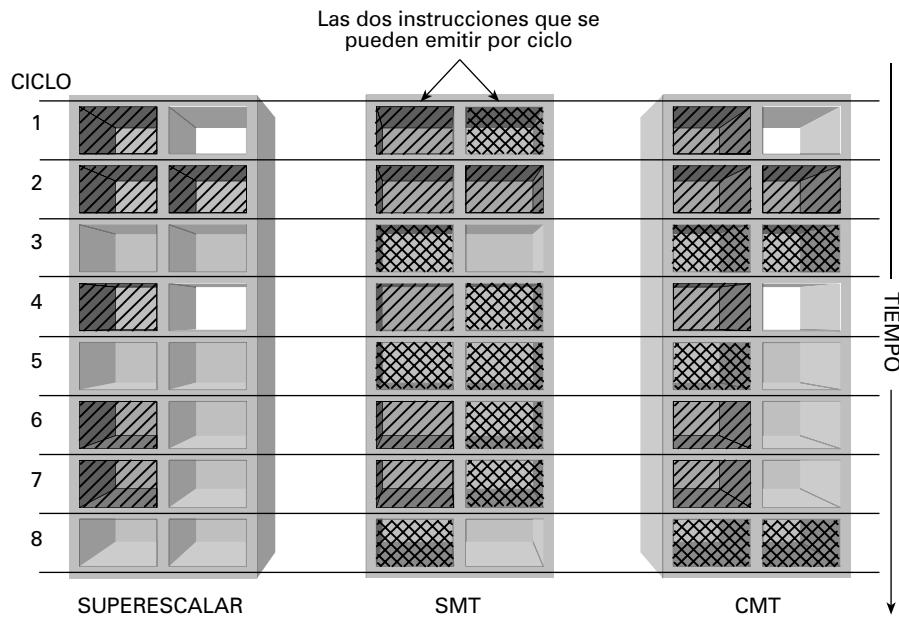


FIGURA 3.22

Diferencias entre un procesador superescalar que emite dos instrucciones por ciclo, un procesador con SMT y uno con CMT.

## Resumen de decisiones de diseño de técnicas de aumento de prestaciones para procesadores

### PLANIFICACIÓN DINÁMICA DE INSTRUCCIONES

| Decisión                                       | Alternativas                           | Decisiones asociadas                                                                                                          |
|------------------------------------------------|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <b>Etapas de ejecución de una instrucción</b>  |                                        |                                                                                                                               |
| <b>Ubicación de la lógica de planificación</b> | Centralizada                           | Tamaño y diseño de la ventana de instrucciones                                                                                |
|                                                | Distribuida                            | Número y diseño de las estaciones de reserva<br>Compartición de las unidades funcionales por diferentes estaciones de reserva |
| <b>Resolución de riesgos WAR y WAW</b>         | Paradas<br>Renombramiento de registros |                                                                                                                               |
| <b>Ejecución de los accesos a memoria</b>      | En orden<br>Fuera de orden             |                                                                                                                               |

### PREDICCIÓN DINÁMICA DE SALTOS

| Decisión                   | Alternativas                               | Decisiones asociadas                                                                                                                                                  |
|----------------------------|--------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Estructura hardware</b> | Buffer de predicción                       | Número de entradas<br>Número de puertos de acceso                                                                                                                     |
|                            | BTB                                        | Número de puertos de acceso<br>Almacenamiento de la dirección destino de salto o de la instrucción destino de salto                                                   |
|                            | Estructuras unificadas u otras estructuras | Buffer de predicción+BTB<br>Índice sucesor en la memoria de instrucciones, pila de dirección de retorno, buffer para la predicción de la confianza de los saltos, etc |
| <b>Tipo de predictor</b>   | Predicción implícita                       |                                                                                                                                                                       |
|                            | Predicción local                           | Número de bits                                                                                                                                                        |
|                            | Predicción global                          | Longitud del GBH y número de bits para la predicción local                                                                                                            |
|                            | Predicción híbrida                         | Algoritmo de fusión de predicciones<br>Utilización de información acerca de la confianza de las predicciones                                                          |
|                            | Predicción adaptativa                      | Número y tipo de predictores posibles<br>Algoritmo de adaptación (o de torneo)                                                                                        |

**EMISIÓN MÚLTIPLE DE INSTRUCCIONES Y ESPECULACIÓN**

| Decisión               | Alternativas              | Decisiones asociadas                                                                                                                                          |
|------------------------|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Tipo de emisión</b> | Estática (VLIW)           | Número de subinstrucciones por VLIW<br>Relación del procesador con el compilador<br>Técnicas SW para aumentar el ILP                                          |
|                        | Dinámica (superescalares) | Planificación estática o dinámica<br>Política de emisión                                                                                                      |
|                        | Dinámica con especulación | Política de terminación o commit<br>Diseño del ROB<br>Mecanismos de resolución de la especulación de saltos y accesos a memoria<br>Tratamiento de excepciones |

**PARALELISMO A NIVEL DE THREAD**

| Decisión                                                                               | Alternativas         | Decisiones asociadas                                                                        |
|----------------------------------------------------------------------------------------|----------------------|---------------------------------------------------------------------------------------------|
| <b>Número de threads que pueden ejecutarse en paralelo</b>                             |                      |                                                                                             |
| <b>Posibilidad de emisión de instrucciones de diferentes threads en el mismo ciclo</b> | Sí (SMT)<br>No (CMT) | En ambos casos: qué recursos se replican, reparten y comparten<br>Políticas de compartición |

**BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS**

- FARGUHAR, E. & BUNCE, P. J. (1994): *The MIPS Programmer's Handbook* (1.<sup>a</sup> ed.), Morgan Kaufmann.
- HENNESSY, J. L. & PATTERSON, D. A. (2007): *Computer Architecture: A Quantitative Approach* (4.<sup>a</sup> ed.), Morgan Kaufmann.
- HWANG, K. (1992): *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw Hill.
- KANE, G. & HEINRINCH, J. (1991): *MIPS RISC Architecture* (2.<sup>a</sup> ed.), Prentice Hall.
- OMONDI, A. R. (1999): *The Microarchitecture of Pipelined and Superscalar Computers* (1.<sup>a</sup> ed.), Springer.
- ORTEGA, J.; ANGUITA, M. & PRIETO, A. (2005): *Arquitectura de Computadores*, Thomson.
- PARHAMI, B. (2007): *Arquitectura de Computadoras. De los microprocesadores a las supercomputadoras* (1.<sup>a</sup> ed.), McGraw Hill.
- SHEN, J. P. & LIPASTI, M. (2006): *Arquitectura de Computadores. Fundamentos de los procesadores superescalares* (1.<sup>a</sup> ed.), McGraw Hill.
- SHIVA, S. G. (2005): *Advanced Computer Architectures*, CRC Press.
- SILC, J.; ROBIC, B. & UNGERER, T. (1999): *Processor Architecture. From Dataflow to Superscalar and Beyond*, Springer.
- SIMA, D.; FOUNTAIN, T. & KARSUK, P. (1997): *Advanced Computer Architectures: A Design Space Approach*, Addison Wesley.

## PROBLEMAS

- 3.1.** Se ejecuta el siguiente fragmento de código en un procesador nanoMIPS que utiliza planificación basada en el algoritmo de la Pizarra (centralizada) y que tiene dos ALUs de enteros (latencia 1 ciclo), dos sumadores/restadores en coma flotante (latencia 4 ciclos), dos multiplicadores en coma flotante (latencia 7 ciclos) y un divisor en coma flotante (latencia 20 ciclos). Ninguna de estas unidades funcionales está segmentada. Mostrar el diagrama de ejecución de este código y el estado de las estructuras hardware utilizadas para la planificación dinámica al terminar el ciclo 16.

|       |           |
|-------|-----------|
| L.D   | F2,0(R1)  |
| MUL.D | F4,F2,F0  |
| DIV.D | F1,F2,F8  |
| ADD.D | F6,F4,F6  |
| S.D   | 0(R2), F6 |
| ADD.D | F7,F0,F6  |
| SUB.D | F6,F5,F3  |

- 3.2.** Mostrar el diagrama de ejecución del mismo código del ejercicio 3.1 con una planificación de Tomasulo con tres estaciones de reserva para operaciones con enteros, tres estaciones de reserva para suma y resta en coma flotante, una estación de reserva para multiplicación en coma flotante y una para división en coma flotante. Suponer que cada estación de reserva tiene su propia unidad funcional, con las latencias especificadas en el ejercicio anterior, y que ninguna de ellas está segmentada. Mostrar además el estado de las estructuras hardware utilizadas para la planificación al terminar el ciclo 11.
- 3.3.** Repetir el ejercicio 3.2 si sólo hay una unidad funcional de cada tipo compartida por las estaciones de reserva, pero todas las unidades funcionales de coma flotante están segmentadas.

- 3.4.** Comparar el diagrama de ejecución del siguiente fragmento de código en los dos siguientes diseños del nanoMIPS:

- Planificación basada en la Pizarra. Cuatro unidades de enteros, dos de suma/resta en coma flotante y dos de multiplicación/división. Latencias de 1, 4 y 9 ciclos respectivamente. Unidades funcionales sin segmentar.
- Planificación basada en Tomasulo. Cuatro estaciones de reserva de enteros, dos de suma/resta en coma flotante y dos de multiplicación/división. Una unidad funcional de cada tipo, todas ellas segmentadas. Mismas latencias que en el apartado a.

|       |            |
|-------|------------|
| MUL.D | F3,F1,F0   |
| DIV.D | F4,F3,F8   |
| DIV.D | F6,F3,F2   |
| ADD.D | F8,F2,F0   |
| ADD.D | F8,F1,F7   |
| SUB.D | F7,F3,F1   |
| MUL.D | F6,F5,F9   |
| S.D   | F6,100(R4) |

- 3.5.** Un procesador está segmentado en 7 etapas: IF, ID, IS, EX1, EX2, MEM y WB. En este procesador las instrucciones de salto condicional calculan la dirección destino de salto en la etapa ID

y evalúan la condición y cargan el nuevo PC en la etapa EX1. Comparar el rendimiento de la predicción estática de salto no tomado con la de un BTB que utiliza un predictor de 1 bit, obteniendo la predicción y la dirección destino de salto al final de la fase ID. Los saltos que no se toman no se almacenan en el BTB. Para realizar la comparación de ambos esquemas de predicción se utiliza una carga típica en la que el 18% de las instrucciones son saltos condicionales. De estas instrucciones el 85% son saltos tomados. Además, la tasa de fallos del BTB es del 16%. Calcular el CPI del procesador con cada uno de los esquemas, suponiendo que el CPI de las instrucciones sin tener en cuenta los riesgos de control es 1.

- 3.6.** Se diseña un procesador segmentado con 6 etapas: F, S, X1, M, X2 y WB. La dirección destino de salto se obtiene en la fase X1 y la condición se evalúa en la etapa X2. Mostrar el diagrama temporal de una instrucción de salto en todos los casos posibles para estas dos alternativas:
- Se utiliza un buffer de predicción de saltos con un predictor local de 2 bits que obtiene la predicción en la fase F. Este buffer tiene una tasa de fallos del 10%.
  - Se utiliza una estructura buffer+BTB que obtiene sus resultados en la fase S con una tasa de fallos del 15% porque se basa en un predictor local de 1 bit.

Para realizar la comparación entre los dos esquemas, calcular el CPI del procesador con cada uno de ellos, suponiendo que el CPI de las instrucciones sin tener en cuenta los riesgos de control es 1. El 14% de las instrucciones son saltos condicionales y de estos saltos, el 75% son saltos tomados.

- 3.7.** Se ejecuta la secuencia de código en un procesador de tipo nanoMIPS:

```
for (i=0;i++;i<5)
 R = A(i) - B(i);
 if R < 0
 c=c++;
```

Comparar el rendimiento de los siguientes tres esquemas de predicción:

- Predicción estática de salto no tomado.
- Predicción estática basada en el desplazamiento del salto: si es negativo, se predice que el salto se toma, si es positivo, se predice que el salto no se toma.
- Predicción dinámica con buffer de predicción+BTB y un predictor de 1 bit, cuyo valor inicial es 0 (suponer que la predicción de la dirección destino de salto se acierta siempre).

Para realizar esta comparación, tener en cuenta que en las tres primeras iteraciones  $A(i) > B(i)$  y en que en las dos siguientes,  $A(i) > B(i)$ . Además, la penalización por salto cuando se acierta la predicción es de 0 ciclos, y cuando se falla es de 3 ciclos.

- 3.8.** Se ejecuta la siguiente secuencia de código en un procesador de tipo MIPS que realiza predicción dinámica de saltos con una estructura de buffer de predicción unida a un BTB:

```
for (i=0;i++;i<5)
 if A(i) ≠ B(i)
 A(i)=B(i);
```

Que traducido al repertorio de instrucciones del procesador queda:

|        |      |               |
|--------|------|---------------|
|        | ADD  | R1,R0,R0      |
|        | ADDI | R2,R0,#20     |
| loop:  | LW   | R5,A(R1)      |
|        | LW   | R6,B(R1)      |
|        | BEQ  | R5, R6, final |
|        | SW   | R6, A(R1)     |
| final: | ADDI | R1,R1,#4      |
|        | BNEQ | R1,R2,loop    |

Los vectores A y B son los siguientes:

$$A = (1 \ 2 \ 3 \ 5)$$

$$B = (1 \ 2 \ 4 \ 9)$$

- a) Si se utiliza un predictor (1,1), con un valor inicial del GBH de 0 (es decir, el último salto que se ha ejecutado no se ha tomado) y partimos de la siguiente situación inicial en el buffer de predicción, explicar cómo se van realizando las predicciones y cómo se va actualizando esta estructura al ejecutar el código propuesto sobre los vectores A y B:

| Etiqueta de salto | Predicción si GBH=0 | Predicción si GBH=1 |
|-------------------|---------------------|---------------------|
| ...               |                     |                     |
| BEQ               | 1                   | 1                   |
| BNEQ              | 1                   | 0                   |
| ...               |                     |                     |

- b) Repetir el apartado anterior si se utiliza un predictor (1,2), con un valor inicial del GBH de 0 y el siguiente estado inicial del buffer:

| Etiqueta de salto | Predicción si GBH=0 | Predicción si GBH=1 |
|-------------------|---------------------|---------------------|
| ...               |                     |                     |
| BEQ               | 10                  | 01                  |
| BNEQ              | 11                  | 01                  |
| ...               |                     |                     |

- c) Comparar las dos opciones estudiadas y escoger una de ellas teniendo en cuenta el tamaño del buffer, la complejidad de gestión y actualización, la tasa de fallos, etc. Justificar adecuadamente la decisión.

### 3.9. Se ejecuta el siguiente fragmento de código:

|       |       |            |
|-------|-------|------------|
| loop: | L.D   | F0,0(R1)   |
|       | ADD.D | F4,F0,F2   |
|       | S.D   | F4,0(R1)   |
|       | ADDI  | R1,R1,#8   |
|       | BNE   | R1,R2,loop |

Si la latencia de la suma en coma flotante es de cuatro ciclos, ¿cuántas instrucciones se emiten por ciclo?

- a) Si se tiene un procesador VLIW que empaqueta dos referencias a memoria, dos operaciones en FP y una operación entera en cada ciclo. Mostrar cómo se empaquetan las instrucciones de este bucle (desenrollando el bucle tantas veces como haga falta) para aprovechar al máximo los recursos del procesador y que no haya ningún ciclo en el que esté completamente desocupado.
- b) Si se ejecuta una iteración del bucle en el nanoMIPS superescalar estático.
- c) Si se ejecuta una iteración del bucle en el nanoMIPS superescalar dinámico con emisión de 4 instrucciones por ciclo y cuatro estaciones de reserva para enteros, cuatro para suma/resta en coma flotante y dos para multiplicación/división.
- 3.10.** Se ejecuta el siguiente código en un nanoMIPS especulativo que dispone de un BTB que obtiene sus predicciones al final de la fase de emisión y una unidad funcional de cada tipo, con todas las unidades funcionales de coma flotante segmentadas. Las latencias de estas unidades funcionales son de 1 ciclo para la ALU de enteros, 4 ciclos para el sumador/restador en FP y 10 ciclos para el multiplicar/divisor en FP. Además, se dispone de un número ilimitado de entradas en el buffer de reordenamiento y de 2 estaciones de reserva para operaciones enteras, 3 para suma y resta en FP y 2 para multiplicación y división en FP.
- |        |                |
|--------|----------------|
| MUL.D  | F2,F4,F8       |
| ADD.D  | F4,F4,F10      |
| SUB.D  | F6,F2,F4       |
| BLTZ   | F6,salto       |
| ADD.D  | F2,F4,F10      |
| MUL.D  | F6,F8,F4       |
| MUL.D  | F6,F6,F2       |
| MUL.D  | F6,F6,F6       |
| SUB.D  | F6,F6,F8       |
| ADD.D  | F6,F6,F6       |
| salto: | SUB.D F2,F6,F4 |
|        | DIV.D F4,F2,F2 |
|        | S.D F4,8(R0)   |
- Si los valores iniciales de los registros son los siguientes:
- | F0 | F2 | F4 | F6 | F8 | F10 |
|----|----|----|----|----|-----|
| 0  | 3  | 1  | 5  | 10 | 2   |
- a) Suponiendo que el BTB predice que la instrucción de salto no salta, mostrar el diagrama de ejecución de este código.
- b) Mostrar el estado de las estaciones de reserva, del buffer de reordenamiento y del control de escrituras al final del ciclo en el que la instrucción SUB.D F2,F6,F4 realiza su terminación.
- 3.11.** Mostrar el diagrama de ejecución del código del ejercicio 3.10 si se rediseña el procesador para permitir emisión múltiple de dos instrucciones por ciclo y además, en un intento de ser más realistas, contamos con un número limitado de entradas en el ROB, en este caso 12 entradas.

## AUTOEVALUACIÓN

1. ¿Cuáles son las diferencias entre la planificación dinámica de instrucciones centralizada y distribuida?
2. ¿Qué es una estación de reserva?
3. ¿Qué diferencias hay entre un buffer de predicción de saltos y un BTB?
4. ¿Por qué no tiene mucho sentido utilizar un buffer de predicción como estructura hardware para la predicción dinámica de saltos en el nanoMIPS?
5. ¿En qué consiste la técnica de Branch Folding?
6. ¿Cómo funciona un predictor de saltos multnivel (2,2)?
7. ¿Cómo se diseña un procesador VLIW? ¿Cuál es el principal objetivo que se persigue?
8. ¿En qué consiste la especulación basada en hardware?
9. ¿Qué es el buffer de reordenamiento?
10. ¿Qué diferencias hay entre el Coarse MultiThreading (CMT) y el Symmetric MultiThreading (SMT)?

# 4

---

## Técnicas de aumento de prestaciones para memoria y E/S

---

### Contenidos

- 4.1. Técnicas de optimización para la memoria caché
- 4.2. Técnicas de optimización para la memoria principal
- 4.3. Técnicas de optimización conjunta para todos los niveles de la jerarquía: visión global de la jerarquía de memoria
- 4.4. Técnicas de optimización para los buses de E/S
- 4.5. Técnicas de optimización para los dispositivos de E/S
- 4.6. Técnicas de optimización para la gestión de E/S

Como se ha estudiado en el capítulo 2, la introducción de una jerarquía de memoria en prácticamente todas las arquitecturas de computadoras actuales pretende conseguir una estructura de memoria con un coste por byte de información casi tan bajo como el de la memoria más barata de la jerarquía, pero con latencia comparable a la del nivel más rápido. Para ello la memoria se organiza en niveles, cuanto más cercanos al procesador, más pequeños, rápidos y caros.

Si el procesador que accede a esta memoria incorpora optimizaciones como las estudiadas en el capítulo 3, es muy posible que los accesos a memoria se conviertan en el cuello de botella del sistema, ya que esta memoria no es capaz de alimentar de instrucciones y/o de datos a un procesador tan potente a un ritmo suficiente.

La latencia de una memoria suele estar limitada por la tecnología utilizada para fabricarla. Por este motivo las mejoras en la latencia de una memoria suelen implicar mejoras o cambios en la tecnología. Sin embargo, el ancho de banda suele estar limitado por el coste, por lo que sí que es posible mejorarla mediante técnicas de diseño optimizadas.

En este capítulo se presentan estas técnicas de diseño, para la memoria caché y la memoria principal, que permiten sacar el máximo partido de la jerarquía de memoria de un sistema, y que aplicadas a una memoria básica como la presentada en el capítulo 2 de este libro, conseguirán el rendimiento deseado para la arquitectura.

De la misma forma, en este capítulo se estudian las técnicas que permiten que el sistema de E/S esté al mismo nivel de rendimiento que un procesador y una jerarquía de memoria optimizados.

Para obtener el rendimiento deseado del sistema de E/S hay que tener en cuenta, no sólo el funcionamiento de los dispositivos de E/S, sino también la forma en la que estos dispositivos se conectan a la computadora y la forma en la que se gestionan las transferencias de información. Por eso se dividen las técnicas de optimización del sistema de E/S en tres grandes bloques: las que permiten mejorar el rendimiento de los buses incluidos en este sistema, las que permiten mejorar las prestaciones de los dispositivos de E/S y por último, las que permiten optimizar la gestión del sistema de E/S.

## 4.1 Técnicas de optimización para la memoria caché

Si se analiza la expresión del tiempo medio de acceso a memoria propuesta en el capítulo 2:

$$\bar{t} = t_{\text{accesoMC}} + TF \cdot pF$$

A partir de esta expresión parece obvio que cualquier optimización que pretenda mejorar el rendimiento de la memoria caché debe ir orientada a reducir alguno de estos tres aspectos: la penalización por fallo, la tasa de fallos o el tiempo de acceso a la memoria caché.

### 4.1.1. Reducción de la penalización por fallo

Como se estudió en el capítulo 2, prácticamente todas las jerarquías de memoria actuales incorporan más de un nivel de memoria caché para reducir en todo lo posible la penalización por fallo.

De esta forma, cuando el primer nivel de memoria caché, que es el más cercano al procesador, tenga un fallo, la resolución de este fallo no obligará a salir del chip y buscar y traer un bloque completo desde la memoria principal, sino que todavía existirá la posibilidad de resolver el fallo desde otro nivel de la memoria caché, con una penalización por fallo mucho menor.

Esta técnica de diseño basada en la utilización de cachés multinivel hoy en día no se considera una optimización, ya que se incorpora por defecto en casi todas las arquitecturas, al igual que ocurría con técnicas como la de la palabra crítica primro o rearranque rápido.

Sin embargo existen otras técnicas más específicas que pueden ayudar a reducir todavía más esta penalización por fallo.

#### 4.1.1.1. FALLOS PRIORITARIOS

Esta técnica permite resolver los fallos sin que se hayan resuelto todas las escrituras pendientes antes, ya que éstas son muy costosas y aumentarían mucho la penalización por fallo.

En el caso de una caché con escritura directa, los problemas surgen cuando se utiliza un buffer de escritura, ya que la palabra que ha producido el fallo puede no estar actualizada en el siguiente nivel de memoria pero sí en este buffer.

La solución más sencilla sería esperar a que se vaciara el buffer de escritura para resolver el fallo. Así se aseguraría que el siguiente nivel de la jerarquía de memoria está actualizado y resuelve el fallo correctamente.

Pero esto aumenta mucho la penalización por fallo, ya que habría que esperar un volcado completo del buffer antes de poder traer el bloque necesario. Para dar prioridad al fallo, primero se comprueba si la palabra que ha producido el fallo se encuentra en el buffer. Si no es así, se continúa resolviendo el fallo sin necesidad de realizar el volcado del buffer.

Si la palabra que ha provocado el fallo se encuentra en el buffer, puede diseñarse la jerarquía de diferentes maneras. Por ejemplo, puede actualizarse en el siguiente nivel sólo esta palabra antes de resolver el fallo. Pero si hay localidad espacial en los códigos, puede que esto resuelva el problema sólo temporalmente, porque a continuación se pediría otra palabra del mismo bloque. Por eso suele volcarse todo el buffer en el siguiente nivel antes de resolver el fallo.

En el caso de la caché con post-escritura el problema surge cuando al producirse el fallo se debe reemplazar un bloque sucio. Esto implica llevar el bloque sucio al siguiente nivel antes de traer a la caché el necesario para resolver el fallo.

En este caso es el propio buffer de escritura el que ayuda a dar prioridad a los fallos, ya que el bloque sucio se escribe en este buffer (lo que es mucho más rápido que escribirlo en el siguiente nivel), luego se trae el bloque que se necesita para resolver el fallo, y el procesador ya puede comenzar a trabajar. Es decir, es necesario utilizar un buffer de escritura para implementar esta técnica en el caso de las cachés con post-escritura. Cuando el buffer se llene, o cuando se necesite alguno de sus bloques para resolver un fallo, o en tiempos de inactividad, se vuelca el contenido del buffer en el siguiente nivel para que la jerarquía quede actualizada.

Obviamente, aunque la técnica se suela relacionar con los fallos de lectura (a veces incluso se denomiña dar prioridad a los fallos de lectura), en una caché de post-escritura con asignación en escritura (que es lo más habitual), también se proporciona la misma prioridad a los fallos de escritura, que se resuelven exactamente igual que los de lectura.

#### Ejemplo 4.1

##### Ejemplo de fallos prioritarios para la reducción de la penalización por fallo.

Utilizamos de nuevo el ejemplo 2.6 de la jerarquía de memoria del capítulo 2, en este caso, con la caché de post-escritura. Sin ninguna optimización teníamos el mismo comportamiento para las lecturas y las escrituras:

$$t_{MEM} = t_{acíerto} + TF \cdot pF$$

Con una penalización por fallo, suponiendo que se pueden solapar los accesos a memoria principal con las transferencias de información por el bus:

$$pF = 16 \cdot (\text{latencia}_{MP}) + \%sucios \cdot 16 \cdot (\text{latencia}_{MP}) = 16 \cdot (1 + \%sucios) \cdot \text{latencia}_{MP} = 16 \cdot 1.1 \cdot 85 = 1496 \text{ ns}$$

Por lo que:

$$t_{MEM} = t_{acuerdo} + TF \cdot pF = 4 + 0.12 \cdot 1496 = 183.52 \text{ ns}$$

Si suponemos que se utiliza palabra crítica primero y sabiendo que sólo el 10% de los bloques son modificados durante su permanencia en la memoria caché, tenemos:

$$pF = 1 \cdot (latencia_{MP} + t_{bus}) + \%sucios \cdot 16 \cdot (latencia_{MP}) = 1 \cdot (85 + 0.5) + 0.1 \cdot 16 \cdot 85 = 221.5 \text{ ns}$$

Por lo que:

$$t_{MEM} = t_{acuerdo} + TF \cdot pF = 4 + 0.12 \cdot 221.5 = 30.58 \text{ ns}$$

Ahora, para dar prioridad a los fallos, incluimos en esta jerarquía de memoria un buffer de escritura en el que podemos volcar los bloques sucios cuando tienen que ser reemplazados para resolver un fallo de caché. Volcar un bloque sucio completo a este buffer supone una penalización de 15 ns ( $t_{volcado BE}$ ) y además este buffer de escritura permite el volcado de los bloques sucios un 95% de las ocasiones. Es decir, el 5% de las veces que se intenta volcar un bloque sucio a este buffer para dar prioridad a los fallos, el buffer está lleno u ocupado con tareas de gestión, por lo que el volcado debe hacerse a memoria principal.

La penalización por fallo es en este caso:

$$pF = 1 \cdot (latencia_{MP} + t_{bus}) + \%sucios \cdot ((1 - TF_{BE}) \cdot t_{volcado BE} + TF_{BE} \cdot 16 \cdot (latencia_{MP})) = 1 \cdot (85 + 0.5) + 0.1 \cdot (0.95 \cdot 15 + 0.05 \cdot 16 \cdot 85) = 93.72 \text{ ns}$$

Por lo que:

$$t_{MEM} = t_{acuerdo} + TF \cdot pF = 4 + 0.12 \cdot 93.72 = 15.24 \text{ ns}$$

Es decir, conseguimos un speedup de 12.04 gracias a la utilización de las técnicas de palabra crítica primero y de fallos prioritarios ( $183.52/15.24 = 12.04$ ).

#### 4.1.1.2. OPTIMIZACIONES DEL BUFFER DE ESCRITURA

Cuando un buffer de escritura se llena, sea en una caché de escritura directa o en una caché de post-escritura, se debe volcar su contenido al siguiente nivel de la jerarquía de memoria.

En muchos casos, este volcado del contenido del buffer se suma a la penalización por fallo, por lo que cualquier optimización en la utilización del buffer puede hacer que esta penalización disminuya.

Un ejemplo claro son las técnicas de write-merging, que ayudan a optimizar la utilización del buffer de escritura en el caso de las cachés de escritura directa, que van escribiendo en él palabra a palabra.

Un buffer de escritura se organiza como una memoria caché. Por lo tanto está dividido en marcos, y cada uno de ellos lleva asociada una etiqueta que indica qué bloque está ubicado en ese marco en un momento concreto.

Si el buffer de escritura no está optimizado, cada escritura del procesador en el buffer ocupará la primera palabra de un marco del buffer. Pero el resto de palabras del marco quedarán desocupadas.

Por lo tanto, se desaprovechará mucho espacio del buffer, que se llenará mucho más rápido y obligará a hacer volcados con más frecuencia de la realmente necesaria.

Sin embargo, la localidad espacial hace que en muchos casos las escrituras se hagan a palabras consecutivas de un mismo bloque. La técnica del write-merging permite controlar si esto es así. De manera que cuando se realizan escrituras consecutivas a palabras de un mismo bloque, en el buffer de escritura se pueden escribir consecutivamente en el mismo marco (figura 4.1).

Esto permite aprovechar mejor el espacio disponible y reducir el número de volcados del buffer.

#### 4.1.1.3. CACHÉ DE VÍCTIMAS

Esta caché es una memoria pequeña y completamente asociativa en la que se almacenan los últimos bloques reemplazados por si se puede aprovechar la localidad temporal (figura 4.2).

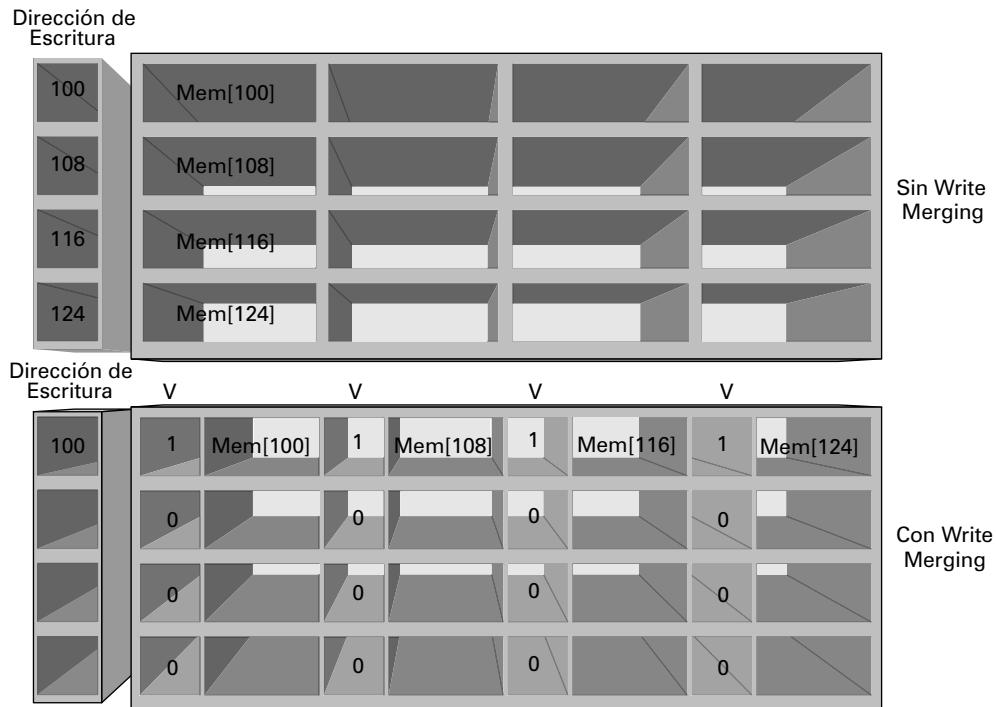


FIGURA 4.1

Optimización del buffer de escritura con write-merging.

De esta manera, cuando se produce un fallo, antes de ir al siguiente nivel de la jerarquía de memoria a buscar el bloque necesario, se busca en la caché de víctimas. Si el bloque se encuentra allí, se reemplazará mucho más rápido, por tratarse de una memoria pequeña y rápida. Y la localidad temporal hace que sea probable que el bloque buscado se haya reemplazado recientemente y esté todavía en la caché de víctimas.

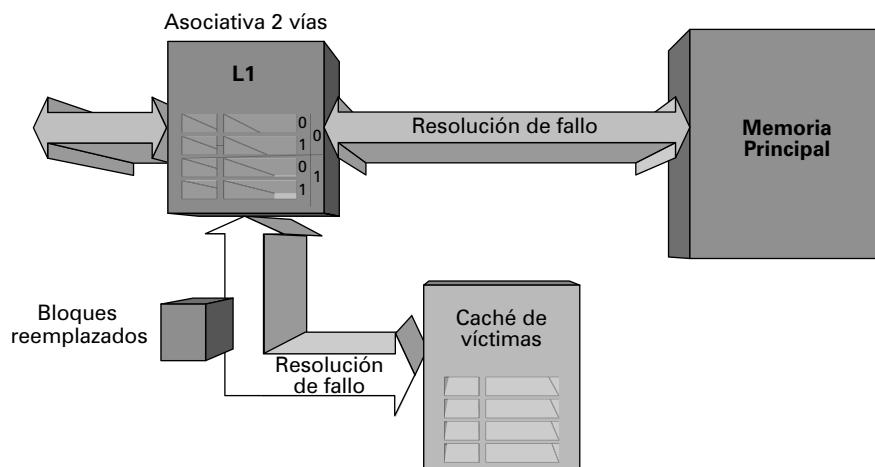


FIGURA 4.2

Caché de víctimas.

Esta técnica reduce considerablemente la penalización por fallo en casi todas las arquitecturas, aunque esta mejora es crítica en los casos en los que se producen fallos por conflicto en cachés conemplazamiento directo.

### Ejemplo 4.2

#### Ejemplo de utilización de una caché de víctimas para la reducción de la penalización por fallo.

Sigamos optimizando la jerarquía de memoria del ejemplo 4.1. Supongamos que estamos trabajando con la caché de post-escritura, a la que, recordemos, ya habíamos incorporado las técnicas de palabra crítica primero y de fallos prioritarios hasta conseguir un tiempo medio de acceso de memoria de 15.24 ns.

Vamos a incorporar a esta jerarquía una caché de víctimas, es decir, una memoria caché completamente asociativa que almacene los últimos bloques reemplazados en la memoria caché. Esta nueva estructura HW consigue resolver los fallos de memoria caché en un 28% de las ocasiones (es decir, el 72% de las veces, sigue siendo necesario acudir a la memoria principal para resolver el fallo), y cuando lo hace, tarda 19 ns en resolver el fallo ( $t_{CV}$ ).

En este caso, la penalización por fallo quedaría:

$$pF = (1 - TF_{CV}) \cdot t_{CV} + TF_{CV} \cdot (\text{latencia}_{MP} + t_{bus}) + \%sucios \cdot ((1 - TF_{BE}) \cdot t_{volcado\ BE} + TF_{BE} \cdot 16 \cdot (\text{latencia}_{MP})) = \\ = 0.28 \cdot 19 + 0.72 \cdot (85 + 0.5) + 0.1 \cdot (0.95 \cdot 15 + 0.05 \cdot 16 \cdot 85) = 75.10 \text{ ns}$$

Por lo que:

$$t_{MEM} = t_{acuerdo} + TF \cdot pF = 4 + 0.12 \cdot 75.10 = 13.01 \text{ ns}$$

Es decir, conseguimos un speedup de 14.11 gracias a la utilización de las técnicas de palabra crítica primero, de fallos prioritarios y de caché de víctimas ( $183.52/13.01 = 14.11$ ).

#### 4.1.1.4. CACHÉ NO BLOQUEANTE

Hay que tener en cuenta que en el caso de procesadores segmentados que permitan terminación fuera de orden, una instrucción puede pararse debido a un fallo de memoria, pero esto no implica que se paren las siguientes.

Para que se puedan seguir ejecutando instrucciones que no producen fallos de memoria, es necesario que la memoria caché sea no bloqueante, es decir, que pueda atender a los accesos a memoria que son aciertos mientras se resuelve el fallo de una instrucción.

Cuando se utiliza este tipo de memoria, la evaluación del rendimiento del procesador es más complicada, ya que un fallo de memoria no implica necesariamente una parada. Se debe tener en cuenta cuánto tiempo del que la memoria caché ha tardado en resolver el fallo ha estado parado el procesador realmente y cuánto de este tiempo ha quedado oculto porque se ha solapado con la ejecución de otras instrucciones.

Es decir, esta técnica no reduce realmente la penalización por fallo, que sigue siendo la misma, pero sí que permite realizar trabajo útil durante parte de esta penalización, por lo que la penalización efectiva que se percibe, puede llegar a ser mucho menor.

#### 4.1.2. Reducción de la tasa de fallos

En el capítulo 2 se estudió cómo el aumento del tamaño de la memoria caché, del tamaño de bloque y de la asociatividad puede reducir la tasa de fallos de la memoria caché. Aunque también se analizó que en todos los casos estas mejoras implicaban al mismo tiempo que la memoria caché fuera más lenta o que la penalización por fallo fuera mayor.

A continuación se presentan técnicas específicas para reducir la tasa de fallos más allá de estas normas básicas de diseño a la hora de escoger la mejor organización para la memoria caché.

#### 4.1.2.1. CACHÉ PREDICTIVA

Este tipo de memoria caché intenta mantener la velocidad de una caché directa, pero introduciendo algún tipo de asociatividad para reducir los fallos por conflicto típicos de estas memorias. Las cachés predictivas suelen ser asociativas por conjuntos en las que siempre se busca primero en uno de los marcos del conjunto. Es decir, se predice en qué marco estará el bloque que se busca.

Esta predicción se realiza con algún tipo de operación con los bits que componen la dirección del bloque buscado. Si esta predicción falla, se busca en el resto de marcos del conjunto, ya que al ser la memoria asociativa por conjuntos, el bloque podría estar ubicado en cualquiera de ellos.

De esta manera, si se acierta con la predicción inicial, la caché se ha comportado como si fuera directa en cuanto al tiempo de acceso (sólo se ha realizado la comparación de una etiqueta) pero, si esta predicción falla, se ha comportado como una memoria asociativa por conjuntos en cuanto a la tasa de fallos (figura 4.3).

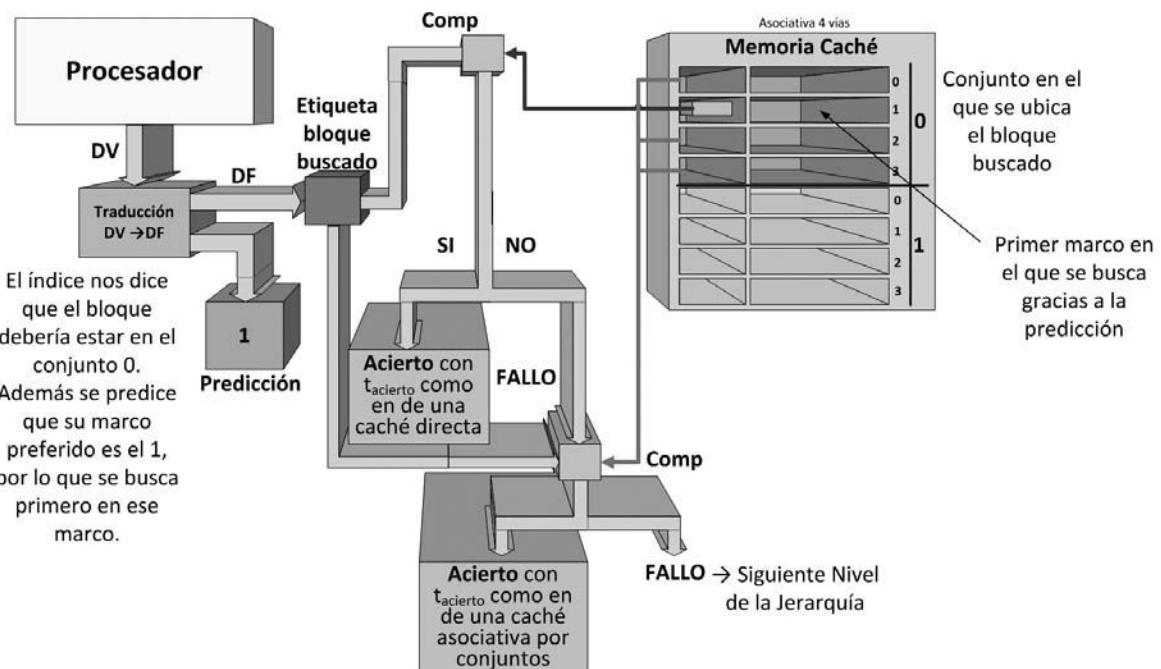


FIGURA 4.3

Caché predictiva.

#### 4.1.2.2. CACHÉ PSEUDOSOACIATIVA

Este tipo de memoria de nuevo intenta aprovechar las ventajas de una caché directa en cuanto al tiempo de acceso y de una caché con más asociatividad en cuanto a la tasa de fallos.

En el caso de una caché pseudoasociativa, la memoria es directa. Es decir, cada bloque de información puede ubicarse en un único marco de la memoria caché. Pero cuando se produce un fallo, antes de buscar el bloque en el siguiente nivel de memoria, se busca el bloque en otro marco de la caché (en su segundo posible emplazamiento).

Por ejemplo, se invierte el bit más significativo del campo índice de la dirección y se busca en ese marco. Es decir, en cuanto a la tasa de fallos es como si la memoria fuera asociativa por conjuntos de 2 vías, ya que siempre existen dos posibilidades para ubicar un bloque dentro de la caché.

Pero en cuanto al tiempo de acceso, como primero se compara una única etiqueta, en los aciertos es una caché completamente directa (figura 4.4).

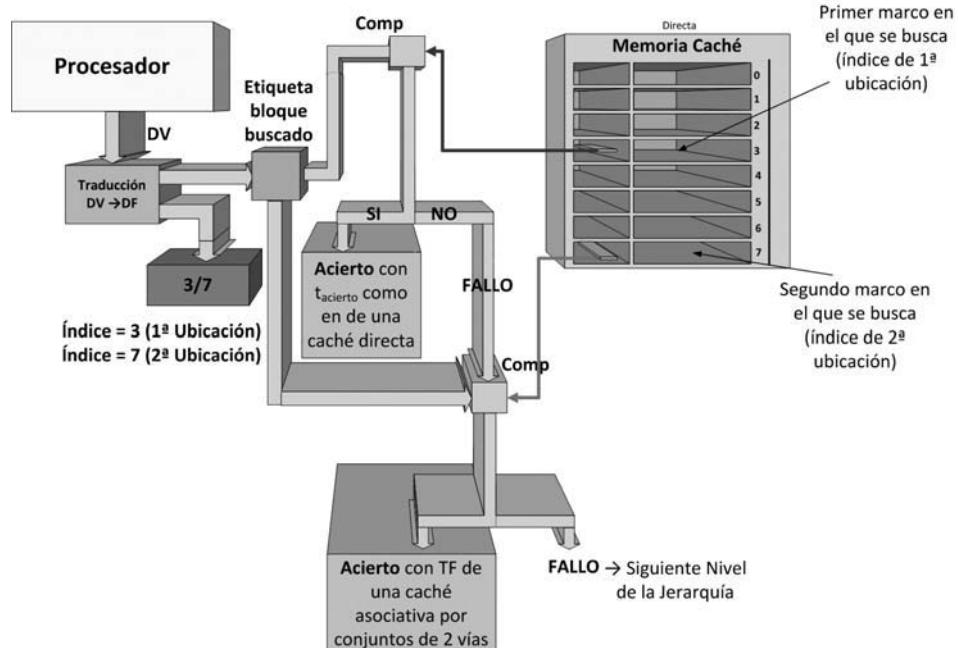


FIGURA 4.4

Caché pseudoasociativa.

**Ejemplo 4.3****Ejemplo de caché predictiva.**

Comencemos ahora a introducir en la jerarquía técnicas que nos permitan reducir la tasa de fallos de la memoria caché.

Hasta ahora no nos habíamos preocupado por la organización de la caché que se incluía en la jerarquía de memoria. Se trata de una memoria con emplazamiento directo, ya sabemos que con tiempo de acceso de 4 ns y tasa de fallos de un 12%.

Esta memoria puede convertirse en una caché predictiva, diseñándola como una memoria asociativa por conjuntos de 4 vías. De esta forma, cuando se busca un bloque en la caché, primero se busca en un marco concreto del conjunto que le corresponde. Esta predicción se realiza con los dos bits últimos de la etiqueta (justo antes del índice).

La predicción acierta en un 60% de los casos, por lo que en todos estos casos se realiza una única comparación de etiqueta que permite mantener el tiempo de acceso de 4 ns. En los casos restantes, es necesario realizar tres comparaciones de etiqueta más, por lo que finalmente el tiempo de acceso es de 5 ns. Pero gracias a este aumento de asociatividad, que permite pasar de una caché directa a una asociativa por conjuntos de 4 vías, la tasa de fallos se reduce a un 9%.

Por lo tanto tenemos exactamente la misma penalización por fallo que en el último ejemplo:

$$pF = (1 - TF_{CV}) \cdot t_{CV} + TF_{CV} \cdot (latencia_{MP} + t_{bus}) + \%sucios \cdot ((1 - TF_{BE}) \cdot t_{volcado\ BE} + TF_{BE} \cdot 16 \cdot (latencia_{MP})) = \\ = 0.28 \cdot 19 + 0.72 \cdot (85 + 0.5) + 0.1 \cdot (0.95 \cdot 15 + 0.05 \cdot 16 \cdot 85) = 75.10 \text{ ns}$$

Pero en este caso se modifican los tiempo de acierto y la tasa de fallos, por lo que:

$$t_{MEM} = t_{acierto} + TF \cdot pF = (0.6 \cdot 4 + 0.4 \cdot 5) + 0.09 \cdot 75.10 = 11.16 \text{ ns}$$

Es decir, conseguimos un speedup de 16.44 gracias a la utilización de las técnicas de palabra crítica primero, de fallos prioritarios, caché de víctimas y caché predictiva ( $183.52/11.16=16.44$ ).

#### Ejemplo 4.4

##### Ejemplo de caché pseudoasociativa.

En este ejemplo vamos a utilizar una caché pseudoasociativa en lugar de una predictiva.

De esta manera, cuando se busca un bloque en la caché, primero se busca en un marco concreto porque se comporta como una caché directa. Si la caché acierta, aquí termina la búsqueda y esto ocurre un 88% de las veces. Si falla, se invierten todos los bits del índice y se busca en la localización que indica este índice invertido. Es decir, hay una segunda comparación de etiqueta porque es posible una segunda ubicación para el bloque dentro de la caché.

El tiempo de acceso sigue siendo de 4 ns cuando se realiza una única comparación (como cuando la caché es directa). En los casos restantes, es necesario realizar una comparación más, por lo que finalmente el tiempo de acceso es de 4.5 ns. Pero gracias a este aumento de asociatividad, que permite pasar de una caché directa a una asociativa por conjuntos de 2 vías (hay dos ubicaciones posibles para cada bloque), la tasa de fallos se reduce a un 10%.

Por lo tanto tenemos exactamente la misma penalización por fallo que en el último ejemplo:

$$pF = (1 - TF_{CV}) \cdot t_{CV} + TF_{CV} \cdot (latencia_{MP} + t_{bus}) + \%sucios \cdot ((1 - TF_{BE}) \cdot t_{volcado\ BE} + TF_{BE} \cdot 16 \cdot (latencia_{MP})) = \\ = 0.28 \cdot 19 + 0.72 \cdot (85 + 0.5) + 0.1 \cdot (0.95 \cdot 15 + 0.05 \cdot 16 \cdot 85) = 75.10 \text{ ns}$$

Pero de nuevo se modifican los tiempos de acierto y la tasa de fallos, por lo que:

$$t_{MEM} = t_{acierto} + TF \cdot pF = (0.88 \cdot 4 + 0.12 \cdot 4.5) + 0.10 \cdot 75.10 = 11.57 \text{ ns}$$

Es decir, conseguimos un speedup de 15.86 gracias a la utilización de las técnicas de palabra crítica primero, de fallos prioritarios, caché de víctimas y caché pseudoasociativa ( $183.52/11.57 = 15.86$ ).

Comparando este ejemplo con el realizado para la caché predictiva, se observa que la tasa de fallos se mejora en mayor medida con la caché predictiva que con la pseudoasociativa. Esto se debe a que la caché predictiva aumenta más la asociatividad de la caché y esto reduce en mayor medida los fallos por conflicto. Por otro lado, en el caso de la caché predictiva, el mayor número de comparaciones hace que el tiempo de acceso a la caché aumente más que para la caché pseudoasociativa.

Así que se obtendrá mejor rendimiento con una alternativa u otra dependiendo de cómo se compensen las variaciones de estos dos factores: tiempo de acceso y tasa de fallos.

#### 4.1.2.3. TÉCNICAS DE PREBÚSQUEDA

Esta técnica consiste en traer a la memoria caché, o a un buffer cercano, bloques de memoria que todavía no han sido referenciados por el procesador. Se trata de traer a la memoria caché, o a una estructura cercana, los bloques que se predice que el procesador va a necesitar antes de que provoquen un fallo.

Cuando la prebúsqueda es por hardware, suelen implementarse técnicas de prebúsqueda por fallo, de manera que cada vez que se produce un fallo, se busca el bloque que lo ha provocado y el siguiente (siguiendo el principio de localidad espacial, es probable que el procesador lo pida a continuación).

Pero en algunos casos, para evitar las modificaciones que deben hacerse en el hardware para implementar la prebúsqueda, ésta se hace por software, de manera que es el compilador o el propio desarrollador el encargado de predecir los bloques que van a referenciarse.

Sea implementada por hardware o por software, esta técnica se basa en la utilización del ancho de banda de memoria que de otra manera se desaprovecharía, pero si no se implementa correctamente puede interferir con la resolución de los fallos y reducir el rendimiento de la jerarquía de memoria, por lo tanto hay que ser extremadamente cuidadosos en su implementación.

En algunos casos se utilizan estructuras hardware denominadas buffers de prebúsqueda en las que se almacenan los bloques prebuscados para que no reemplacen en la memoria caché a bloques que el procesador sí ha solicitado realmente.

#### 4.1.3. Reducción del tiempo de acceso

Con lo estudiado hasta el momento acerca de la organización de una memoria caché básica, sabemos que las cachés pequeñas y sencillas (con poca asociatividad y por lo tanto, con pocas comparaciones de etiquetas en cada acceso) son las más rápidas.

Pero de nuevo existen técnicas específicas para mejorar este aspecto concreto del rendimiento de una memoria caché.

##### 4.1.3.1. EVITAR LA TRADUCCIÓN DE DIRECCIONES VIRTUALES A DIRECCIONES FÍSICAS

Cuando el procesador realiza un acceso a memoria, es necesario traducir la dirección virtual que genera a una dirección física con la que acceder a la memoria caché y a la memoria principal (en la figura 4.5 se muestra un ejemplo en el que la traducción se realiza con un TLB y la memoria virtual es paginada).

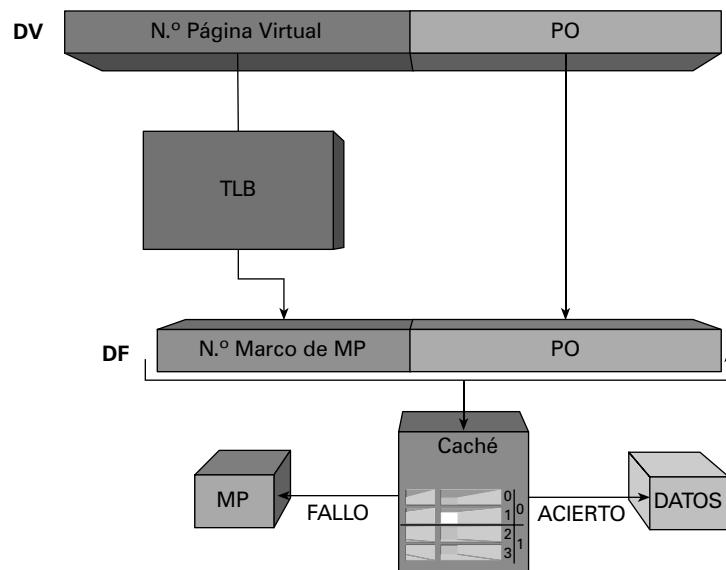


FIGURA 4.5

Memoria caché accedida con direcciones físicas.

Para ahorrarse esta traducción, cuyo tiempo se suma al tiempo de acceso a la memoria caché, pueden utilizarse memorias caché direccionadas con las direcciones virtuales, sin el paso previo de traducción a dirección física. La traducción sólo se hace cuando hay un fallo que hace necesario para su resolución un acceso a un nivel de la memoria que es accedido con direcciones físicas (figura 4.6).

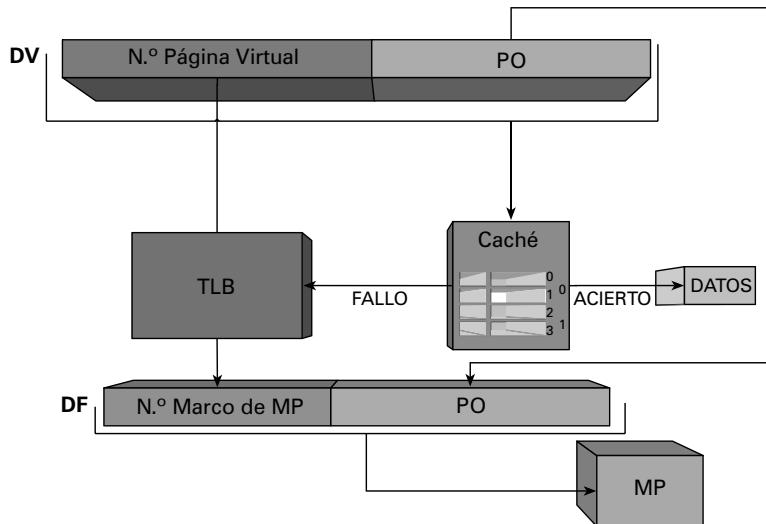


FIGURA 4.6

Memoria caché accedida con direcciones virtuales.

Esta técnica presenta dos inconvenientes fundamentales:

- Cuando se realiza un cambio de contexto (se pasa a ejecutar un nuevo proceso), las direcciones virtuales se refieren a otras direcciones físicas, por lo que hay que borrar el contenido de la caché y comenzar desde cero. Esto no permite aprovechar óptimamente el principio de localidad en algunos casos.
- Además, es posible que haya más de una dirección virtual que se refiera a la misma dirección física (aliasing o colisión). Esto puede llevar a tener dos copias de la misma información en la caché y a problemas de coherencia.

Estos problemas tienen diversas soluciones, tanto hardware como software. Desde el punto de vista que más nos interesa en este libro, el hardware, las alternativas más extendidas para reducir los tiempos que se emplean en la traducción de direcciones son soluciones intermedias entre el acceso con direcciones físicas y el acceso con direcciones virtuales, también llamadas cachés de dirección híbrida:

- **Memoria caché accedida con direcciones físicas pero realizando la traducción de dirección en paralelo.** Como se puede observar en la figura 4.7, ya que las operaciones más frecuentes son las lecturas de memoria, mientras se hace la traducción de dirección virtual a física, se puede extraer el índice directamente de la dirección virtual. Con este índice, puede accederse a memoria caché y leer todas las etiquetas y bloques de los marcos en los que podría ubicarse el bloque que se está buscando. Cuando ya se ha traducido la dirección virtual a física, ya se tiene la etiqueta del bloque buscado y se puede seleccionar ese bloque de entre los que se habían leído o detectar que se ha producido un fallo. Para que se pueda utilizar esta técnica (figura 4.8), es necesario que pueda extraerse el índice directamente de la dirección virtual (tiene que estar dentro del Page Offset (PO), que es la parte de la dirección virtual que no se modifica al hacer la traducción a dirección física) y que la operación en memoria sea una lectura (obviamente no se puede escribir en memoria y modificar un contenido sin saber antes si se trata del bloque adecuado).

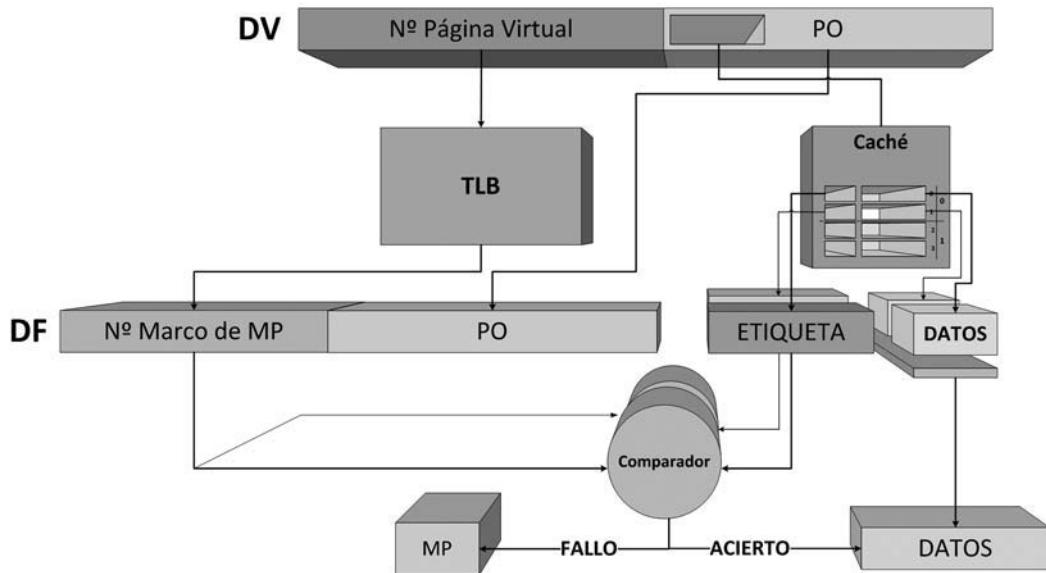


FIGURA 4.7

Memoria caché accedida con direcciones físicas pero realizando la traducción de dirección en paralelo.

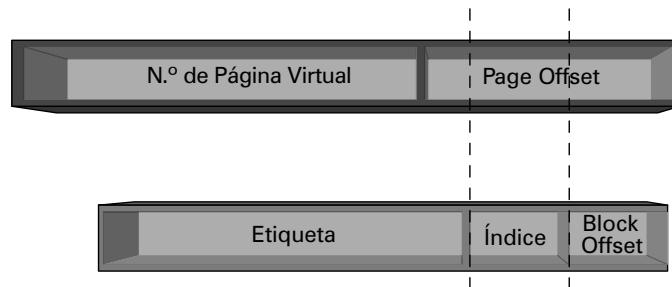


FIGURA 4.8

Correspondencia entre dirección virtual y física necesaria para poder aplicar la técnica de traducción en paralelo.

- **Memoria caché accedida con direcciones virtuales pero con etiquetas físicas (físicamente accedida, pero virtualmente indexada).** Si no cabe la posibilidad de utilizar la técnica anterior porque el índice no se puede extraer de la dirección virtual, se utiliza un índice que provenga de la dirección virtual. Sin embargo, para evitar los problemas asociados a las cachés accedidas con direcciones virtuales, para extraer la etiqueta se sigue utilizando la dirección física. Por lo tanto esta alternativa es mixta y emplea las dos direcciones para completar un acceso a memoria caché (figura 4.9).

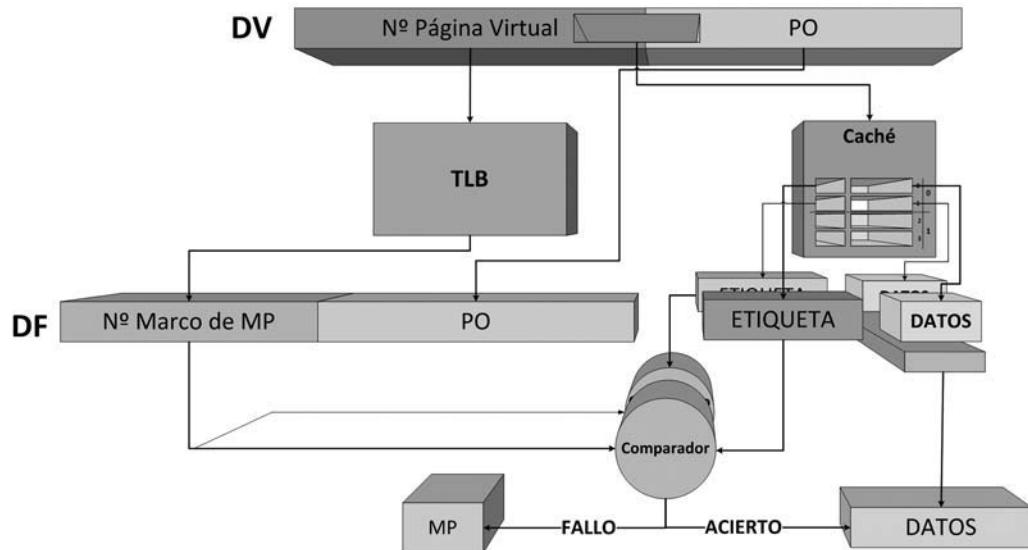


FIGURA 4.9

Memoria caché físicamente accedida pero virtualmente indexada.

**Ejemplo 4.5**

Ejemplo del efecto de un TLB no ideal en el tiempo medio de acceso a memoria con una caché accedida con direcciones físicas.

Hasta ahora no hemos tenido en cuenta en ningún ejemplo el tiempo que se invierte en traducir la dirección virtual generada por el procesador a la dirección física necesaria para acceder a la memoria caché. Es como si hubiéramos supuesto un TLB ideal que siempre es capaz de realizar esta traducción en un tiempo despreciable.

En el ejemplo de un nivel de memoria caché que hemos venido utilizando en el capítulo 2 y en éste es como si hubiéramos supuesto lo siguiente:

$$t_{MEM} = t_{DV \rightarrow DF} + t_{acuerdo} + TF \cdot pF$$

con  $t_{DV \rightarrow DF} = 0$ .

En este ejemplo vamos a ser algo más realistas y vamos a suponer que el TLB no es ideal, es decir, que en algunas ocasiones no puede realizar la traducción y por lo tanto es necesario recurrir a la memoria principal para obtenerla.

Supongamos que el TLB tiene una tasa de fallos del 4%, y que cuando falla, el 90% de las veces basta con un acceso a memoria principal mientras que el 10% restante son necesarios dos accesos para obtener la traducción. Y el acceso al TLB se realiza en 1 ns.

Teniendo en cuenta estos datos, el tiempo medio de acceso a memoria sería:

$$t_{MEM} = t_{DV \rightarrow DF} + t_{acuerdo} + TF \cdot pF$$

Con:

$$t_{DV \rightarrow DF} = t_{acceso\ TLB} + TF_{TLB} \cdot pF_{TLB} = 1 + 0.4 \cdot pF_{TLB} = 1 + 0.04 \cdot 93.5 = 4.74\text{ ns}$$

$$pF_{TLB} = 0.9 \cdot \text{latencia}_{MP} + 0.1 \cdot 2 \cdot \text{latencia}_{MP} = 0.9 \cdot 85 + 0.1 \cdot 2 \cdot 85 = 93.5\text{ ns}$$

Por lo que en realidad el tiempo medio de acceso a memoria en el ejemplo con palabra crítica primero, fallos prioritarios, caché de víctimas y caché pseudoasociativa sería:

$$t_{MEM} = t_{DV \rightarrow DF} + t_{acuerdo} + TF \cdot pF = 4.74 + (0.85 \cdot 4 + 0.15 \cdot 4.5) + 0.10 \cdot 75.10 = 16.32 \text{ ns}$$

Por eso es tan importante aplicar técnicas de optimización que permitan reducir el impacto de la traducción de dirección en el tiempo de acceso a memoria.

### Ejemplo 4.6

#### Ejemplo del efecto de un TLB no ideal en el tiempo medio de acceso a memoria con una caché accedida con direcciones virtuales.

Aunque la memoria caché utilice direcciones virtuales para su acceso, si se es realista con el rendimiento del TLB, éste sigue teniendo efecto en el tiempo medio de acceso a memoria.

Supongamos exactamente el mismo TLB que en el ejemplo anterior (tiempo de acceso de 1 ns, tasa de fallos del 4%, y en los fallos, el 90% de las veces basta con un acceso a memoria principal mientras que el 10% restante son necesarios dos accesos para obtener la traducción).

Si la caché es accedida con direcciones virtuales y acierta, no se llega a hacer la traducción de dirección virtual a física. Pero si la caché falla, hay que hacer la traducción a dirección física antes de salir a la memoria principal a resolver el fallo, por lo tanto el rendimiento del TLB afecta, en este caso, a la penalización por fallo.

El tiempo medio de acceso a memoria sería en el ejemplo con palabra crítica primero, fallos prioritarios, caché de víctimas y caché pseudoasociativa:

$$t_{MEM} = t_{acuerdo} + TF \cdot pF$$

Y hay que recalcular la penalización por fallo teniendo en cuenta el tiempo que se tarda en traducir de dirección virtual a física:

$$pF = t_{DV \rightarrow DF} + (1 - TF_{CV}) \cdot t_{CV} + TF_{CV} \cdot (latencia_{MP} + t_{bus}) + \%sucios \cdot ((1 - TF_{BE}) \cdot t_{volcado\ BE} + TF_{BE} \cdot 16 \cdot (latencia_{MP})) = 0.28 \cdot 19 + 0.72 \cdot (85 + 0.5) + 0.1 \cdot (0.95 \cdot 15 + 0.05 \cdot 16 \cdot 85) = 79.84 \text{ ns}$$

Con:

$$t_{DV \rightarrow DF} = t_{acceso\ TLB} + TF_{TLB} \cdot pF_{TLB} = 4.74 \text{ ns}$$

Por lo que,

$$t_{MEM} = t_{acuerdo} + TF \cdot pF = (0.85 \cdot 4 + 0.15 \cdot 4.5) + 0.10 \cdot 79.84 = 12.06 \text{ ns}$$

En este caso, tener en cuenta que el TLB no es ideal empeora ligeramente el tiempo de acceso a memoria, pero no tanto como con una caché accedida con direcciones físicas, ya que el tiempo de traducción de dirección sólo se sufre cuando hay un fallo. Sin embargo, ya se han comentado los principales inconvenientes de acceder con direcciones virtuales a la memoria caché.

#### 4.1.3.2. CACHÉS SEGMENTADAS

La técnica de segmentación no sólo se utiliza para mejorar el rendimiento de los procesadores, sino también de otros componentes de la computadora como puede ser la memoria caché.

Como ocurría con el procesador, en realidad esta técnica no reduce el tiempo de acceso de un único acceso a memoria, pero al permitir solapar la ejecución de diferentes accesos, reduce el tiempo medio observado por el procesador.

Cada acceso pasa a realizarse en más de un ciclo de reloj, pero estos ciclos pueden ser más cortos. Y en cada uno de ellos se ejecuta una de las etapas en las que puede dividirse el acceso a la memoria caché: traducción de dirección virtual a física, decodificación y acceso a fila, decodificación y acceso a columna, comparación de etiqueta, etc.

#### 4.1.3.3. CACHÉS DE TRAZAS

Está técnica de optimización es exclusiva de cachés de instrucciones, y no sólo reduce el tiempo de acceso sino que también reduce la tasa de fallos.

En este tipo de caché no se almacenan instrucciones individuales, sino que se almacenan trazas completas de instrucciones. Una traza no es más que una secuencia de instrucciones en el orden en el que se predice que van a ser buscadas (esto incluye los saltos, las llamadas a procedimientos ...).

A menudo se almacenan diferentes secuencias para un mismo fragmento de código, mientras se van resolviendo las predicciones de los saltos, por lo que en estos casos se desaprovecha espacio almacenando instrucciones replicadas.

Pero cada vez que se accede a la caché de trazas se recupera un conjunto de instrucciones en lugar de una sola (por eso se reduce el tiempo de acceso). Además, como las instrucciones recuperadas están en el orden en el que el procesador las va a solicitar, se reduce la tasa de fallos. Y por último, se optimiza la utilización del espacio de la caché ya que no se almacenan en ella instrucciones que se predice que no se van a ejecutar.

#### Ejemplo 4.7

Diseño de una jerarquía de memoria con dos niveles de caché (el primer nivel de escritura directa y el segundo nivel de post-escritura) completamente optimizada.

Supongamos ahora que tenemos el mismo ejemplo de jerarquía de memoria que en el capítulo 2, el ejemplo 2.7.

La caché de nivel 1 tiene un tamaño de bloque de 8 palabras, un tiempo de acceso de 1 ns y una tasa de fallos del 5%. La caché de nivel 2 tiene un tamaño de bloque de 16 palabras, un tiempo de acceso de 9 ns y una tasa de fallos del 9%.

Además la caché de nivel 1 utiliza escritura directa y la caché de nivel 2 utiliza post-escritura, con un 26% de bloques que son modificados mientras están ubicados en ella.

La transferencia de una palabra entre memoria principal y la caché de nivel 2 supone 0.5 ns y la transferencia de una palabra entre la memoria caché de nivel 2 y la de nivel 1 supone 0.1 ns. La latencia de acceso a memoria principal es de 85 ns.

Con todos estos datos sabíamos que:

$$t_{MEM} = \%lectura \cdot t_{MEM} (\text{lectura}) + \%escritura \cdot t_{MEM} (\text{escritura}) = 0.7 \cdot 66.69 + 0.3 \cdot 75.69 = 69.39 \text{ ns}$$

Y este tiempo es intolerable para un procesador actual. Pero ahora que sabemos cómo se aplican las diferentes técnicas de optimización de memoria caché, podemos ver hasta qué punto nos sería posible reducir este tiempo medio de acceso.

Vamos a incorporar las siguientes modificaciones y optimizaciones a esta jerarquía de memoria inicial:

- Añadimos un buffer de escritura para el nivel 1 de caché. Este buffer permite realizar las escrituras directas en un tiempo de 1.5 ns en un 90% de las ocasiones.
- Además el nivel 1 de caché se transforma en una memoria que funciona sin asignación en escritura.
- Utilizamos la técnica de palabra crítica primero en la caché de nivel 1.
- En el nivel 2 de caché, damos prioridad a los fallos con un buffer de escritura que permite volcar los bloques sucios el 92% de las veces en un tiempo de 18 ns.
- También se añade a este nivel una caché de víctimas que el 31% de las ocasiones permite resolver los fallos en 16 ns.
- Se reduce la tasa de fallos de la caché de nivel 1 del 5% al 4% convirtiéndola en una caché pseudoasociativa sin que esto tenga repercusiones significativas en su tiempo de acceso.
- Se reduce la tasa de fallos del nivel 2 del 9% al 7% utilizando para ello técnicas de prebúsqueda hardware.

Con todo esto, tenemos:

$$t_{MEM}(\text{lectura}) = t_{acuerdoL1} + TF_{L1} \cdot pF_{L1}$$

$$t_{MEM}(\text{escritura}) = (1 - TF_{L1}) (t_{acuerdoL1} + (1 - TF_{BE1}) \cdot t_{BE1} + TF_{BE1} \cdot t_{acuerdoL2}) + TF_{L1} \cdot t_{L2}$$

En ambos casos la penalización por fallo del nivel 1 es la misma, con la técnica de palabra crítica primero:

$$pF_{L1} = t_{L2} + t_{bus\ L1L2}$$

El tiempo medio de acceso a la caché de nivel 2 (ya que cuando la caché de nivel 1 falla y se busca el bloque en la caché de nivel 2, ésta puede acertar o fallar), se calcula de la misma manera:

$$t_{L2} = t_{acuerdoL2} + TF_{L2} \cdot pF_{L2}$$

Y como la caché de nivel 2 es de post-escritura:

$$pF_{L2} = (1 - TF_{CV}) \cdot t_{CV} + TF_{CV} \cdot 16 \cdot (\text{latencia}_{MP}) + 0.26 \cdot ((1 - TF_{BE2}) \cdot t_{volcado\ BE2} + TF_{BE2} \cdot 16 \cdot (\text{latencia}_{MP}))$$

Por lo tanto:

$$pF_{L2} = (1 - 0.69) \cdot 16 + 0.69 \cdot 16 \cdot 85 + 0.26 \cdot ((1 - 0.08) \cdot 18 + 0.08 \cdot 16 \cdot 85) = 975.95 \text{ ns}$$

$$t_{L2} = 9 + 0.07 \cdot 975.95 = 77.32 \text{ ns}$$

$$pF_{L1} = 77.32 + 0.1 = 77.42 \text{ ns}$$

Y los tiempos medios para lectura y escritura quedan:

$$t_{MEM}(\text{lectura}) = 1 + 0.04 \cdot 77.42 = 4.1 \text{ ns}$$

$$t_{MEM}(\text{escritura}) = (1 - 0.04) (1 + (1 - 0.1) \cdot 1.5 + 0.1 \cdot 9) + 0.04 \cdot 77.32 = 6.21 \text{ ns}$$

En media, si de nuevo tenemos que un 70% de los accesos son lecturas y un 30% son escrituras:

$$t_{MEM} = \% \text{lectura} \cdot t_{MEM}(\text{lectura}) + \% \text{escritura} \cdot t_{MEM}(\text{escritura}) = 0.7 \cdot 4.1 + 0.3 \cdot 6.21 = 4.73 \text{ ns}$$

Este tiempo medio de acceso a memoria sí que es razonable para una arquitectura actual pero es que con todas las técnicas de optimización aplicadas se ha conseguido un speedup de casi 15 respecto del diseño sin optimizar ( $69.39/4.73 = 14.67$ ).

Además hay que tener en cuenta que este tiempo medio se ha conseguido, de momento, optimizando sólo la memoria caché, pero que cualquier optimización que podamos realizar sobre la memoria principal (próxima sección) influirá directamente en la penalización por fallo de la caché de nivel 2 y permitirá reducir todavía más este tiempo medio de acceso.

## 4.2 Técnicas de optimización para la memoria principal

La memoria principal es el nivel de la jerarquía de memoria que se sitúa a continuación de la memoria caché. Por lo tanto, se accederá a esta memoria cuando ocurran fallos de caché.

Esto implica que la latencia de la memoria principal afecta a la penalización por fallo de la memoria caché. Sin embargo, la mayor parte de las técnicas de optimización de memoria principal van dirigidas a aumentar su ancho de banda (que también influye en la penalización por fallo) ya que resulta bastante más sencillo que reducir la latencia, que depende casi exclusivamente de la tecnología.

Con la generalización de las cachés multinivel y estas técnicas que aumentan el ancho de banda de memoria principal, se han comenzado a utilizar bloques de un tamaño bastante mayor en la caché de nivel 2, lo que ha reducido su tasa de fallos considerablemente.

En el capítulo 2 ya se estudiaron las mejoras que las diferentes tecnologías de memoria principal introdujeron con su aparición. Esta sección se centra exclusivamente en las mejoras que pueden conseguirse mediante el diseño, todas ellas relacionadas con el ancho de banda.

Hay que tener en cuenta que la mayor parte de estas técnicas no sólo afectan al diseño de la memoria principal sino también al diseño de su controlador, encargado del mapeo de la dirección física y de la planificación de los accesos.

### 4.2.1. Incremento del ancho de banda

#### 4.2.1.1. MEMORIA PRINCIPAL CON PALABRA ENSANCHADA

Tradicionalmente la jerarquía de memoria completa trabaja con un ancho de una palabra, ya que la mayor parte de los accesos a memoria del procesador son de esta longitud.

Pero aumentando el ancho de palabra de la memoria principal, por ejemplo a 2 o 4 palabras, se aumentará el ancho de banda ya que en cada acceso se podrá recuperar el doble o el cuádruple de información. En la mayor parte de los casos, la caché de nivel 1 permanece con un ancho de una palabra y es el resto de la jerarquía la que se ensancha, tanto las memorias como los buses que las interconectan (ejemplo con cuatro palabras de ancho en la figura 4.10). De esta manera basta con utilizar un multiplexor para seleccionar las palabras que pasan de la zona ensanchada de la jerarquía a la zona con ancho de una palabra.

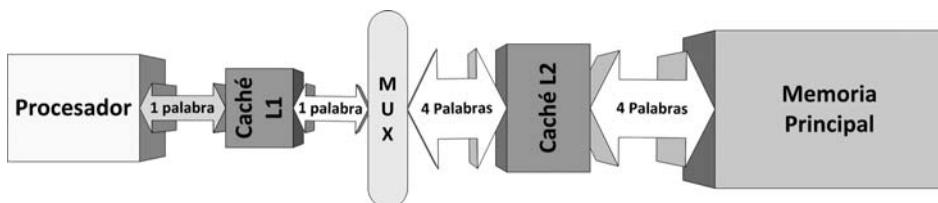


FIGURA 4.10

Memoria principal ensanchada a 4 palabras.

El ancho de banda que se consigue si se ensancha la jerarquía de memoria a N palabras es N veces el que se tiene cuando el ancho es de una palabra. El problema de esta técnica es que encarece enormemente los diseños debido a su gran consumo de área y de potencia. Además, al ensanchar las memorias, éstas suelen empeorar su latencia. Así que, aunque esta técnica se suele utilizar como referencia del rendimiento que sería deseable obtener, casi nunca se utiliza en la práctica.

#### 4.2.1.2. MEMORIA PRINCIPAL ENTRELAZADA

En este caso la memoria se organiza en bancos lógicos, normalmente de una palabra de ancho cada uno (ejemplo con cuatro bancos en la figura 4.11).

Si se direccionan varios bancos al mismo tiempo, con una única dirección, cada uno de ellos podrá recuperar una palabra (pero sólo se observa la latencia una vez porque trabajan en paralelo, el acceso se realiza simultáneamente en todos los bancos). A todos los efectos es como tener una memoria con un ancho mayor de una palabra, la principal diferencia es que la transferencia de las palabras que se recuperan en paralelo se multiplexa en el tiempo para compartir los recursos que siguen teniendo ancho de palabra uno entre todos los bancos. Por ejemplo, el bus de memoria sigue siendo de una palabra de ancho.

El entrelazado de la información, es decir, el reparto de los datos entre los diferentes bancos lógicos, puede realizarse a dos niveles:

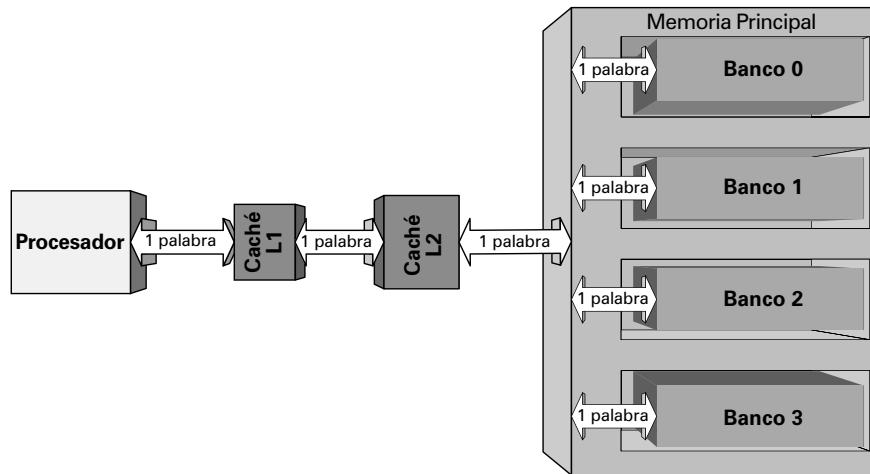


FIGURA 4.11

Memoria principal entrelazada con 4 bancos.

- **A bajo nivel (nivel de palabra).** Direcciones consecutivas de memoria se almacenan en bancos consecutivos.
- **A alto nivel.** Direcciones consecutivas de memoria se almacenan en el mismo banco de memoria.

Casi todas las memorias utilizan el entrelazado a nivel de palabra, porque consigue optimizar los accesos secuenciales a memoria, que son los más frecuentes. De esta manera, si  $N$  es el número de bancos que componen la memoria, con este tipo de entrelazado cada palabra se almacena en el banco dado por su dirección módulo  $N$  (figura 4.12).

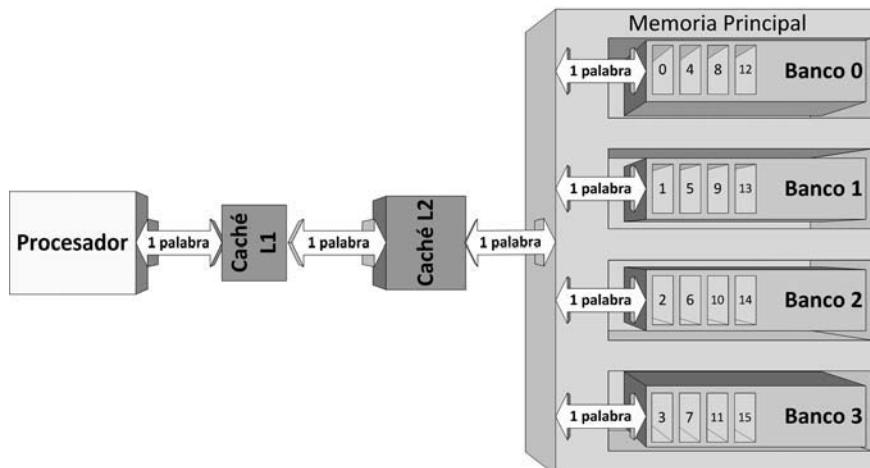


FIGURA 4.12

Ejemplo de entrelazado a bajo nivel en una memoria principal entrelazada con 4 bancos.

Uno de los principales inconvenientes de esta técnica es que las arquitecturas actuales utilizan muy pocos chips de memoria (que cada vez tienen una capacidad mayor). Esto supone un problema cuando

se desea implementar memoria entrelazada, ya que resulta mucho más caro crear los diferentes bancos de memoria.

Además, otro inconveniente de la memoria entrelazada es que cualquier ampliación de memoria resulta más costosa. Y se debe tener en cuenta que en el caso de accesos no secuenciales el paralelismo que puede extraerse en los accesos a memoria será mucho menor que el planificado inicialmente.

Por último cabe destacar que para obtener el rendimiento óptimo de una memoria entrelazada, el número de bancos,  $N$ , debe ser realmente elevado, ya que debe ser mayor que la latencia de cada uno de ellos en ciclos. De esta forma se pueden ir solapando los tiempos de dirección y transferencia de datos por el bus, con las latencias de los bancos de memoria (ejemplo 4.8).

Como conclusión, aunque con un entrelazado de  $N$  bancos, el ancho de banda de la memoria debería ser  $N$  veces el ancho de banda que en una memoria principal de ancho una palabra, en media la mejora suele ser  $\sqrt{N}$ .

#### Ejemplo 4.8

##### Ejemplo de memoria entrelazada con distintas configuraciones.

Supongamos que tenemos una memoria entrelazada con 8 bancos de memoria y que el tiempo de direccionamiento es 1 ciclo (tdir), la latencia de acceso es de 5 ciclos y el tiempo de bus es de 1 ciclo (tbus).

En este caso tenemos que  $N=8 > \text{latencia}=5$ , veamos cómo se comportaría la memoria:

|                                                            |                                                                                                                                                                                               |
|------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Ciclo 1 (tdir)                                             | Direccionamiento de la primera palabra que el procesador solicita.                                                                                                                            |
| Ciclos 2-6 (latencia)                                      | Latencia de acceso de los 8 bancos entrelazados en paralelo.                                                                                                                                  |
| Ciclos 7-14 (8 veces tbus y en paralelo, tdir y latencia)  | Transferencia de las 8 palabras recuperadas por el bus.<br>Direccionamiento de la segunda palabra que el procesador solicita.<br>Latencia de acceso de los 8 bancos entrelazados en paralelo. |
| Ciclos 15-22 (8 veces tbus y en paralelo, tdir y latencia) | Transferencia de las 8 palabras recuperadas por el bus.<br>Direccionamiento de la tercera palabra que el procesador solicita.<br>Latencia de acceso de los 8 bancos entrelazados en paralelo. |

Supongamos ahora que la latencia de acceso a memoria es de 10 ciclos en lugar de ser de 5 ciclos, es decir, supongamos que estamos en la situación contraria con  $N=8 < \text{latencia}=10$ :

|                                                               |                                                                                                                                                                                               |
|---------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Ciclo 1 (tdir)                                                | Direccionamiento de la primera palabra que el procesador solicita.                                                                                                                            |
| Ciclos 2-11 (latencia)                                        | Latencia de acceso de los 8 bancos entrelazados en paralelo.                                                                                                                                  |
| Ciclos 12-19 (tbus y en paralelo tdir y parte de la latencia) | Transferencia de las 8 palabras recuperadas por el bus.<br>Direccionamiento de la segunda palabra que el procesador solicita.<br>Latencia de acceso de los 8 bancos entrelazados en paralelo. |
| Ciclos hasta el 22 (parte de la latencia)                     | Latencia de acceso de los 8 bancos entrelazados en paralelo.                                                                                                                                  |
| Ciclos 23-30 (tbus y en paralelo tdir y parte de la latencia) | Transferencia de las 8 palabras recuperadas por el bus.<br>Direccionamiento de la tercera palabra que el procesador solicita.<br>Latencia de acceso de los 8 bancos entrelazados en paralelo. |
| Ciclos hasta el 33 (parte de la latencia)                     | Latencia de acceso de los 8 bancos entrelazados en paralelo.                                                                                                                                  |

Se observa que en este último caso se sufre mucho más la penalización debida a las latencias de acceso a memoria. De ahí que en las configuraciones con  $N < \text{latencia}$  no se pueda conseguir el ancho de banda óptimo deseado con este tipo de técnica de aumento de prestaciones.

#### 4.2.1.3. CANALES DE MEMORIA PRINCIPAL INDEPENDIENTES

El elevado coste que implica ensanchar la memoria principal y los inconvenientes que presenta la memoria entrelazada han obligado a proponer nuevas técnicas de optimización del ancho de banda de la memoria principal. La que más se ha extendido es la utilización de canales de memoria independientes. Esta técnica es una generalización de la memoria entrelazada. La única diferencia está en que cada banco de memoria es completamente independiente de los demás, es decir, cada uno tiene su propio bus de direcciones y de datos y su propio controlador.

Por ejemplo, las tecnologías Dual Channel o Three Channel que se encuentran actualmente en la mayor parte de computadores personales permiten tener dos o tres bancos de memoria completamente independientes respectivamente, aumentando así el ancho de banda en un factor 2 ó 3 respectivamente.

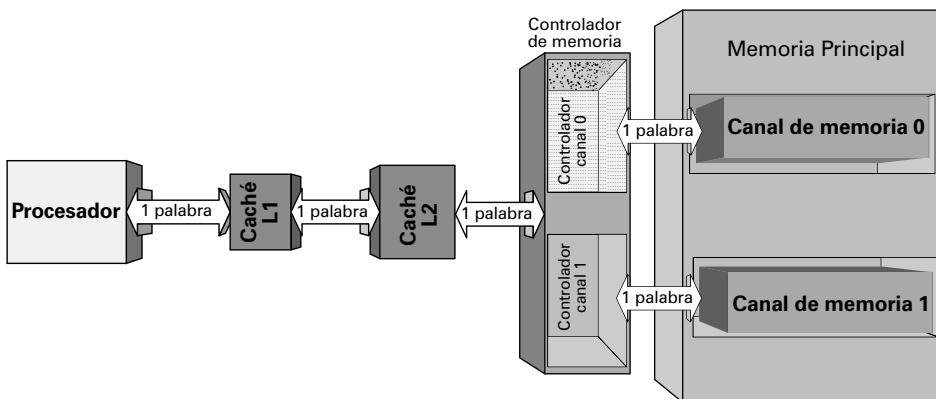


FIGURA 4.13

Memoria principal con dos canales de memoria independientes.

#### Ejemplo 4.9

Comparación de las diferentes técnicas de optimización del ancho de banda de memoria principal.

Supongamos una memoria principal sin optimizar en la que el direccionamiento de una palabra supone 1 ciclo de reloj, la latencia de acceso sean 100 ciclos y la transferencia de una palabra por el bus sea de nuevo de 1 ciclo. Para facilitar la comprensión del ejemplo, supongamos que en esta memoria no es posible solapar el direccionamiento, con la latencia de acceso o con las transferencias.

Si el tamaño de bloque de la caché de nivel 2 es de 64 palabras, recuperar este bloque de memoria principal supondría un tiempo:

$$t_{bloque} = 64 \cdot (1 + 100 + 1) = 6528 \text{ ciclos}$$

Si decidimos ensanchar la jerarquía de memoria, por ejemplo, a cuatro palabras, este tiempo sería:

$$t_{bloque} = \frac{64}{4} \cdot (1+100+1) = 1632 \text{ ciclos}$$

Este cálculo no es muy realista, ya que al ensanchar la jerarquía de memoria, tanto las tareas de gestión/mantenimiento como el propio hardware se complican, y esto suele ralentizar su funcionamiento. Pero a

efectos de estimación, supongamos que las consecuencias del ensanchamiento no son muy importantes. En realidad, ya sabemos que el principal problema de esta optimización es que suele ser impracticable debido a su elevado coste.

Si probamos a utilizar una memoria entrelazada con cuatro bancos, tenemos:

$$t_{bloque} = \frac{64}{4} \cdot (1+100) + 64 \cdot 1 = 1680 \text{ ciclos}$$

Esta solución, mucho más barata y asequible que el ensanchamiento de toda la jerarquía, consigue un rendimiento bastante similar con costes mucho menores. El problema está en que ya hemos visto que si el número de bancos es inferior al número de ciclos que se invierten en la latencia de acceso, no se puede aprovechar al máximo.

Por último, podemos optar por la solución de canales de memoria independientes. Hoy en día podemos encontrarnos opciones de Dual Channel o de Three Channel:

$$t_{bloque} = \frac{64}{2} \cdot (1+100+1) = 3264 \text{ ciclos}$$

$$t_{bloque} = \left\lceil \frac{64}{3} \right\rceil \cdot (1+100+1) = 2244 \text{ ciclos}$$

Y es esta última alternativa la más utilizada debido al compromiso que obtiene entre coste, complejidad y prestaciones.

#### 4.2.1.4. MEMORIA PRINCIPAL SEGMENTADA

Esta técnica se aplica de manera muy similar que en las memorias caché, pero en este caso se utilizan todavía más etapas de segmentación, ya que se trata de un acceso bastante más complejo (mapeo de dirección física a ubicación en memoria principal, activación de señales RAS y CAS, decodificación de direcciones de fila y columna, etc).

Depende casi exclusivamente de que el controlador de memoria soporte esta técnica, ya que al ser el encargado de planificar los accesos a la memoria principal, será también el principal encargado de gestionar la temporización de las etapas, de evitar posibles conflictos por recursos, etc.

### 4.3 Técnicas de optimización conjunta para todos los niveles de la jerarquía: visión global de la jerarquía de memoria

#### 4.3.1. Soporte a las técnicas de optimización del procesador

Una vez comprendidas las técnicas de optimización de la jerarquía de memoria conviene pararse a pensar si son adecuadas para alimentar de instrucciones y datos a un procesador que incorpore las técnicas de optimización estudiadas en el capítulo 3 a un ritmo suficiente.

En primer lugar, para obtener el máximo rendimiento de un procesador segmentado, es necesario que la memoria caché sea no bloqueante.

Además, para realizar emisión múltiple de instrucciones, será necesario que, como mínimo, la memoria caché de instrucciones sea segmentada y/o multipuerto. También podría utilizarse una caché de trazas.

Y además, en el caso de la ejecución especulativa, se deberán ignorar las excepciones que causen las referencias a memoria con direcciones incorrectas, habituales cuando se utilizan esquemas de especulación mínimamente sofisticados.

Teniendo en cuenta estas consideraciones, y una vez vista la mejora que se consigue en la jerarquía de memoria básica introducida en el capítulo 2 gracias a todas las técnicas de optimización estudiadas en este capítulo, se puede concluir que es posible para la jerarquía de memoria servir de instrucciones y datos a un procesador optimizado con las prestaciones requeridas.

### 4.3.2. Almacenamiento local vs memoria caché

En algunas arquitecturas actuales se ha decidido dar un paso atrás y no incluir memoria caché dentro de la jerarquía de memoria sino un nivel denominado normalmente de almacenamiento local, que no es más que un segundo nivel de banco de registros.

Las arquitecturas que se deciden por esta opción son mucho más complicadas de programar, ya que exigen que desde las aplicaciones se ordene cargar a este almacenamiento local, casi siempre mediante DMA de bloques de información (en el orden de magnitud de KB), la información que el procesador necesita para trabajar en cada momento.

El almacenamiento local se está utilizando para reducir la complejidad del hardware en sistemas que incluyen varios procesadores en un único chip. De esta manera el diseño es más sencillo (se evitan todas las tareas de gestión de aciertos y fallos de la memoria caché, las comparaciones de etiquetas, etc) y se evita implementar protocolos de coherencia de caché. Además, los tiempos de acceso a un almacenamiento local son más rápidos que a una memoria caché, aunque obviamente, se traspasa a los desarrolladores la responsabilidad de que los datos que necesita el procesador estén en el almacenamiento local. Por eso esta técnica se utiliza normalmente en procesadores de propósito específico, en los que las aplicaciones son siempre del mismo tipo y es relativamente sencillo predecir los datos que tienen que llevarse al almacenamiento local.

En un procesador de propósito general, prescindir de los niveles de memoria caché de la jerarquía y pretender que todas las aplicaciones, del tipo que sean, predigan los datos que van a necesitar en este almacenamiento local, llevaría a una degradación importante de este rendimiento.

En la misma dirección que los almacenamientos locales, se pueden encontrar arquitecturas de propósito específico que sí implementan memoria caché pero que permiten bloquear algunas zonas de este nivel de la jerarquía (locking) para que funcionen como un almacenamiento local. Normalmente este bloqueo se realiza cuando se trabaja intensivamente con un conjunto de bloques de la caché, ya que bloqueando esta zona de la memoria, se garantiza un tiempo de acceso más rápido y que ninguno de estos bloques se va a reemplazar mientras el procesador los necesita.

## 4.4 Técnicas de optimización para los buses de E/S

La expresión del ancho de banda de un bus que se estudió en el capítulo 2 es:

$$BW = \text{ancho de datos} \cdot f \cdot n.^{\circ} \text{ de transferencias por ciclo}$$

A partir de esta expresión se intuye que cualquier optimización que pretenda mejorar el ancho de banda de un bus debe ir orientada a incrementar alguno de estos tres aspectos: el ancho de datos, la frecuencia de operación del bus o el número de transferencias por ciclo.

Se estudiará a continuación que además existen técnicas que permiten mejorar los tres aspectos simultáneamente.

### 4.4.1. Ancho de datos y frecuencia de operación

Tradicionalmente, el ancho de datos de un bus viene determinado por el número de líneas de datos que incorpora (siempre hay señales de control, de reloj, etc, que no se utilizan para la transferencia de información).

En el capítulo 2 (sección 2.7) se señaló que este ancho está limitado por las interferencias que se producen entre unas líneas y otras, especialmente a frecuencias de operación altas.

Por este motivo es difícil encontrar buses con un ancho de datos por encima de 64 o 128 bits. De hecho, la tendencia de optimización actual es la utilización de buses con un ancho cada vez menor, casi serie en lugar de paralelo, y que utilicen señalización diferencial para soportar frecuencias de funcionamiento mayores.

¿En qué consiste esta técnica de optimización? Los buses tradicionales dedican un conductor para cada bit de información que se desea transmitir por el bus. Para saber el valor lógico de cada uno de los bits de información, se compara el valor de la tensión que lleva cada uno de los conductores con una tierra global.

Sin embargo, al trabajar a frecuencias de bus elevadas, la tensión de cada uno de los conductores puede sufrir fluctuaciones que hagan que valores de 1 lógico se acabén interpretando por error como un 0 lógico, y viceversa.

La señalización diferencial utiliza dos conductores para cada bit de información que se desea transmitir por el bus. Para saber el valor lógico de cada uno de los bits de información, se comparan las tensiones de los dos conductores. De esta manera, si existen interferencias o ruidos que originen fluctuaciones en las tensiones de los conductores, serán similares en los dos que se dedican a cada bit (siempre ubicados próximos físicamente), por lo que el bus será mucho más robusto ante el ruido y se podrá trabajar a frecuencias mucho mayores sin problemas.

### Ejemplo 4.10

#### Ejemplo de optimización de un bus mediante la utilización de señalización diferencial.

Supongamos que se desea optimizar un bus con un ancho de datos de 32 bits, que realiza una transferencia por ciclo y cuya frecuencia de funcionamiento es de 333 MHz.

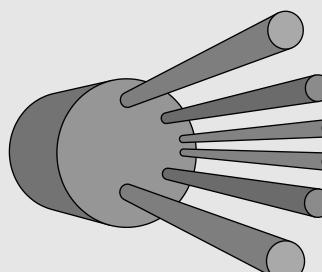
El ancho de banda original de este bus es de:

$$BW = \text{ancho de datos} \cdot f \cdot n.^o \text{ de transferencias por ciclo} = 4B \cdot 333 \text{ MHz} \cdot 1 = 1.3 \text{ GB/s}$$

El problema que presenta este bus es que se observa que tanto si se aumenta el ancho de datos como si se aumenta la frecuencia de funcionamiento, comienzan a aparecer interferencias entre las líneas del bus.

Para poder evitar estas interferencias se decide utilizar señalización diferencial. En el bus original, cada una de las líneas lleva una tensión de entre 0 y 5 V. Los valores entre 0 y 1.5 V se interpretan como 0 lógico, y los valores entre 3.5 y 5 V se interpretan como 1 lógico (el resto de valores son indeterminados).

Supongamos que en el bus original se produce una interferencia que resta 0.5 V a todas las líneas del bus, entonces:



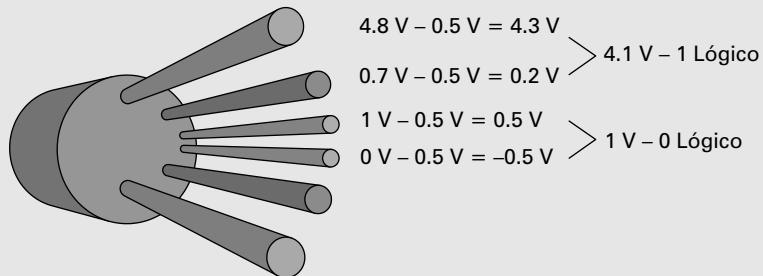
$$3.8 \text{ V} - 0.5 \text{ V} = 3.3 \text{ V} - \text{Indefinido}$$

$$4.0 \text{ V} - 0.5 \text{ V} = 3.5 \text{ V} - 1 \text{ Lógico}$$

$$0 \text{ V} - 0.5 \text{ V} = -0.5 \text{ V} - \text{Indefinido}$$

$$3.5 \text{ V} - 0.5 \text{ V} = 3.0 \text{ V} - \text{Indefinido}$$

En algunas líneas comenzarían a aparecer indeterminaciones. Pero si convertimos el bus a señalización diferencial, aunque exista una interferencia que reste 0.5 V a todas las líneas, la información que se transmite por el bus no se corrompe:



En este caso, las 32 líneas que se empleaban en el bus original para transmitir 32 bits, ahora se emplean para transmitir 16 bits. Pero gracias a la mayor robustez frente al ruido, la frecuencia de funcionamiento del bus puede aumentarse hasta 1.5 GHz. A una frecuencia mayor no daría tiempo a completar una transferencia completa por ciclo de reloj, por eso se decide dejar este valor para la frecuencia.

El ancho de banda del bus optimizado es:

$$BW = \text{ancho de datos} \cdot f \cdot n.^o \text{ de transferencias por ciclo} = 2B \cdot 1.5 \text{ GHz} \cdot 1 = 3 \text{ GB/s}$$

#### 4.4.2. Número de transferencias por ciclo

Durante muchos años, ya que no se podía aumentar el ancho de banda de los buses incrementando el ancho de datos o la frecuencia debido a los problemas que generaban las interferencias y la disipación de calor, la única forma de optimizar el ancho de banda era mejorar en todo lo posible el modo de operación del bus y sus protocolos de transferencia, sincronización y arbitraje para que le diera tiempo a realizar más de una transferencia de información por ciclo.

Para ello se aprovechan los flancos de subida y de bajada de las señales de reloj, o se combinan varias señales de reloj desfasadas. Casi siempre se realiza un número de transferencias por ciclo que sea potencia de dos.

#### Ejemplo 4.11

##### Utilización de varias transferencias por ciclo para optimizar el ancho de banda de un bus.

Hace ya unos años, el bus estándar para la conexión de tarjetas gráficas en placas de PC se denominaba AGP. Se trataba de un bus bastante tradicional, con una frecuencia de funcionamiento de 66 MHz y un ancho de datos de 32 bits.

Las diferentes versiones de este bus fueron mejorando sus prestaciones incrementando simplemente el número de transferencias de información por ciclo, de manera que se pasó del AGP inicial al AGP 2X (dos transferencias por ciclo), al AGP 4X (cuatro transferencias por ciclo) y por último, al AGP 8X (ocho transferencias por ciclo):

$$BW(AGP) = \text{ancho de datos} \cdot f \cdot n.^o \text{ de transferencias por ciclo} = 4B \cdot 66 \text{ MHz} \cdot 1 = 264 \text{ MB/s}$$

$$BW(AGP2X) = \text{ancho de datos} \cdot f \cdot n.^o \text{ de transferencias por ciclo} = 4B \cdot 66 \text{ MHz} \cdot 2 = 528 \text{ MB/s}$$

$$BW(AGP4X) = \text{ancho de datos} \cdot f \cdot n.\text{o de transferencias por ciclo} = 4B \cdot 66 \text{ MHz} \cdot 4 = 1.05 \text{ GB/s}$$

$$BW(AGP8X) = \text{ancho de datos} \cdot f \cdot n.\text{o de transferencias por ciclo} = 4B \cdot 66 \text{ MHz} \cdot 8 = 2.1 \text{ GB/s}$$

Después de esta última versión, la técnica de aumentar el número de transferencias por ciclo no se pudo utilizar más y se cambió el estándar de conexión de tarjetas gráficas a PCI Express x16, como veremos más adelante, un bus que incorpora técnicas de optimización más sofisticadas.

#### 4.4.3. Utilización de protocolos de comunicaciones de alto rendimiento

Las técnicas de optimización de buses que se han impuesto en la actualidad se basan en la utilización de protocolos de comunicación de alto rendimiento.

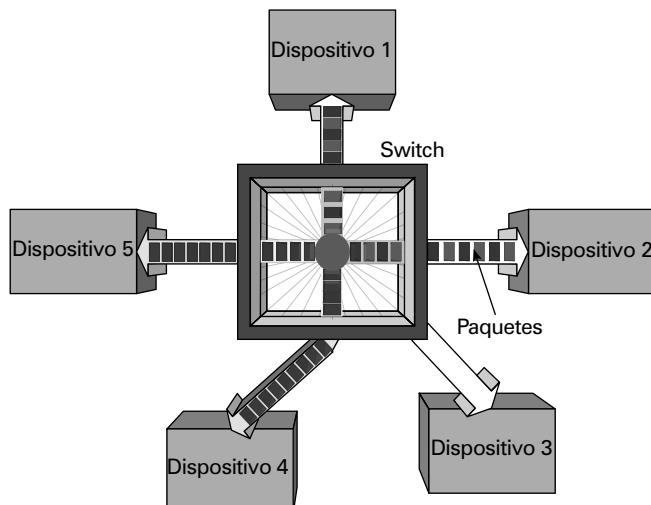
Los buses de E/S dentro de una computadora cada vez se parecen más a las redes de comunicaciones y se maneja la misma terminología: conexión punto a punto, paquete de datos, switch.

La idea principal es evitar la señalización de control, y utilizar protocolos de red dentro de las arquitecturas, codificando la información de control junto con los datos, enruteando los paquetes resultantes mediante switches desde el origen hasta el destino.

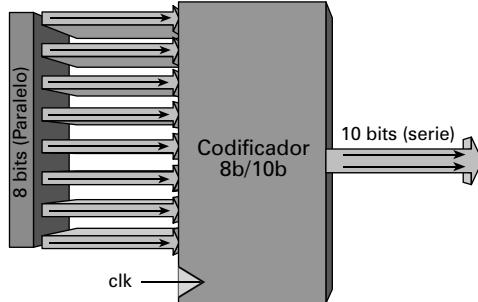
Claros ejemplos de este tipo de optimización son buses como PCI Express o Hipertransporte.

#### CASO PRÁCTICO 4.1. PCI Express.

Se trata de una tecnología de conexión punto a punto basada en el antiguo bus PCI que permite la mínima latencia de comunicaciones en enlaces entre chips. Está basada en uno o varios switches dentro del sistema que actúan como controladores de todos los dispositivos que utilizan PCI-Express y que se encargan de la gestión de los paquetes que se utilizan para transferir información por el bus.

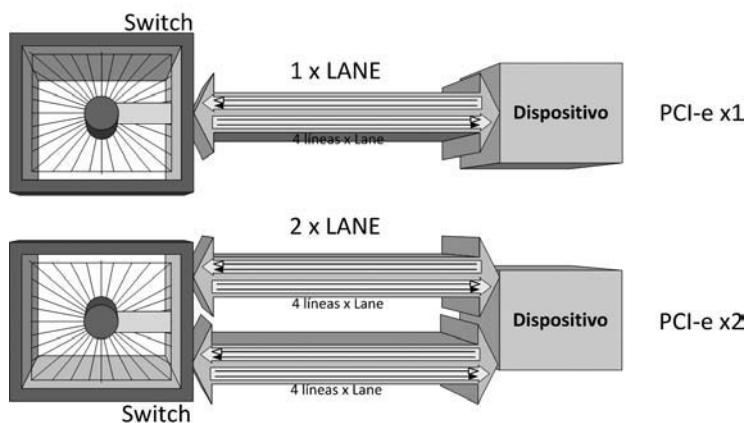


El protocolo de codificación de paquetes que utiliza PCI-Express es síncrono, codificando la señal de reloj junto con los datos de manera que se transmiten 10 bits de información por cada 8 de datos. Existe también un modo isócrono definido en el estándar para que se pueda funcionar a tiempo real, reservando un ancho de banda determinado para el dispositivo que lo necesita.



El estándar soporta cuatro tipos básicos de paquetes, los de transacción con memoria, transacción de E/S, configuración y mensajes (gestión avanzada de interrupciones).

El ancho de datos de este bus puede ser de 1, 2, 4, 8, 12 o 16 bits, con un canal para cada dirección y señalización diferencial. De ahí las versiones PCI Express x1, x2, x4, x8, x12 o x16, esta última se ha convertido en el bus estándar para conexión de tarjetas gráficas en los PCs. Por lo tanto un enlace PCI Express puede ser x1, x2, x4, x8, x12 o x16 según el número de lanes que lo compongan. Un lane no es más que el conjunto de 4 líneas, dos en un sentido y dos en el contrario (porque se utiliza señalización diferencial y es un bus full-duplex), que permite la transmisión de 1 bit en los dos sentidos al mismo tiempo. Los dispositivos PCI-Express negocian con el switch cuántos canales o lanes podrán utilizar para sus comunicaciones.



La frecuencia de funcionamiento de este bus es siempre de 2.5 GHz, pero la versión 1.0 hace 0.8 transferencias por ciclo, y la 2.0, justo el doble, 1.6.

$$\begin{aligned}BW(\text{PCI Express x16 1.0}) &= \text{ancho de datos} \cdot f \cdot n.\text{o de transferencias por ciclo} = 2B \cdot 2.5 \text{ GHz} \cdot 0.8 = \\&= 4 \text{ GB/s en cada sentido (porque es full-duplex)}\end{aligned}$$

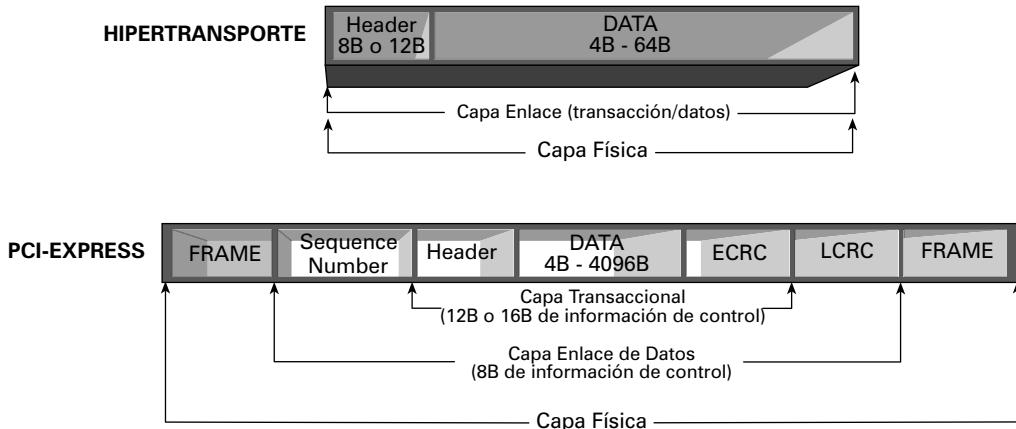
$$\begin{aligned}BW(\text{PCI Express x16 2.0}) &= \text{ancho de datos} \cdot f \cdot n.\text{o de transferencias por ciclo} = 2B \cdot 2.5 \text{ GHz} \cdot 1.6 = \\&= 8 \text{ GB/s en cada sentido (porque es full-duplex)}\end{aligned}$$

## CASO PRÁCTICO 4.2. Hipertransporte.

También se trata de una tecnología de conexión punto a punto que permite la mínima latencia de comunicaciones en enlaces entre chips, pero en este caso, el estándar está diseñado como un protocolo de conexión universal, que no se limita a los buses dentro de una computadora.

También está basado en uno o varios switches dentro del sistema que actúan como controladores de todos los dispositivos que utilizan Hipertransporte y que se encargan de la gestión de los paquetes que se utilizan para transferir información por el bus.

De nuevo el protocolo de codificación de paquetes es síncrono, aunque en este caso es mucho más eficiente que el que utiliza PCI Express (se reduce mucho la información de control, se utilizan menos cabeceras, etc). El tamaño de los paquetes es siempre un múltiplo de 4 bytes.



El ancho de datos de este bus puede ser de 2, 4, 8, 16 o 32 bits, con un canal para cada dirección y señalización diferencial. La frecuencia de funcionamiento varía entre 800 MHz y 3.2 GHz dependiendo de la versión, y siempre se realizan dos transferencias de información por ciclo.

Este bus se está utilizando como bus del sistema y como bus de memoria, no sólo para la conexión de dispositivos periféricos como PCI Express hasta el momento. Un ejemplo típico de ancho de banda para bus del sistema sería:

$$BW(HT \text{ versión 3.0}) = \text{ancho de datos} \cdot f \cdot n.\text{o de transferencias por ciclo} = 4B \cdot 2.6 \text{ GHz} \cdot 2 = 20.8 \text{ GB/s en cada sentido (porque es full-duplex)}$$

## 4.5. Técnicas de optimización para los dispositivos de E/S

Esta sección se centra en la optimización del disco duro ya que es el dispositivo en el que reside la memoria virtual, y por lo tanto, sus optimizaciones afectarán al rendimiento de la jerarquía de memoria.

En cuanto a las optimizaciones del resto de dispositivos, al igual que en el capítulo 2, el funcionamiento de los periféricos concretos escapaba de los objetivos de este libro, el estudio de las optimizaciones específicas para un monitor, una impresora o un cámara de vídeo, por poner algunos ejemplos, escapan de nuevo a las intenciones de este libro.

### 4.5.1. Optimización para los dispositivos de almacenamiento: RAID

Un RAID es un conjunto de discos duros trabajando en paralelo (Redundant Array of Inexpensive Disks). Se trata de una solución para incrementar el ancho de banda del sistema de almacenamiento, ya que éste presenta el mismo problema que la memoria principal: la latencia del sistema depende fundamentalmente de la tecnología con la que está fabricado, por eso los esfuerzos de diseño deben ir más orientados a mejorar el ancho de banda.

Además esta solución permite mejorar el rendimiento del sistema de almacenamiento desde otros dos puntos de vista: la disponibilidad y la tolerancia a fallos, tremadamente importantes en multitud de arquitecturas.

¿En qué consiste un RAID? La idea principal es repartir los datos entre diferentes discos duros para que puedan ser accedidos en paralelo. Esta técnica se denomina stripping.

Pero así se disminuye la fiabilidad del sistema de almacenamiento, si se utilizan varios discos duros, aumenta la probabilidad de que se produzca un fallo en uno de ellos. Por eso la técnica de stripping suele combinarse con redundancia (completa o mediante códigos de paridad). De esta manera se tiene tolerancia a fallos, porque si hay un error con un dato en un determinado disco, ese mismo dato estará almacenado en otro/s disco/s o información que permita recuperarlo. El problema es que la introducción de la información de redundancia complica la gestión del RAID y ocupa espacio.

Existen diferentes niveles de RAID dependiendo de cómo se combinen y se utilicen estas dos técnicas, el stripping y la redundancia, ya que para diseñar un RAID hay que tomar dos decisiones fundamentales:

- La granularidad del stripping. Si es muy fina, un acceso de E/S siempre involucrará a todos los discos del RAID. De esta manera se aumenta el ancho de banda pero no es posible hacer varias operaciones de E/S en paralelo. Si es más gruesa, un acceso puede ser servido por un único disco y por lo tanto, si el controlador del RAID lo permite, se podrán hacer varios accesos a disco en paralelo.
- Tipo de redundancia. Se puede escoger la opción del mirroring o redundancia completa, es decir, tener una copia completa de toda la información, o como esto es demasiado costoso, utilizar códigos de paridad. Y además la información de redundancia puede estar localizada en discos concretos o distribuida entre todos los discos.

Teniendo en cuenta estas decisiones, se pueden encontrar los siguientes niveles de RAID:

- **RAID 0.** Se utiliza sólo stripping (figura 4.14), es decir, se reparte la información entre los discos duros que componen el RAID, y la granularidad no está definida (el diseñador puede decidir si es fina o gruesa). Una ventaja de este nivel es que para el sistema operativo todos los discos duros que componen el RAID aparecen como un único disco duro de capacidad la suma de todos ellos, lo que facilita enormemente su gestión. Obviamente la desventaja es que no hay tolerancia a fallos y si un disco falla, se pierden los datos que almacenaba.

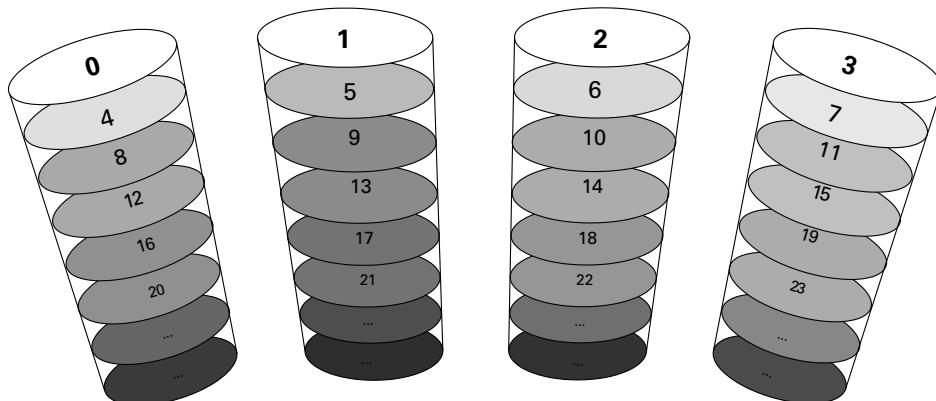


FIGURA 4.14

RAID 0.

- **RAID 1.** Es la solución que proporciona tolerancia a fallos al RAID 0 (figura 4.15), ya que la mitad de los discos se utilizan como discos espejo de la otra mitad (redundancia con mirroring). La desventaja es que resulta una solución muy cara debido a la gran cantidad de espacio que se emplea en almacenar la información de redundancia.

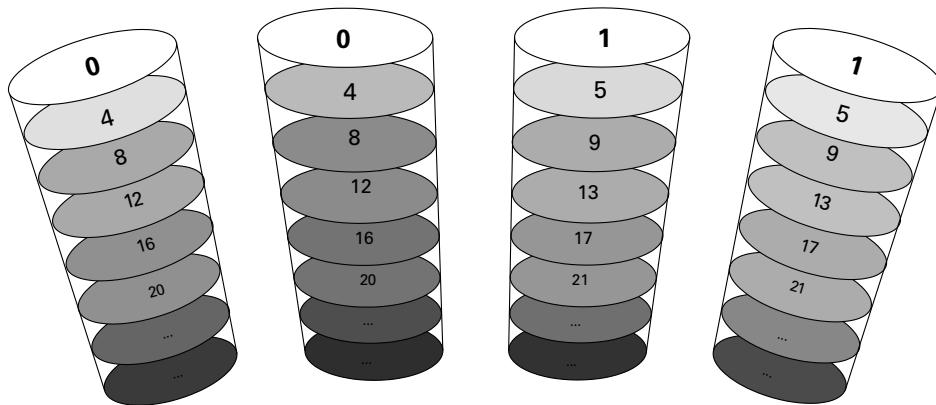


FIGURA 4.15

RAID 1.

- **RAID 3.** Para reducir el coste del RAID 1, se utiliza otra solución que proporcione tolerancia a fallos al RAID 0 sin necesidad de duplicar el número de discos (figuras 4.16 y 4.17). Simplemente se utiliza un disco duro extra para almacenar información de paridad calculada mediante el código de Hamming. En este nivel de RAID la granularidad del stripping es fina, es decir, se reparte la información entre los discos en tamaños de bit o byte. La desventaja es que cada vez que se hace una escritura en un disco, hay que leer todos los demás para calcular la paridad y escribirla en el disco de paridad.
- **RAID 4.** Para superar este inconveniente se utiliza granularidad de stripping gruesa (bloques de información de tamaño mayor, tipo sectores) y se aprovecha que no es necesario leer los discos que no se han modificado para calcular la nueva paridad, basta con su antiguo valor y con los valores que se han modificado (figuras 4.16 y 4.18).
- **RAID 5.** Es como el RAID 4, pero la información de paridad ya no se almacena en un disco dedicado a esta función, sino que se reparte entre todos los discos que componen el RAID (figura 4.18).

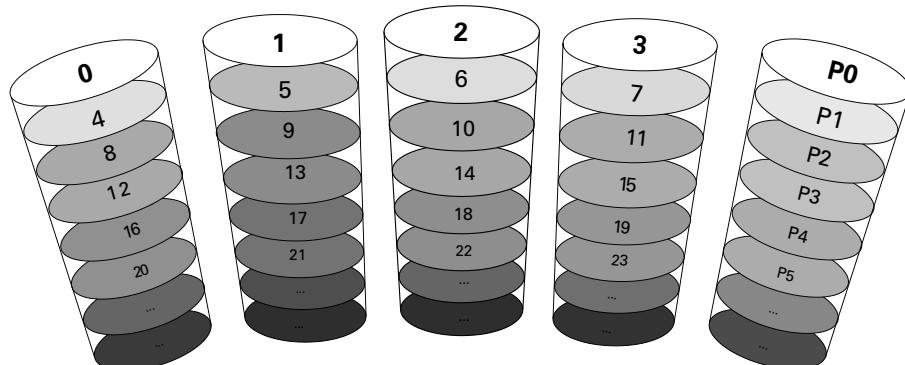


FIGURA 4.16

RAID 3 y RAID 4.

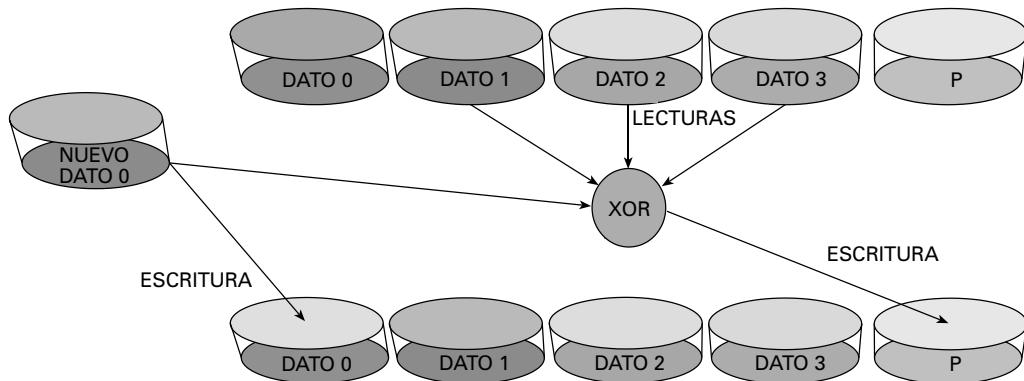


FIGURA 4.17

Actualización de la paridad en RAID 3.

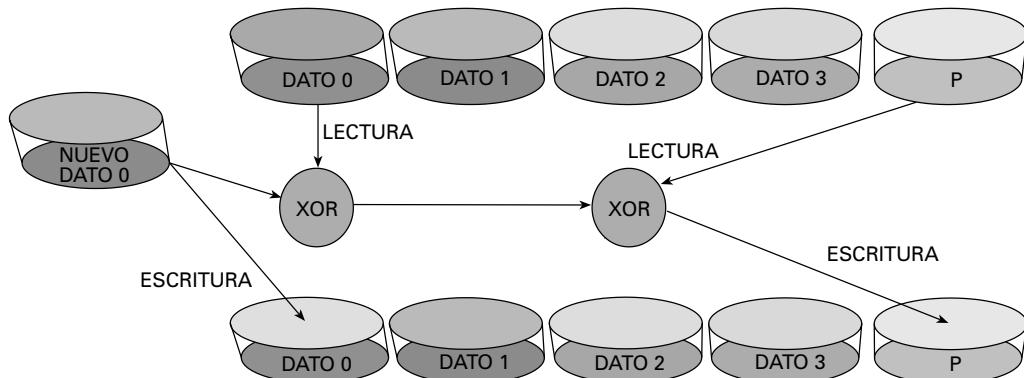


FIGURA 4.18

Actualización de la paridad en RAID 4.

ra 4.19). De esta manera se pueden realizar escrituras en paralelo porque en muchos casos ya no habrá riesgo estructural al actualizar la paridad (siempre y cuando el controlador del RAID lo permita).

- **RAID 6.** Este nivel añade al anterior técnicas de detección y corrección de errores P+Q, más complejas que la paridad simple calculada con Hamming de manera que se pueden corregir fallos de dos discos en lugar de uno solo. A cambio, la información de paridad ocupa más espacio.

Una última decisión que debe tomarse en cuanto al diseño de un RAID está relacionada con la recuperación de información cuando falla un disco (o dos en el caso del RAID 6). Existen dos tipos de recuperación:

- **Off line.** El RAID deja de dar servicio a las peticiones de lectura y escritura mientras se recuperan los datos que se han perdido. Por lo tanto, todo el ancho de banda está completamente disponible para realizar esta recuperación.
- **On line.** El RAID reserva una parte del ancho de banda para realizar la recuperación y el resto se utiliza para seguir atendiendo a las peticiones de lectura y escritura. Por lo tanto, la fiabilidad del sistema de almacenamiento es mayor, pero también el tiempo de recuperación es mayor.

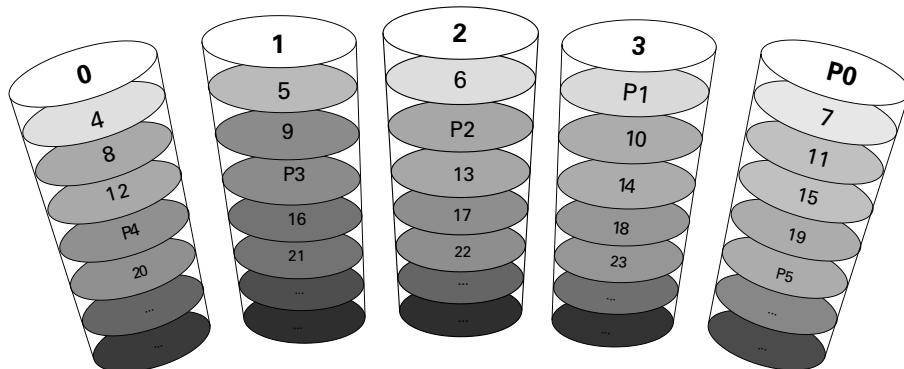


FIGURA 4.19

RAID 5.

Esta decisión puede llegar a ser importante para el rendimiento del RAID, ya que además de medir las prestaciones obtenidas con las métricas ya mencionadas, ancho de banda y latencia, en el caso de un RAID son importantes el tiempo medio hasta fallo de un disco (Mean Time To Failure o MTTF) y el tiempo medio hasta pérdida de datos (Mean Time To Data Loss o MTTDL).

#### Ejemplo 4.12

Cálculo del MTTF y del MTTDL en un RAID 0.

Se conecta a un sistema un RAID 0 de 6 discos SCSI de 40 GB cada uno. El tiempo medio hasta el fallo (Mean Time To Failure, MTTF) de estos discos es de 1.5 millones de horas.

Además, si un disco falla, la recuperación se puede hacer offline, es decir, el RAID deja de funcionar hasta que el fallo se corrija, o se puede hacer online, es decir, mientras el sistema sigue atendiendo peticiones. Para hacer estimaciones respecto a los tiempos de recuperación, se sabe que el bus es un UltraSCSI 320 con un ancho de banda total de 320 MB/s y que para la recuperación online sólo se deja un 10% del ancho de banda total del bus.

Con toda esta información podemos calcular que el tiempo medio hasta que se produzca un fallo en el RAID es de:

$$MTTF = \frac{MTTF \text{ (1 disco)}}{\text{Número de discos}} = \frac{1.5 \text{ millones de horas}}{6} = 250.000 \text{ horas}$$

Ya que la probabilidad de que fallen 6 discos es 6 veces superior a la probabilidad de que falle un único disco. Por eso el tiempo medio hasta fallo del RAID se obtiene dividiendo el tiempo medio de un único disco entre 6.

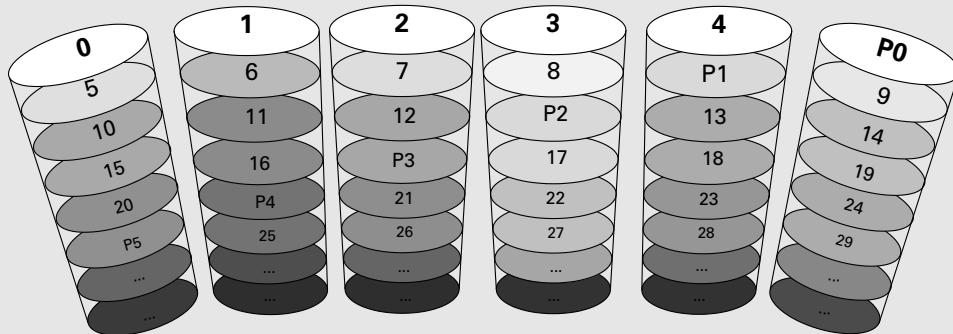
Como se trata de un RAID0 en el que no existe ningún tipo de información redundante, en cuanto falle un disco se pierden datos, por lo que el tiempo medio hasta pérdida de datos coincide con el tiempo medio hasta fallo de disco:

$$MTTDL = MTTF = 250.000 \text{ horas}$$

Y en este caso no existe recuperación posible. Si falla un disco, la información que almacenaba se pierde y es imposible realizar una recuperación.

**Ejemplo 4.13****Cálculo del MTTF y del MTTDL en un RAID 5.**

Supongamos que cambiamos la organización del RAID del ejemplo anterior y lo convertimos en un RAID 5.



El tiempo medio hasta fallo de disco no cambia respecto del ejemplo anterior, porque sólo depende del número y tipo de discos que formen el RAID, no de su organización:

$$MTTF = \frac{MTTF \text{ (1 disco)}}{\text{Número de discos}} = \frac{1.5 \text{ millones de horas}}{6} = 250.000 \text{ horas}$$

Sin embargo en este caso, el tiempo medio hasta pérdida de datos es bastante inferior al tiempo medio hasta fallo de disco: sólo se pierden datos si al fallar un disco, mientras éste se recupera a partir de la información de redundancia, falla un segundo disco. Hay que recordar que con el tipo de código de paridad que se utiliza en un RAID 5, sólo se puede recuperar un fallo, no dos.

Si la recuperación tras el primer fallo se realiza de manera on line, el RAID sigue funcionando mientras se recupera la información del disco que ha fallado, por lo que la recuperación de los 40 GB sólo pueden utilizar un 10% del ancho de banda total:

$$t \text{ recuperación (on line)} = \frac{\text{capacidad de disco}}{\text{ancho de banda de recuperación}} = \frac{40 \text{ GB}}{0.1 \cdot 320 \text{ MB/s}} = 1250 \text{ s}$$

Durante este tiempo, el RAID está funcionando sólo con 5 discos duros, y el tiempo que en media tarda un disco de entre estos 5 en fallar es:

$$MTTF_5 = \frac{MTTF \text{ (1 disco)}}{\text{Número de discos}} = \frac{1.5 \text{ millones de horas}}{5} = 300.000 \text{ horas}$$

Por lo tanto la probabilidad de que en los 1250 s que se tarda en recuperar el disco que ha fallado, falle uno de los otros cinco discos que siguen funcionando es:

$$p = t \text{ recuperación (on line)}/MTTF_5 = 1250 \text{ s}/(300.000 \text{ horas} \cdot 3600) \text{ s} = 1.16 \cdot 10^{-6}$$

Por lo tanto, el tiempo medio hasta pérdida de datos es:

$$MTTDL(\text{on line}) = \frac{MTTF}{p} = \frac{250.000 \text{ horas}}{1.16 \cdot 10^{-6}} = 2.16 \cdot 10^{11} \text{ horas}$$

Si por el contrario la recuperación tras el primer fallo se realiza de manera off line, la recuperación es mucho más rápida ya que tiene todo el ancho de banda disponible:

$$t \text{ recuperación (off line)} = \frac{\text{capacidad de disco}}{\text{ancho de banda de recuperación}} = \frac{40 \text{ GB}}{320 \text{ MB/s}} = 125 \text{ s}$$

Durante este tiempo, el RAID está funcionando sólo con 5 discos duros, y el tiempo que en media tarda un disco de entre estos 5 en fallar es:

$$MTTF_5 = \frac{MTTF \text{ (1 disco)}}{\text{Número de discos}} = \frac{1.5 \text{ millones de horas}}{5} = 300.000 \text{ horas}$$

Por lo tanto la probabilidad de que en los 125 s que se tarda en recuperar el disco que ha fallado, falle uno de los otros cinco discos que siguen funcionando es:

$$p = t \text{ recuperación (on line)}/MTTF_5 = 125 \text{ s}/(300.000 \text{ horas} \cdot 3600) \text{ s} = 1.16 \cdot 10^{-7}$$

Por lo tanto, el tiempo medio hasta pérdida de datos es:

$$MTTDL \text{ (on line)} = \frac{MTTF}{p} = \frac{250.000 \text{ horas}}{1.16 \cdot 10^{-7}} = 2.16 \cdot 10^{12} \text{ horas}$$

Es decir, al hacer la recuperación off line la recuperación es más rápida, y esto disminuye la probabilidad de que ocurra un segundo fallo mientras se recupera el primero, lo que al final se traduce en un MTTDL, en este ejemplo, un orden de magnitud mayor.

## 4.6 Técnicas de optimización para la gestión de E/S

La principal optimización que se incorpora actualmente es la de un procesador de E/S, en el propio dispositivo periférico o en su interfaz, que haga que el procesador ni siquiera intervenga en las fases de inicialización y finalización de la transferencia de E/S.

Por tanto, en este tipo de gestión optimizada, existe un procesador con autonomía propia para realizar estas funciones sin necesidad de intervención del procesador principal. En muchos casos, este procesador incluso dispondrá de una memoria o jerarquía de memoria propia. Y para su diseño se utilizarán el mismo tipo de técnicas que se han estudiado en los capítulos 1 y 3 de este libro.

Obviamente, no es una solución que se escoja en muchos casos debido a su elevado precio, pero en los casos concretos del sistema gráfico y del sistema de comunicaciones, ha resultado ser la única técnica de optimización que consigue las prestaciones adecuadas.

Por lo tanto hoy en día se pueden encontrar en multitud de arquitecturas un procesador de propósito específico dedicado a las tareas de E/S del sistema gráfico (GPU o Graphic Processing Unit) y un procesador de propósito específico dedicado a las tareas de E/S del sistema de comunicaciones (NPU o Network Processing Unit), ambos de similares prestaciones a las del procesador pero diseñados para tareas mucho más concretas y por lo tanto con repertorios de instrucciones específicos para el tipo de aplicación que deben ejecutar.

### 4.6.1. Procesador para el sistema gráfico: GPU

Multitud de aplicaciones hoy en día exigen la generación de gráficos 3D en tiempo real (videojuegos, herramientas CAD, simuladores, etc). La realización de las tareas de E/S necesarias para conseguir la calidad de gráficos deseada en tiempo real consumiría demasiados recursos del procesador de propósito general por lo que hace años se optó por utilizar un procesador específico para la realización de estas tareas, la GPU.

En arquitecturas que utilizan este tipo de procesador, la CPU envía primitivas al procesador gráfico para que procese los datos que describen la escena 3D que se desea mostrar en el display o monitor. Estas primitivas suelen ser llamadas a una API para simplificar la programación de las aplicaciones.

Los datos que describen la escena 3D representan la escena que se debe generar desde diferentes puntos de vista: su geometría, las fuentes de luz, los materiales, las texturas, la posición de la cámara, etc.

Normalmente la GPU comenzará a trabajar con estos datos en una primera etapa de Geometría, denominada así porque se trabaja casi siempre con triángulos y con sus vértices (se suele denominar vertex a un vértice de un triángulo que además de la información de su posición 3D suele incluir información de normales, color y texturas).

En esta primera etapa se aplican operaciones geométricas a los datos de entrada según las primitivas que llegan de la CPU. Estas operaciones pueden ser mover, rotar y proyectar objetos, mover la cámara, iluminar objetos, ignorar los objetos que están fuera de la escena según el punto de vista, etc.

A continuación se suele producir una etapa de Rasterización, en la que se convierte la salida de las operaciones geométricas en una imagen 2D formada por fragmentos discretos.

Después se pasa a la etapa de Fragmentos, en la se realiza todo el trabajo en dos dimensiones, se aplica la mayor parte de las texturas y se realiza la interpolación del color.

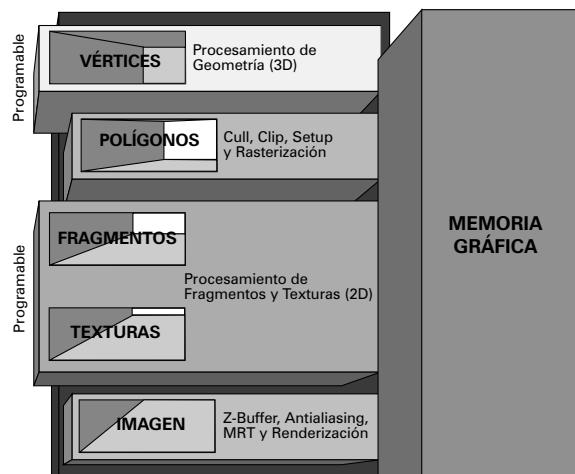
Por último, el renderizado se ocupa de hacer el test de profundidad para ver qué fragmentos son visibles y para calcular los píxeles que formarán la imagen final en el display.

Las GPUs actuales incorporan prácticamente todas las técnicas de diseño de procesadores que se han estudiado para la CPU, ya que si no sería imposible realizar todas estas tareas en tiempo real. Pero las aplican teniendo en cuenta las características concretas del trabajo que realizan, ya que en este caso, no se trata de un procesador que deba ser capaz de ejecutar cualquier aplicación, sólo debe ser capaz de ejecutar ciertas tareas concretas en las que el paralelismo de datos es muy habitual.

Además, las GPUs actuales son programables, aunque de nuevo su modelo de programación difiere ligeramente del de las CPUs debido a la especificidad de sus aplicaciones.

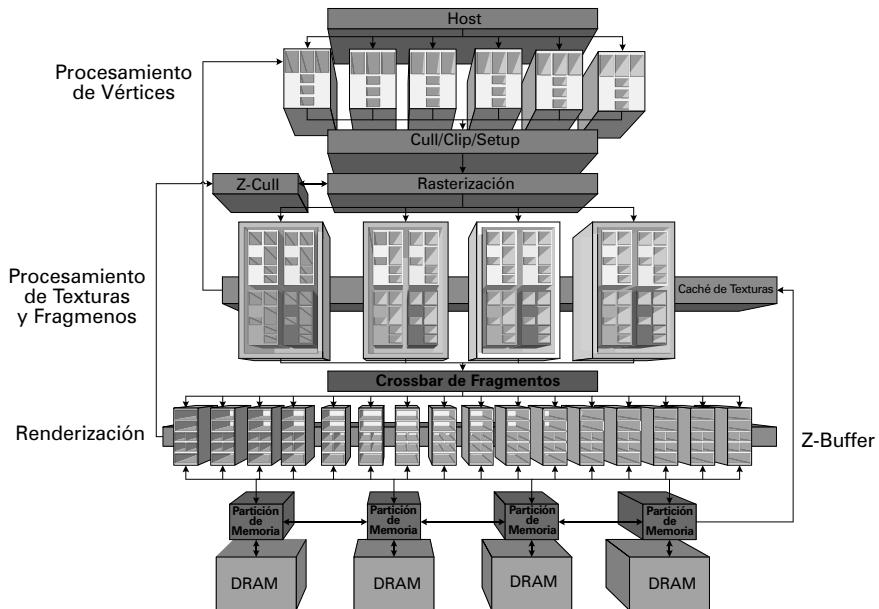
### CASO PRÁCTICO 4.3. GPUs de NVidia.

En la figura se muestra el esquema funcional del que se parte para diseñar las GPUs de la serie 6 de NVidia.



Como se puede observar, se trata de un diagrama de bloques que resume las etapas del cauce gráfico clásico que acabamos de estudiar. Lo más destacable es que se especifica desde un principio que el hardware que se encarga de la etapa de Geometría y de la etapa de procesamiento de Fragmentos y Texturas, debe ser programable.

Este esquema funcional se implementa con la arquitectura que se muestra en la siguiente figura.



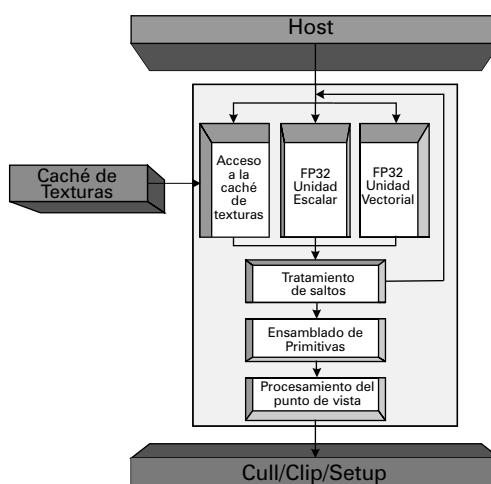
Se puede observar que el diseño de la GPU se basa en dos tipos de módulos programables: los procesadores de vértices (en la etapa de Procesamiento de Vértices) y los procesadores de Fragmentos (en la etapa de Procesamiento de Texturas y Fragmentos).

El resto del hardware está diseñado para realizar siempre las mismas tareas. Además se observa que la jerarquía de memoria de la GPU incluye dos memorias caché, la de texturas y el Z-buffer que se utiliza durante la renderización, y una memoria DRAM que es el equivalente a su memoria principal (se suele denominar memoria gráfica).

En la serie 6 de NVidia los procesadores de vértices pueden acceder a la caché de texturas en algunos casos concretos. Aunque lo normal es que se acceda desde los procesadores de fragmentos. En este caso se hace a través de un hardware específico que prepara los datos y los pone en el formato adecuado.

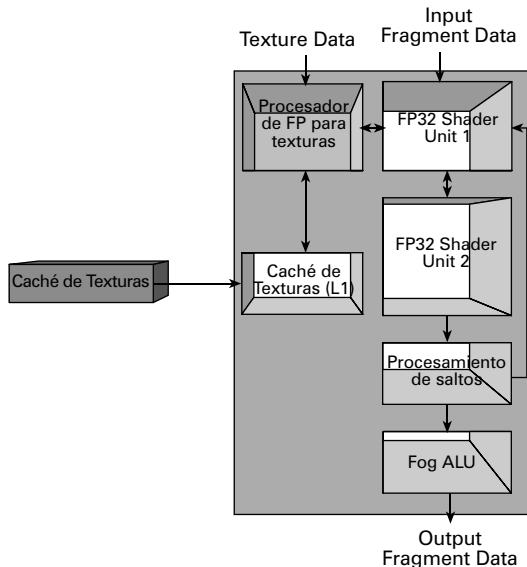
Los procesadores de vértices y fragmentos tienen repertorios de instrucciones diferentes. Ambos trabajan con tipos de datos en coma flotante (16, 24 o 32 bits), de manera que los números enteros se simulan. El hardware no programable puede funcionar con estos mismos tipos o con coma fija.

Los seis procesadores de vértices son completamente independientes. Cada uno de ellos tiene la siguiente arquitectura:



En la GPU se aplica un programa a cada vértice de entrada: transformaciones, skinning, etc. Para ello se emplean dos unidades funcionales en coma flotante, una escalar y otra vectorial. Y si es necesario, se accede a la caché de texturas. En la ejecución de estos programas se hace un tratamiento muy sencillo de las instrucciones de salto. A continuación, se ensamblan primitivas a partir de los vértices (puntos, líneas y triángulos) y se eliminan las primitivas que no serán visibles teniendo en cuenta el punto de vista.

En cuanto a la arquitectura de los procesadores de fragmentos:



De nuevo se aplica un programa a cada fragmento de entrada. En este caso los procesadores no son independientes, ya que se agrupan de cuatro en cuatro para poder aprovechar la localidad espacial trabajando con los píxeles potenciales en  $2 \times 2$ .

Todos los procesadores de fragmentos incorporan una caché de nivel 1 de texturas y un módulo para el tratamiento de los saltos, además de dos unidades funcionales en FP que pueden trabajar en paralelo, otra unidad funcional en FP diseñada específicamente para las operaciones típicas con texturas y una ALU, también específica para cierto tipo de operaciones muy comunes en esta etapa del cauce.

Cuando NVidia diseña las GPUs de su nueva serie, la serie 7, no se realiza un cambio espectacular en la arquitectura. Simplemente se añaden 2 nuevos procesadores de vértices y 2 (x4) nuevos procesadores de fragmentos. Además, se rediseñan los módulos hardware que supusieron un cuello de botella en la serie 6. En concreto, se mejora el rendimiento de los procesadores de fragmentos en la realización de operaciones matemáticas (añadiendo dos mini-ALUs en estos procesadores que se comportan como coprocesadores para las dos unidades funcionales de FP en las operaciones complejas) y se optimizan las técnicas de prebúsqueda de la caché de texturas.

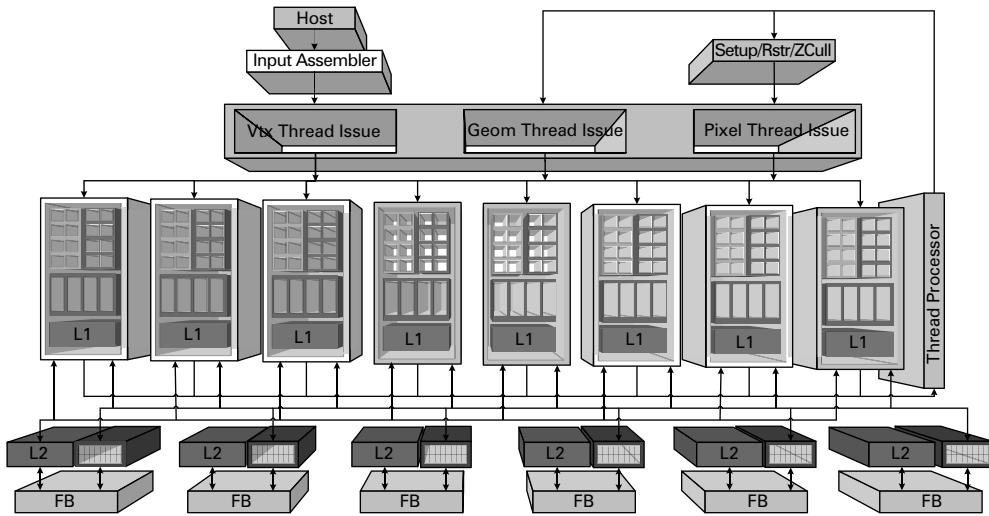
Las arquitecturas de las series anteriores, la 6 y la 7 presentan dos problemas importantes:

- El desequilibrio en la carga de los procesadores de vértices y fragmentos, ya que no todas las imágenes presentan la misma complejidad en tres y dos dimensiones y en muchos casos alguno de los dos tipos de procesadores queda infroutilizado.
- La diferencia entre los repertorios de instrucciones de ambos tipos de procesadores.

Por eso en la serie 8 se busca una arquitectura unificada que no distinga entre ambos tipos de procesadores. Se diseña una única unidad de proceso capaz de realizar todo tipo de tareas y que sea coherente con el cauce gráfico definido por DirectX 10.

Para ello se introducen en las nuevas GPUs tres etapas programables: vértices, geometría y fragmentos. Todas ellas necesitan el mismo repertorio y prácticamente los mismos recursos.

El núcleo de ejecución de las nuevas arquitecturas unificadas se basa en un nuevo hardware que gestiona los tres tipos de threads que se ejecutan en la GPU: de vértices, de geometría y de fragmentos. En el ejemplo de la figura este hardware reparte el trabajo entre 8 módulos, cada uno de ellos contiene 16 Streaming Processors (SP), 4 Texture Filtering Units (TF) y el primer nivel de memoria caché compartida (L1).



Aunque sin entrar en mucho detalle relacionado con las características específicas de las arquitecturas para gráficos, se puede observar gracias a este recorrido por las arquitecturas de las GPUs de NVidia para sus series 6, 7 y 8, que las técnicas empleadas para el diseño de procesadores de E/S son las mismas que para los procesadores de propósito general estudiadas en los capítulos 1 y 3 de este libro. Aunque, claro está, aprovechando en las optimizaciones de los diseños la especificidad de las aplicaciones que se van a ejecutar. En este tipo de procesadores se suelen utilizar muy pocas técnicas dinámicas (planificación de instrucciones, predicción de saltos, especulación, etc) y sin embargo, por el tipo de aplicación que ejecutan, se suele buscar la explotación masiva del paralelismo de datos, ya que es típico realizar las mismas operaciones sobre grandes conjuntos de datos. En resumen podemos encontrar:

- Segmentación.
  - A alto nivel (cauce gráfico).
  - A bajo nivel (dentro de los procesadores de vértices y de fragmentos).
- Tratamiento y predicción de saltos sencilla.
- Paralelismo.
  - A alto nivel (soluciones que incluyen 2 GPUs dentro de una tarjeta o que permiten que dos tarjetas gráficas trabajen en paralelo con SLI o Crossfire).
  - A bajo nivel (varios procesadores trabajando en paralelo en cada etapa, sobre diferentes vértices o fragmentos, unidades vectoriales, rutas de datos multifuncionales).
- Optimizaciones en la memoria.
  - Prebúsqueda de memoria en la caché de texturas.
  - Tecnologías de memoria gráfica que incluyen el entrelazado.
  - Cuatro accesos en paralelo a la memoria principal de la tarjeta (cuatro canales independientes).

## Resumen de decisiones de diseño de técnicas de aumento de prestaciones para memoria y E/S

### AUMENTO DE PRESTACIONES DE LA MEMORIA CACHÉ

| Decisión                               | Alternativas                                                                                                                    | Decisiones asociadas                        |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------|
| Cómo reducir la penalización por fallo | Dar prioridad a los fallos<br>Optimizaciones del buffer de escritura<br>Caché de víctimas<br>Utilización de caché no bloqueante |                                             |
| Cómo reducir la tasa de fallos         | Caché predictiva                                                                                                                | Algoritmo de predicción                     |
|                                        | Caché pseudoasociativa                                                                                                          | Algoritmo de segundo emplazamiento          |
|                                        | Prebúsqueda                                                                                                                     | HW o SW, algoritmo de prebúsqueda           |
|                                        | Caché de trazas (para instrucciones)                                                                                            |                                             |
| Cómo reducir el tiempo de acierto      | Mejoras en la traducción de DV a DF                                                                                             |                                             |
|                                        | Segmentación de la caché                                                                                                        | Diseño de la segmentación, número de etapas |
|                                        | Caché de trazas (para instrucciones)                                                                                            |                                             |

### AUMENTO DE PRESTACIONES DE LA MEMORIA PRINCIPAL

| Decisión                           | Alternativas                                                                             | Decisiones asociadas                        |
|------------------------------------|------------------------------------------------------------------------------------------|---------------------------------------------|
| Cómo reducir la latencia de acceso | Cualquiera que implique una mejora en la tecnología con la que está fabricada la memoria |                                             |
| Cómo aumentar el ancho de banda    | Ensanchado de la memoria principal                                                       | Número de palabras                          |
|                                    | Entrelazado de la memoria principal                                                      | A alto o bajo nivel, número de bancos       |
|                                    | Canales de memoria independientes                                                        | Número de canales                           |
|                                    | Segmentación de la memoria principal                                                     | Diseño de la segmentación, número de etapas |

### AUMENTO DE PRESTACIONES DE BUSES DE E/S

| Decisión                        | Alternativas                                                       | Decisiones asociadas                                                        |
|---------------------------------|--------------------------------------------------------------------|-----------------------------------------------------------------------------|
| Cómo aumentar el ancho de banda | Serialización y aumento de frecuencia de funcionamiento            | Compromiso entre ancho de datos, frecuencia, disipación de potencia y ruido |
|                                 | Señalización diferencial y aumento de frecuencia de funcionamiento | Compromiso entre ancho de datos, frecuencia, disipación de potencia y ruido |
|                                 | Utilización de protocolos de comunicaciones de alto rendimiento    | Propietarios o estándares (PCI-Express, Hipertransporte)                    |

**AUMENTO DE PRESTACIONES DE DISPOSITIVOS DE E/S: ALMACENAMIENTO**

| Decisión                   | Alternativas                      | Decisiones asociadas                                             |
|----------------------------|-----------------------------------|------------------------------------------------------------------|
| Número de discos del RAID  |                                   |                                                                  |
| Granularidad del stripping | Fina<br>Gruesa<br>Sin especificar |                                                                  |
| Información de redundancia | Mirroring                         |                                                                  |
|                            | Códigos de paridad                | Hamming, P+Q                                                     |
| Recuperación               | On-line                           | Porcentaje del ancho de banda del bus dedicado a la recuperación |
|                            | Off-line                          |                                                                  |

**AUMENTO DE PRESTACIONES EN LA GESTIÓN DE E/S**

| Decisión                              | Alternativas                                                               | Decisiones asociadas |
|---------------------------------------|----------------------------------------------------------------------------|----------------------|
| Incorporación de un procesador de E/S | Procesador de red (NPU)<br>Procesador gráfico (GPU)<br>Diseños específicos |                      |

**BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS**

- HANDY, J. (1998): *The cache memory book* (2.<sup>a</sup> ed.), Morgan Kaufmann.
- HENNESSY, J. L. & PATTERSON, D. A. (2007): *Computer Architecture: A Quantitative Approach* (4.<sup>a</sup> ed.), Morgan Kaufmann.
- Hypertransport Consortium*. <http://www.hypertransport.org/>.
- HWANG, K. (1992): *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw Hill.
- JACOB, B.; SPENCER NG. & WANG D. (2007): *Memory Systems: Cache, DRAM, Disk*, Morgan Kaufmann.
- Intel Developer Network for PCI Express Architecture*. <http://www.intel.com/technology/pciexpress/devnet/>.
- LUEBKE, D. & HUMPHREYS, G. (2007): «How GPUs work», *Computer*, 40(2), 96-100.
- MONTRYM, J. & MORETON, H. (2005): «The GeForce 6800», *IEEE Micro*, 25(2), 41-51.
- PARHAMI, B. (2007): *Arquitectura de Computadoras. De los microprocesadores a las supercomputadoras* (1.<sup>a</sup> ed.), McGraw Hill.
- SHIVA, S. G. (2005): *Advanced Computer Architectures*, CRC Press.
- SIMA, D.; FOUNTAIN, T. & KARSUK, P. (1997): *Advanced Computer Architectures: A Design Space Approach*, Addison Wesley.

## PROBLEMAS

- 4.1.** Se diseña una jerarquía de memoria con un único nivel de memoria caché, con instrucciones y datos separados. El procesador realiza  $5 \cdot 10^6$  referencias por segundo a datos (de las cuales el 80% son lecturas y el 20% son escrituras) y  $10^7$  referencias por segundo a instrucciones (obviamente, todas lecturas). La memoria caché de instrucciones tiene un tiempo de acceso de 1.5 ns, una tasa de fallos del 8% y el tamaño de bloque es de 32 palabras. La memoria caché de datos tiene un tiempo de acceso de 1.2 ns, una tasa de fallos del 9% y un tamaño de bloque de 16 palabras. La latencia de memoria principal es de 92 ns y se utiliza la técnica de palabra crítica primero. Es decir se trata de la misma jerarquía de memoria que se diseñó en el problema 2.7.
- Calcular el tiempo medio de acceso a memoria para instrucciones y para datos si la caché es de escritura directa y se han incluido en el diseño las siguientes mejoras:
    - La caché de datos se convierte en una caché sin asignación en escritura.
    - Una caché de víctimas para la caché de datos que puede resolver los fallos el 45% de las veces en un tiempo de 13 ns.
    - Una caché de trazas que sustituye a la caché de instrucciones y que consigue un tiempo de acierto de 0.7 ns y una tasa de fallos del 4%.
  - Calcular el ancho de banda con memoria principal consumido con estas mejoras.
  - Si resolviste el problema 2.7, compara los resultados que obtuviste con los obtenidos ahora gracias a las mejoras realizadas en esta jerarquía.
- 4.2.** Suponiendo que el bus de memoria es de 64 bits de ancho, y funciona a una frecuencia de 667 MHz, calcular el ancho de banda que queda libre para que el sistema de E/S acceda a memoria principal en el problema anterior. Discutir el resultado obtenido y proponer modificaciones al diseño realizado para mejorarlo.
- 4.3.** Se diseña una jerarquía de memoria con dos niveles de caché y las siguientes características:
- Caché de nivel 1 deemplazamiento directo y escritura directa, unificada, con un tiempo de acceso de 1 ns, una tasa de fallos del 11% y tamaño de bloque de 16 palabras.
  - Caché de nivel 2 asociativa por conjuntos de 4 vías y de post-escritura, unificada, con un tiempo de acceso de 9 ns, una tasa de fallos del 6% y un tamaño de bloque de 32 palabras.
  - Memoria principal con una latencia de acceso de 92 ns.
  - Se utiliza la técnica de palabra crítica primero.
  - Tiempos de transferencia de información entre niveles despreciables.
- Sabiendo que el 75% de los accesos a memoria son lecturas y el 25% restante son escrituras (es decir, se tiene de nuevo la jerarquía diseñada en el problema 2.8), calcular el tiempo medio de acceso a memoria si se introducen en el diseño las siguientes mejoras:
- Se implementa un algoritmo de prebúsqueda para la caché de nivel 1 que consigue reducir su tasa de fallos a un 9%.
  - Se da prioridad a los fallos de caché del segundo nivel añadiendo un buffer de escritura con una tasa de fallos del 18% y en el que se tardan 10 ns en volcar un bloque sucio.
- 4.4.** Se diseña una jerarquía de memoria con dos niveles de caché que utilizan la técnica de palabra crítica primero y las siguientes características:

- Caché de nivel 1 de emplazamiento directo y escritura directa, unificada, con un tiempo de acceso de 1 ns, una tasa de fallos del 5% y tamaño de bloque de 16 palabras.
- Caché de nivel 2 asociativa por conjuntos de 4 vías y de post-escritura, unificada, con un tiempo de acceso de 9 ns, una tasa de fallos del 12% y un tamaño de bloque de 32 palabras.
- Memoria principal con una latencia de acceso de 92 ns.
- Tiempos de transferencia de información entre niveles despreciables.

Ahora vamos a centrarnos en reducir en todo lo posible la penalización por fallo de la caché de nivel 2. Para ello:

- a) Introducimos en la jerarquía de memoria un buffer de escritura que permite dar prioridad a los fallos de la caché de nivel 2. Volcar un bloque sucio a este buffer supone un tiempo de 14 ns y el buffer evita hacer este volcado a memoria principal el 82% de las veces.
- b) Añadimos una caché de víctimas también al segundo nivel de caché, que permite resolver los fallos el 50% de las ocasiones en un tiempo de 18 ns.

Calcular el tiempo medio de acceso a memoria cuando se incorporan estas optimizaciones.

**4.5.** Todavía no hemos optimizado suficientemente esta jerarquía de memoria, así que intentamos introducir nuevas modificaciones. El nivel 1 se convierte en una caché pseudoasociativa y el nivel 2 en una caché predictiva. Con estas transformaciones tenemos:

- Cuando la caché de nivel 1 acierta, se sigue comportando como una caché directa y su tiempo de acceso no varía. Esto ocurre el 95% de las veces. Cuando falla, podemos buscar en una segunda localización, lo que implica que en estos casos el tiempo de acceso es de 2 ns. Pero la tasa de fallos global se consigue reducir a un 3.5%.
- Cuando la caché de nivel 2 acierta la predicción de caché predictiva, el tiempo de acceso es de 7.5 ns. Esto ocurre en un 65% de los accesos, en el resto, no varía el tiempo de acceso.

Calcular el tiempo medio de acceso a memoria con estas nuevas optimizaciones. ¿Se consigue una mejora significativa respecto de los resultados obtenidos en el problema 4.4?

**4.6.** Se incluye en una computadora una jerarquía de memoria con las siguientes características:

- Caché de instrucciones de nivel 1 con tiempo de acceso de 1 ns, tasa de fallos del 6% y tamaño de bloque de 16 palabras.
  - Caché de datos de nivel 1 con escritura directa, tiempo de acceso de 2.5 ns, tasa de fallos del 8% y tamaño de bloque de 16 palabras.
  - Buffer de escritura que permite realizar las escrituras directas el 85% de las veces y con un tiempo de acceso de 1 ns.
  - TLB con un tiempo de acceso de 1.5 ns, tasa de fallos del 9% y que en los fallos necesita hacer en media 1.2 accesos a memoria principal para realizar la traducción de dirección virtual a dirección física.
  - Memoria principal con una latencia de acceso de 76 ns.
  - Transferir una palabra de memoria principal a memoria caché supone 1.3 ns.
- a) Calcular el tiempo medio de acceso a memoria para instrucciones y para datos (teniendo en cuenta que el 85% de los accesos a datos son lecturas y el 15% son escrituras) si en las cachés de instrucciones y datos se utiliza la técnica de palabra crítica primero.
  - b) Calcular de nuevo este tiempo si se introducen las siguientes optimizaciones en la jerarquía:

- El buffer de escritura se optimiza con la técnica de write-merging de manera que permite realizar las escrituras directas el 95% de las ocasiones.
  - Se utiliza prebúsqueda hardware en la caché de datos y se reduce su tasa de fallos al 6.5%.
  - La caché de instrucciones se convierte en una caché trazas, con tiempo de medio de acceso a instrucción de 0.2 ns y tasa de fallos del 4%.
  - Se accede a la caché de datos con direcciones virtuales.
- 4.7.** En una jerarquía de memoria con un nivel de caché se decide optimizar el ancho de banda de la memoria principal. Para ello se barajan tres alternativas:
- Ensanchar toda la jerarquía a un ancho de 4 palabras.
  - Utilizar una memoria principal entrelazada de 4 bancos.
  - Incorporar dos canales de memoria principal independientes a la jerarquía.
- Calcular la penalización por fallo de la caché si el tamaño de bloque es de 128 palabras, el tiempo de direccionamiento de una palabra es despreciable, la latencia de memoria principal es de 76 ns y el tiempo de transferencia de una palabra por el bus es de 1.3 ns. Hacer este cálculo con las tres optimizaciones posibles y discutir cuál sería más conveniente para nuestro diseño.
- 4.8.** Incorporar a la jerarquía de memoria diseñada en el problema 4.5 la optimización escogida en el problema 4.6 y calcular la ganancia obtenida.
- 4.9.** Escoger una placa de base actual para PC, dibujar la jerarquía de buses que incorpora y calcular los anchos de banda de los buses más importantes. ¿Qué buses incorporan técnicas de aumento de prestaciones de las estudiadas en este capítulo? ¿Cuáles siguen siendo buses tradicionales?
- 4.10.** Se diseña un RAID5 de 8 discos duros SCSI de 100 GB cada uno. El tiempo medio hasta el fallo (Mean Time To Failure, MTTF) de estos discos es de 1.2 millones de horas. Además, si un disco falla, la recuperación se puede hacer offline o se puede hacer online.
- a) ¿Cuál es el tiempo medio que pasará antes de que uno de los discos falle y sea necesaria una recuperación?
  - b) ¿Qué operaciones de lectura y escritura serán necesarias para realizar esa recuperación? ¿Cuáles de ellas pueden realizarse en paralelo?
  - c) Si el bus es un UltraSCSI 320 con un ancho de banda total de 320 MB/s, ¿cuánto tiempo tardarán las recuperaciones offline y online? Tener en cuenta que para la recuperación online sólo se deja un 15% del ancho de banda total del bus.
  - d) Calcular el MTDL para la recuperación offline y para la online.
- 4.11.** Escoger una tarjeta gráfica actual y buscar información acerca de su arquitectura, en concreto acerca de su GPU y de su memoria de vídeo. ¿Qué técnicas de diseño de las aprendidas hasta el momento incorpora esta arquitectura? ¿Qué peculiaridades aparecen en el diseño debido a su aplicación específica?

## AUTOEVALUACIÓN

1. ¿Cómo funciona una caché de víctimas? ¿Por qué reduce la penalización por fallo de la memoria caché?
2. ¿Por qué la mayor parte de las cachés actuales son no bloqueantes?
3. ¿Qué es una caché pseudoasociativa y cómo funciona?
4. ¿Qué alternativas existen para evitar que la traducción de dirección virtual a dirección física afecte negativamente al tiempo de acceso a la memoria caché? ¿Qué ventajas e inconvenientes tienen cada una de ellas?
5. ¿Cómo funciona una caché de trazas? ¿Qué aspectos del funcionamiento de la memoria caché ayuda a optimizar?
6. ¿Qué diferencias hay entre una memoria principal entrelazada con 4 bancos y otra con 4 canales de memoria independientes?
7. ¿Por qué en la última generación de consolas algunos procesadores incorporan almacenamiento local en lugar de memoria caché?
8. ¿En qué consiste la señalización diferencial? ¿Por qué permite aumentar el ancho de banda de los buses de E/S?
9. ¿Qué técnicas de aumento de prestaciones para buses incorporan tecnologías como PCI-Express o Hipertransporte?
10. ¿Qué diferencias hay entre un RAID 3 y un RAID 4?



# 5

---

# Sistemas multiprocesador y multicamputador

---

## Contenidos

- 5.1. Clasificación de arquitecturas con varios procesadores
- 5.2. Redes dentro de arquitecturas de computadoras
- 5.3. Diseño de arquitecturas de memoria compartida
- 5.4. Diseño de arquitecturas de memoria compartida-distribuida
- 5.5. Diseño de arquitecturas de memoria distribuida

A pesar de las grandes mejoras que se han producido en el rendimiento de arquitecturas monoprocesador gracias a los avances en las tecnologías de fabricación y a las técnicas de diseño estudiadas en los anteriores capítulos, para algunas aplicaciones el rendimiento obtenido con ellas todavía no es suficiente. Las mejoras en microprocesadores cada vez son más sofisticadas, y cada avance implica crecer en complejidad y consumo de área y potencia (es decir, en coste).

Sin embargo, se ha producido una clara mejora en el software y en los sistemas operativos que hace más sencillo y económico explotar el paralelismo de las aplicaciones cuando se ejecutan sobre varios procesadores. Por tanto la utilización de arquitecturas compuestas por varios procesadores es una solución sencilla, natural y en general, con una buena relación coste-prestaciones.

En este capítulo se comienza realizando una clasificación de las arquitecturas compuestas por varios procesadores, atendiendo principalmente a su modelo de memoria (compartida o distribuida). A continuación se presentan los conceptos básicos de redes dentro de arquitecturas de computadoras, ya que la red es el nuevo componente que aparece en este tipo de arquitecturas y cuyos aspectos de diseño pueden tener una gran influencia en el rendimiento global de la arquitectura.

Y a continuación se pasa a analizar en profundidad el diseño de arquitecturas compuestas por más de un procesador, empleando la clasificación proporcionada al principio del capítulo.

Por lo tanto se estudian las técnicas y decisiones de diseño asociadas a las arquitecturas de memoria compartida (multiprocesadores) y a las de arquitectura de memoria distribuida (multicomputadores). En ambos casos se toman como punto de partida los problemas típicos que es necesario resolver en cada tipo de arquitectura al incluir más de un procesador y por utilizar un determinado modelo de memoria, para centrarse en el estudio de aquéllos que pueden ser resueltos desde el propio hardware.

## 5.1 Clasificación de arquitecturas con varios procesadores

Hasta ahora todas las técnicas estudiadas en este libro se han centrado en explotar el paralelismo a nivel de instrucción (grano fino) para aumentar el rendimiento de las arquitecturas con un único procesador, de manera que el propio hardware detecta instrucciones independientes y planifica su ejecución en paralelo.

Pero los sistemas compuestos por varios procesadores explotan el paralelismo a nivel de proceso (grano grueso), asociado a la replicación de los recursos de cómputo. En este caso el paralelismo se detecta a alto nivel, por parte del compilador y del sistema operativo, aunque también puede programarse de forma explícita. A pesar de esta colaboración del compilador y del sistema operativo todavía existen muchos problemas que deben resolverse desde el hardware y muchas optimizaciones relacionadas con su diseño.

Este tipo de arquitecturas compuestas por varios procesadores se utilizan hoy en día en multitud de entornos diferentes, desde las aplicaciones de cálculo científico hasta los servidores de aplicaciones, bases de datos e Internet, pasando por el procesamiento de gráficos y multimedia o las aplicaciones de alta disponibilidad.

Para estudiar las diferentes alternativas de diseño en el caso de arquitecturas paralelas, existen distintas clasificaciones. La clásica es la taxonomía de Flynn, que distingue entre cuatro categorías de arquitecturas atendiendo a su explotación del paralelismo de datos y del paralelismo funcional: SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single data) y MIMD (Multiple Instruction Multiple Data).

Sin embargo en este libro se utiliza la clasificación que atiende al diseño de la jerarquía de memoria del sistema. Según esta clasificación se pueden diseñar arquitecturas de memoria compartida, compartida-distribuida o distribuida. Pero además hoy en día se debe distinguir en esta clasificación entre las arquitecturas tradicionales y las arquitecturas on-chip, en las que a pesar de tener varios procesadores, no es dentro de un sistema de gran tamaño sino de un único chip o empaquetado que integra todos los componentes de la arquitectura.

### 5.1.1. Arquitecturas de memoria compartida

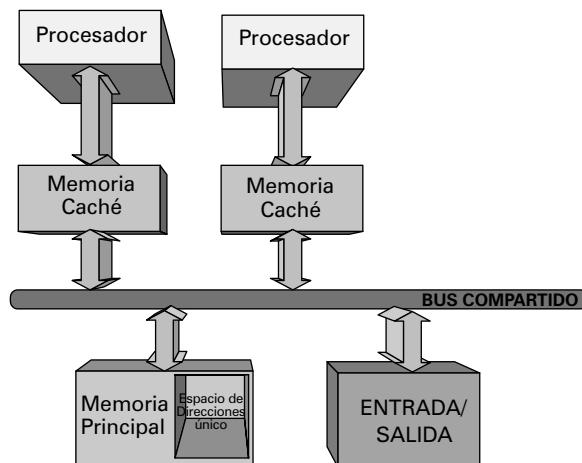
En este caso la arquitectura está compuesta por un conjunto de procesadores (casi siempre homogéneos) que comparten una única memoria centralizada, normalmente la memoria principal.

En este tipo de diseño el espacio de direcciones físicas es global: todos los nodos tienen una visión de la memoria simétrica. Por ello estas arquitecturas suelen denominarse multiprocesadores o arquitecturas SMP (Symmetric MultiProcessing). Además, como los tiempos de acceso a todas las direcciones de memoria son uniformes, es decir, siempre se tarda lo mismo en realizar un acceso a memoria principal, también pueden denominarse arquitecturas UMA (Uniform Memory Access). A veces también se denominan sistemas fuertemente acoplados, ya que todos los procesadores comparten un componente tan crítico como la memoria principal.

Los diferentes procesadores y módulos de memoria se interconectan a través de un bus o red de medio compartido. De la misma forma, los mecanismos de E/S también suelen ser compartidos y se encuentran conectados a este bus (figura 5.1). Los procesos que se ejecutan en paralelo en los diferentes procesadores que componen la arquitectura se sincronizan y comunican a través de la memoria compartida e interactúan mediante variables compartidas. Es decir, la comunicación entre procesos es implícita y se realiza con instrucciones de load y store (leyendo y escribiendo variables compartidas en la memoria centralizada).

Gracias a este tipo de comunicación entre procesos, la programación de estos sistemas es muy similar a la de sistemas monoprocesador, lo que suele suponer una gran ventaja. Además, la comunicación implícita entre procesos a través de la memoria, ahorra la sobrecarga en la red de comunicaciones de la arquitectura. Esto hace que los desarrolladores se tengan que centrar sólo en el acceso a recursos críticos (memoria y E/S, ya que son los recursos compartidos) para optimizar el rendimiento de sus códigos.

Como se estudiará más adelante en profundidad (sección 5.3), la compartición de la memoria principal conlleva siempre la aparición de tres problemas que deben resolverse para el correcto funcionamiento de la arquitectura: la coherencia, la consistencia y la sincronización. Pero estos tres problemas que presenta la compartición de la memoria pueden ser resueltos o bien por el sistema operativo, el compilador o la programación explícita (consistencia y sincronización) o bien por el propio hardware (coherencia).



**FIGURA 5.1**

Arquitectura de memoria compartida.

Aunque en las arquitecturas de memoria compartida suelen utilizarse cachés muy optimizadas para minimizar la tasa de fallos y así reducir el tráfico en el bus compartido y satisfacer las demandas de los procesadores, el principal problema que suelen presentar estos diseños está en su baja escalabilidad. El bus compartido tiene un ancho de banda limitado y se suele convertir en un cuello de botella. Además contri-

buyen a esta baja escalabilidad los problemas de contención por los recursos compartidos, ambos críticos, la memoria y la E/S. Para mejorar la escalabilidad de este tipo de arquitecturas casi todos los esfuerzos se centran en mejorar la jerarquía de memoria, para evitar los accesos a la memoria principal compartida, y en mejorar el rendimiento del bus compartido, tanto su latencia como su ancho de banda.

Aún así en muchos casos es necesario diseñar arquitecturas de memoria compartida-distribuida para superar esta limitación en la escalabilidad.

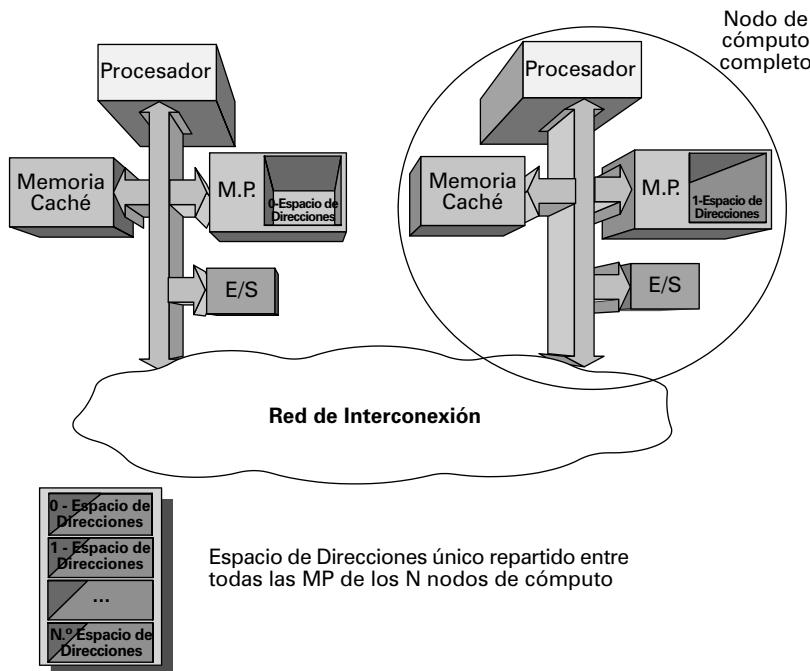


FIGURA 5.2

Arquitectura de memoria compartida-distribuida.

### 5.1.2. Arquitecturas de memoria compartida-distribuida

En este tipo de arquitectura cada nodo es un procesador con su propia memoria principal y sus propios mecanismos de E/S. Los nodos trabajan independientemente pero están conectados mediante una red de comunicaciones que les permite intercambiar información mediante paso de mensajes (figura 5.2).

Sin embargo existe un único espacio de direcciones de memoria, es decir, la misma dirección física en dos nodos del sistema corresponde a la misma ubicación de memoria principal. Por lo tanto, se distribuye estáticamente entre memorias principales físicamente diferentes un único espacio de direcciones, de manera que a cada memoria principal de cada nodo le corresponde almacenar una parte de este espacio único y global de memoria.

Desde el punto de vista del programador la memoria es compartida y cualquier procesador puede acceder a cualquier posición de memoria mediante una instrucción de load o store. Por lo tanto, para utilizar un sistema de este tipo tiene que haber una capa de middleware entre las aplicaciones y el hardware que haga que la distribución física de la memoria sea transparente a los usuarios.

En este caso las arquitecturas suele denominarse NUMA (Non Uniform Memory Access) ya que el tiempo de acceso a memoria principal no es siempre el mismo, depende de la ubicación del dato. Un acceso a memoria puede ser:

- **Local.** A la memoria principal del propio nodo.
- **Remoto.** A la memoria principal de otro nodo, por lo tanto el load o el store tiene que ser traducido a un mensaje de petición por la red, para acceder a la memoria remota en la que se encuentra almacenado el dato.

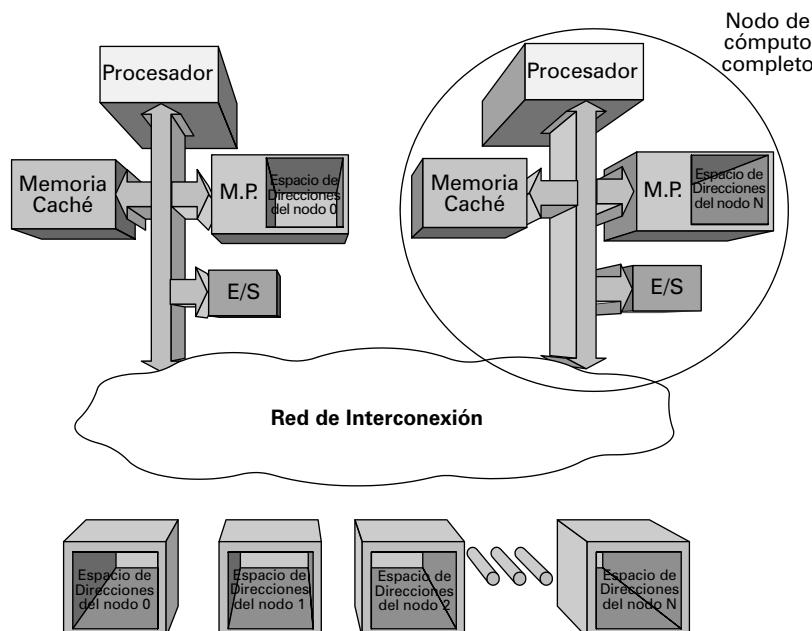
Como se puede observar, la comunicación entre procesos es implícita, a través del espacio de direcciones de memoria compartido. Aunque en realidad las operaciones de carga/almacenamiento pueden implicar un intercambio de mensajes entre dos procesadores, como se ha mencionado este proceso es transparente a los programadores y usuarios y es realizado por algún tipo de middleware.

Con este tipo de arquitectura se intentan mantener las ventajas de las arquitecturas de memoria compartida, especialmente, el hecho de que la programación de estos sistemas sea muy similar a la de sistemas monoprocesador. Sin embargo, también se intenta superar su principal inconveniente, la baja escalabilidad. Al distribuir físicamente la memoria entre diferentes nodos y al utilizar una red que no esté basada en medio compartido, esta desventaja desaparece casi por completo.

Los grandes retos en este tipo de arquitectura siguen siendo la resolución de los problemas de coherencia, consistencia y sincronización (que aparecen debido a la compartición de un único espacio de direcciones) y el desarrollo del middleware que gestione las memorias físicamente distribuidas como si se tratara de una única memoria principal. También existe un reto importante relacionado con minimizar los efectos que la latencia de la red de comunicaciones añade a los accesos a memoria remotos.

### 5.1.3. Arquitecturas de memoria distribuida

En este tipo de arquitecturas de nuevo cada nodo es un procesador con su propia memoria principal y sus propios mecanismos de E/S. Los nodos trabajan independientemente pero están conectados mediante una red de comunicaciones que les permite intercambiar información mediante paso de mensajes (figura 5.3). A veces se denominan sistemas o arquitecturas débilmente acopladas.



**FIGURA 5.3**

Arquitectura de memoria distribuida.

Pero, en el caso de la memoria distribuida completamente, la memoria principal se encuentra dividida, física y lógicamente en espacios de direcciones físicas independientes. Es decir, la misma dirección física de memoria en diferentes procesadores hace referencia a distintas ubicaciones físicas, cada una dentro de la memoria principal local de cada nodo.

Como cada procesador puede acceder únicamente a su espacio de direcciones local, la comunicación entre procesos debe ser explícita mediante paso de mensajes.

En este tipo de arquitecturas es habitual que los nodos que componen el sistema sean heterogéneos o terminen siéndolo. Estas arquitecturas suelen denominarse multicomputadores, para dar una idea de que están compuestas por computadoras completas e independientes que trabajan conjuntamente para resolver un único problema. Como se verá en secciones posteriores, de nuevo será necesario un middleware entre las aplicaciones y el hardware que permita utilizar una arquitectura de este tipo como un único sistema y no como nodos independientes que simplemente están conectados por una red de comunicaciones.

Las principales ventajas de las arquitecturas de memoria distribuida son su escalabilidad y el uso de comunicación explícita, en muchos casos, mucho más fácil de comprender y depurar (aunque el modelo de programación es completamente diferente que el de sistemas monoprocesador y la transformación de los códigos para su ejecución no es obvia). En este caso, los desarrolladores se deben centrar sólo en el patrón de comunicaciones de las aplicaciones para optimizar el rendimiento de sus ejecuciones.

Además, el hardware de estas arquitecturas es más sencillo, eliminando la necesidad de resolver los problemas de coherencia de caché. Los problemas de sincronización y de consistencia tampoco aparecen en este tipo de arquitectura, ya que no se comparte ni la memoria ni su espacio de direcciones.

Sin embargo, aparecen otros problemas nuevos que sí es necesario resolver, en concreto, hay que tener en cuenta que la comunicación entre procesos introduce una mayor complejidad en la programación y que además introduce una latencia de comunicaciones que puede llegar a ser importante. Por último, mencionar que habrá que disponer de mecanismos que permitan a los usuarios tener una imagen de sistema único o repartir el trabajo entre los nodos que componen el sistema, por poner sólo algunos ejemplos.

#### 5.1.4. Arquitecturas on-chip de memoria compartida

Las espectaculares mejoras en la tecnología de fabricación de procesadores han hecho que, en lugar de aprovechar el incremento en la densidad de integración de transistores para implementar nuevas técnicas de aumento de prestaciones en sistemas monoprocesador, se haya aprovechado directamente para integrar varios núcleos de procesador dentro de un único chip. Es decir, para diseñar multiprocesadores on-chip (MPoC, MultiProcessor on Chip) o sistemas completos on-chip (SoC, System on Chip).

Aunque no es objeto de este libro entrar en las particularidades de diseño de este tipo de arquitectura, que son muchas, sí que trataremos de estudiar los aspectos comunes con las arquitecturas tradicionales.

En el caso de las arquitecturas on-chip de memoria compartida, todo lo explicado en la sección 5.1.1 es válido. Es decir, la comunicación entre procesadores es implícita a través de la memoria compartida y esta compartición hace que aparezcan los problemas de coherencia, consistencia y sincronización ya mencionados.

Sí que cabe mencionar dos particularidades de las arquitecturas de memoria compartida on-chip. La primera es que en este tipo de arquitecturas no siempre se comparte la memoria principal, en algunos casos se comparte algún nivel de memoria caché. Dependiendo de la filosofía de diseño del fabricante se compartirá un nivel más cercano o más lejano al procesador, pero normalmente el nivel que se comparte es el segundo o el tercero de caché.

La segunda, es que en el caso de arquitecturas on-chip de memoria compartida puede aparecer heterogeneidad con más facilidad que las arquitecturas tradicionales. Hay que tener en cuenta que existen diferentes maneras de diseñar multiprocesadores on-chip:

- Incluyendo en un chip los recursos de dos o más procesadores de manera que estos sean indistinguibles.

- Integrando dos o más núcleos de procesador independientes en el mismo chip.
- Empaquetando juntos dos o más procesadores.

Estas tres formas de diseño permiten variar con cierta facilidad las características de alguno de los núcleos del multiprocesador si se considera conveniente para aumentar sus prestaciones. Los posibles problemas introducidos por la heterogeneidad quedan minimizados al estar todos los recursos integrados en un mismo chip (o empaquetado), por lo que se pueden asumir mejor que en las arquitecturas tradicionales.

En las figuras 5.4 y 5.5 se observan las arquitecturas on-chip de memoria compartida más comunes. Aunque los primeros diseños fueron homogéneos, este tipo de diseño obliga a decidir si incluir en la arquitectura un número reducido de núcleos de procesador complejos o un número considerable de núcleos de procesador más sencillos.

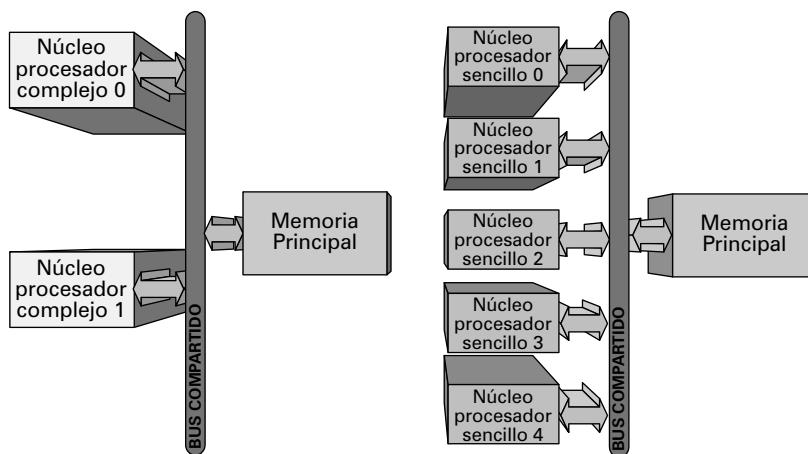


FIGURA 5.4

Arquitecturas on-chip de memoria compartida (I).

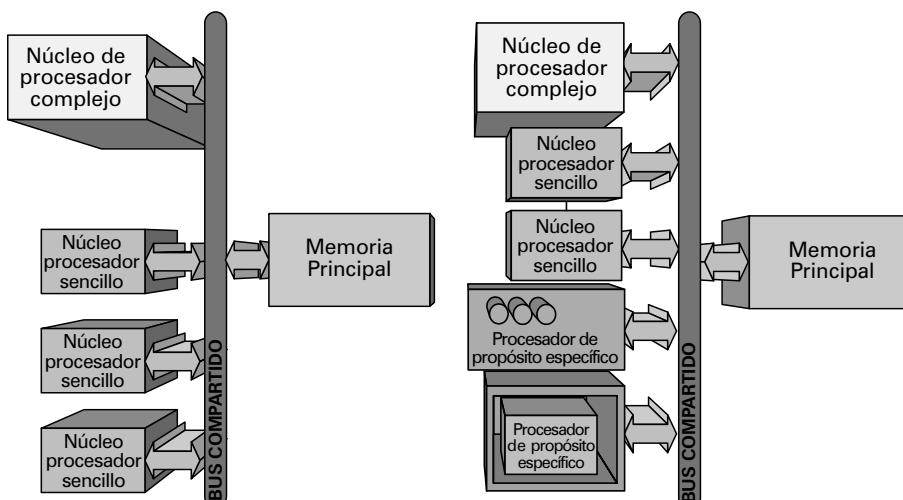


FIGURA 5.5

Arquitecturas on-chip de memoria compartida (II).

En el primer caso, el rendimiento de la arquitectura será alto cuando las aplicaciones que se ejecuten en ella sean poco paralelizables. Sin embargo, en el caso de aplicaciones en las que sea más sencillo explotar el paralelismo, un número de núcleos mayor, aunque sean más sencillos, permitirá aprovechar este paralelismo mucho mejor.

Para evitar tomar esta decisión, se comienzan a proponer diseños en los que se incluye un núcleo más complejo y tantos núcleos sencillos como sea posible. De esta forma, el núcleo complejo será el encargado de ejecutar las aplicaciones poco paralelizables y de repartir el trabajo entre el resto de los núcleos cuando las aplicaciones presenten un grado de paralelismo mayor. Este núcleo actúa por tanto de maestro, centralizando el control de la ejecución de las aplicaciones, y suele ser el que ejecuta el sistema operativo.

De hecho, una vez que se introduce heterogeneidad en los diseños, el siguiente paso es incluir núcleos de procesador de funcionalidad específica que permitan acelerar la ejecución de aplicaciones concretas. Es habitual incluir núcleos SIMD (vectoriales) o de tipo DSP para acelerar las aplicaciones gráficas, de tratamiento de señal, etc.

En los últimos diseños de arquitecturas on-chip de memoria compartida, además de incluir este tipo de núcleos, se ha comenzado a prescindir de la jerarquía de memoria tradicional. En estas propuestas, el núcleo de propósito general más complejo, el que actúa como maestro, sí que suele incorporar uno o varios niveles de memoria caché. Sin embargo, los núcleos de propósito general más sencillos o los de propósito específico, trabajan con los almacenamientos locales descritos en el capítulo 4 (sección 4.3.2). Estos almacenamientos restan complejidad al diseño, evitando toda la gestión asociada a una memoria caché, y normalmente se utilizan para simplificar la arquitectura y reducir el consumo de potencia.

Obviamente, y como ya se ha discutido previamente, la simplificación del hardware conlleva un aumento de responsabilidad para el desarrollador, el compilador y el sistema operativo, que en este caso deben asumir la responsabilidad de llevar desde la memoria compartida hasta los almacenamientos locales mediante técnicas de DMA los datos que cada núcleo de procesador necesita en cada momento.

### 5.1.5. Arquitecturas on-chip de memoria distribuida

Al igual que ocurre en el caso de las arquitecturas tradicionales, la existencia de un bus y de un nivel de la jerarquía de memoria que es compartido, limita la escalabilidad de las arquitecturas de memoria compartida.

Por eso aparecen, también en arquitecturas on-chip, los diseños de memoria distribuida (figura 5.6). Las características son las mismas que en el caso de las arquitecturas tradicionales: comunicación explícita entre procesos a través de paso de mensajes y problema de equilibrio de carga.

La carencia de imagen de sistema único no es un problema en este caso, ya que todos los núcleos de procesador e incluso la red, se encuentran en el mismo chip o empaquetado, por lo que aparecen siempre como un sistema único para el sistema operativo, el compilador y el programador.

En el caso de este tipo de arquitecturas, los núcleos de procesador pueden ser homogéneos o heterogéneos, de propósito general o de propósito específico, pero prácticamente en todos los casos incorporan una jerarquía de memoria tradicional. Además de diseñar estos núcleos, será necesario diseñar una red completa en el chip (Network On Chip, NoC) y un protocolo de paso de mensajes asociado que permita la comunicación entre ellos (figura 5.6).

## 5.2 Redes dentro de arquitecturas de computadoras

En las arquitecturas paralelas compuestas por varios procesadores aparece un nuevo componente (además del procesador, la jerarquía de memoria y el sistema de E/S) cuyas decisiones de diseño afectarán al rendimiento global del sistema: la red de interconexión. Esta red puede ser de área extensa (WAN), de área local (LAN) o de área de sistema (SAN) dependiendo del tipo de arquitectura.

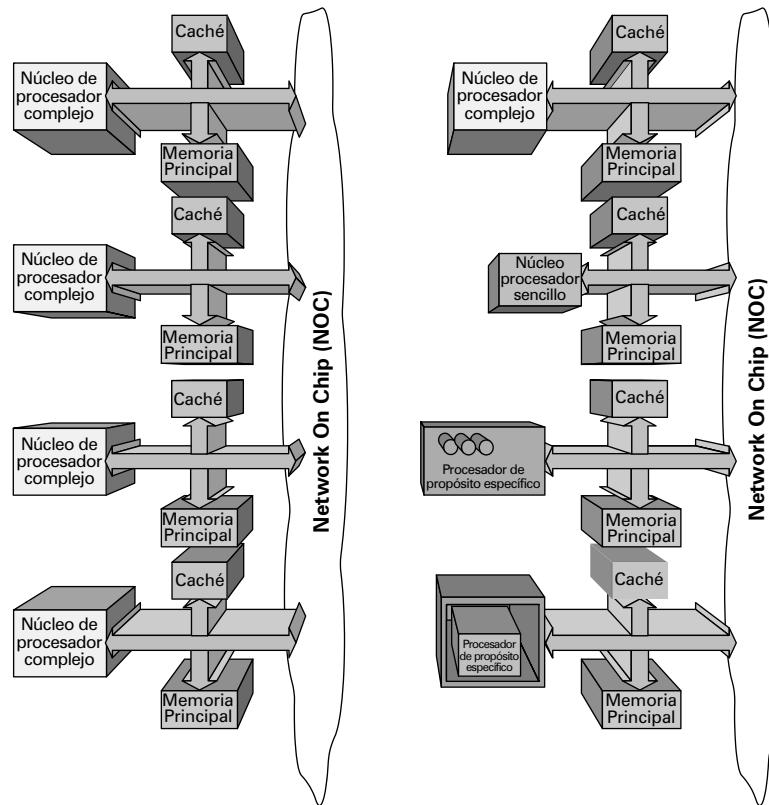


FIGURA 5.6

Arquitecturas on-chip de memoria distribuida.

Aunque los conceptos fundamentales de redes de computadoras escapan del alcance de este libro, sí que es importante analizar brevemente los aspectos más relacionados con las arquitecturas de computadoras que incorporan redes en su interior y con su rendimiento. Es decir, los aspectos relacionados con la topología de las redes y con las técnicas de conmutación, encaminamiento y control de flujo que utilizan.

Y también con las prestaciones de las redes, que de nuevo se pueden cuantificar (por lo menos las prestaciones extremo a extremo) con las métricas que ya se han empleado para la memoria y para el sistema de E/S en capítulos anteriores de este libro: la latencia y el ancho de banda.

En el caso de la red, la latencia se define como el tiempo total de comunicación, es decir, el tiempo que tarda en transferirse un mensaje completo por la red. Casi siempre se utilizan mensajes pequeños para caracterizar esta latencia (unos pocos bytes), pero en la literatura especializada se pueden encontrar multitud de definiciones diferentes de esta latencia que pueden tener en cuenta diferentes aspectos de rendimiento de la red de comunicaciones. Igual que se pueden encontrar definiciones para otras métricas algo más complejas como puedan ser la escalabilidad de la red, su disponibilidad, su calidad de servicio o su eficiencia.

En cuanto al ancho de banda de la red, se define como el número de bytes que es capaz de transmitir la red por unidad de tiempo, al igual que se definía el ancho de banda de un bus.

En la figura 5.7 se observan los niveles de transferencia de información en cualquier arquitectura paralela. La red de interconexión debe proporcionar los mecanismos que permitan trasmitir información entre los nodos que forman la arquitectura. En general, se denomina mensaje a la unidad de información

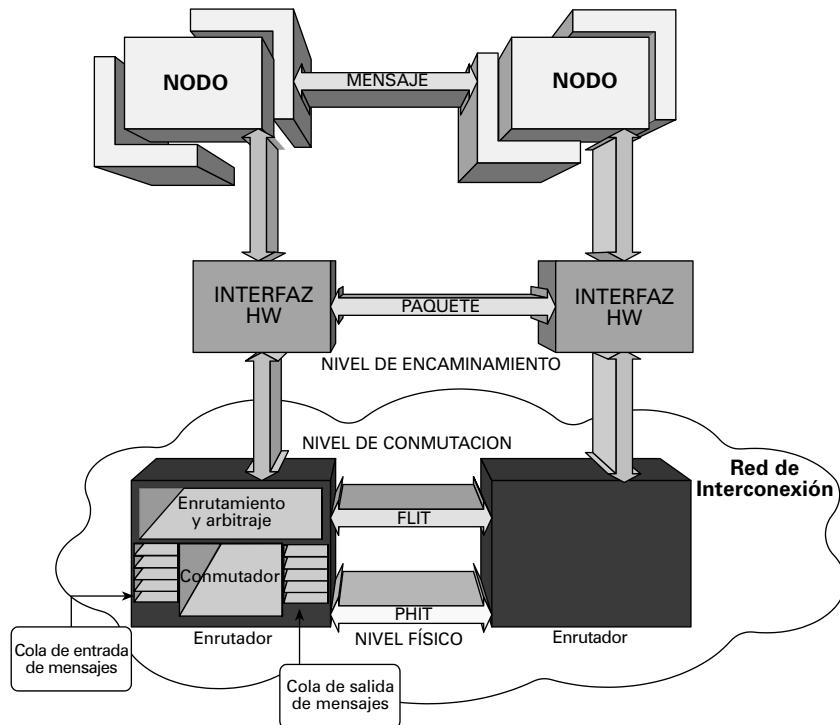


FIGURA 5.7

Niveles de transferencia de información en una red de interconexión.

que intercambian estos nodos. Es necesario, como mínimo, un interfaz hardware de red que permita el acceso de cada nodo a la red de interconexión.

Este interfaz de red puede ubicarse en distintos lugares dentro de la arquitectura, dependiendo del diseño concreto del que se trate en cada caso. Puede estar integrado junto con el procesador, junto con la memoria caché, puede conectarse a la memoria principal de manera similar a como lo hace la tarjeta gráfica o puede conectarse como otro dispositivo periférico a través de algún bus de expansión.

En cualquier caso, la transferencia de un mensaje, en la mayor parte de los casos, requerirá que los interfaces de red generen más de una transferencia de información y se suele denominar paquete a la unidad de transferencia de información entre interfaces. Estos paquetes tienen tamaños muy variables que van desde unos pocos bytes hasta los kilobytes.

En general, el interfaz de red es el encargado de dar formato a estos paquetes, realizar el control de flujo y proporcionar mecanismos de direccionamiento dentro de la red (para que pueda especificarse el nodo origen y el nodo destino de cualquier comunicación). También se responsabiliza del almacenamiento temporal de datos, tanto de entrada como de salida y en algunos casos, de realizar tareas de encaminamiento.

Aunque son los enrutadores (switches) los que permiten que los paquetes de información lleguen desde el origen hasta el destino siguiendo un camino o secuencia de enlaces y enrutadores. Para cada paquete que llegue por alguna de sus entradas, el enrutador decide el canal de salida por el que se debe reenviar para que alcance su destino. Cada entrada y cada salida del enrutador suele tener asignado un espacio de almacenamiento (buffer) que permite guardar temporalmente la información que se está enrutando.

Los enlaces proporcionan conexiones punto a punto para comunicar enrutadores entre sí o enrutadores con interfaces de red. Cuando se utiliza el término canal, suele referirse al conjunto formado por

un enlace, los controladores de este enlace en el origen y en el destino y los recursos de almacenamiento asociados a dicho enlace en ambos extremos.

La unidad de información que se transfiere por un canal entre dos commutadores se denomina Flit (FLow control unIT) y suele estar formada por unos pocos bytes de información (hasta 32 o 64 bytes como mucho). Por último, la unidad de información que se transfiere por un enlace físico en cada ciclo de red (la unidad mínima de información para la red de interconexión, ya no se puede dividir más) es el Phit (PHysical unIT). En este caso se trata de unos pocos bits de información (no más de 128 bits).

### 5.2.1. Topología

En una arquitectura de memoria compartida, un nodo es un procesador con uno o varios niveles de memoria caché. El mensaje que un nodo le envía a otro es la consecuencia de un acceso a memoria y es la propia memoria la que actúa como red de interconexión, por lo que los controladores de memoria integran a los interfaces de red.

La red de interconexión es transparente al programador ya que la comunicación entre dos procesos se realiza con una escritura en memoria (store) seguida de una lectura de memoria (load). Para que la comunicación tenga lugar de manera correcta es necesario solucionar de manera explícita el problema de la sincronización como se estudiará en secciones posteriores, es decir, hay que garantizar que la lectura tiene lugar después de la escritura.

En el extremo contrario están las arquitecturas de memoria distribuida, en las que cada nodo consta de un procesador con uno o varios niveles de memoria caché, memoria principal y mecanismos de E/S, incluida una tarjeta de red que actúa como interfaz de red. En este caso se utilizan funciones explícitas de send y receive para la comunicación entre procesos, y como todo el proceso de comunicación es bastante más complejo que en el caso de las arquitecturas de memoria compartida, suele ser necesaria una capa software en el interfaz de comunicaciones. Con las arquitecturas de memoria compartida-distribuida ocurre lo mismo, y además es necesario un middleware que traduzca las instrucciones load y store a los send y receive necesarios, de manera transparente al usuario.

Como se puede observar, aunque existen aspectos comunes en las redes de interconexión los distintos tipos de arquitectura, existen otros muchos elementos diferenciadores que se materializan, en primer lugar, en el tipo de topología de red que se utiliza en cada caso.

Esta topología determina la estructura física de las interconexiones que componen la red. Normalmente se representa mediante un grafo en el que los vértices son enrutadores o interfaces de red y las aristas son los enlaces entre enrutadores o entre enrutadores e interfaces.

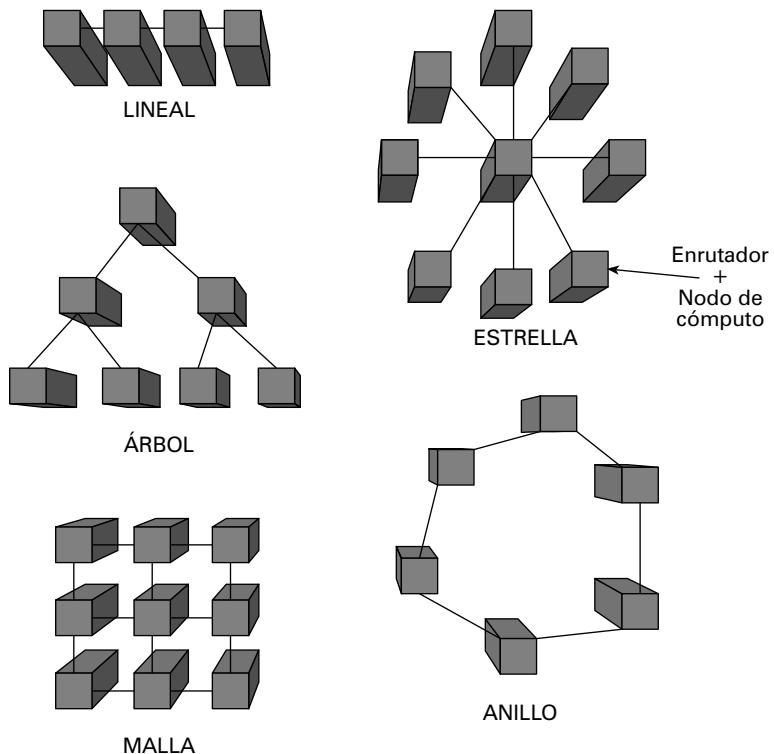
Cuando en este grafo se observa un patrón que se repite, las redes suelen denominarse regulares, mientras que si en este grafo no se puede reconocer ningún patrón repetitivo, las redes suelen denominarse irregulares.

Esta clasificación no suele ser muy relevante hoy en día, sin embargo, existe otra clasificación que es mucho más significativa desde el punto de vista de la arquitectura de computadoras. Las redes pueden clasificarse en estáticas y dinámicas.

#### 5.2.1.1. REDES ESTÁTICAS

También denominadas redes directas. En este caso los enrutadores están fuertemente acoplados a los nodos de cómputo, de manera que la red está formada por conexiones directas punto a punto entre los nodos (figura 5.8). En los grafos que representan este tipo de topologías, los vértices representan a los enrutadores, incluidos dentro de los propios nodos de cómputo. Obviamente en este tipo de red la conexión de un nodo a un enrutador suele ser mucho más rápida que una conexión entre enrutadores, ya que en el primer caso se trata de una conexión local.

Estas redes son costosas y poco escalables, pero se pueden encontrar, en casos muy concretos, tanto en arquitecturas de memoria compartida como en arquitecturas de memoria distribuida.

**FIGURA 5.8**

Ejemplos de redes estáticas.

### 5.2.1.2. REDES DINÁMICAS

Los enruteadores son dispositivos independientes de los nodos de cómputo, por lo que las conexiones ya no son punto a punto. Un enruteador puede estar conectado a uno o varios nodos de cómputo y a uno o varios enruteadores. En este caso, las conexiones nodo-enruteador presentan unas prestaciones muy similares a las conexiones enruteador-enruteador. Cada enruteador puede comunicar según el momento, a dos nodos de cómputo diferentes.

Estas redes se subdividen a su vez en otros dos tipos:

#### 1. Redes de medio compartido o buses

Se trata de redes en las que hay un único enruteador interno basado en un medio compartido por todos los nodos conectados a la red. Sólo un nodo puede utilizar la red en un momento dado para transmitir información, es decir, no se pueden implementar con estas redes múltiples conexiones uno-a-uno simultáneas. Sin embargo es muy fácil realizar difusiones de información a múltiples nodos, ya que al tratarse de un medio compartido, todos los nodos tienen acceso a su contenido en un momento dado.

Un aspecto muy importante en este tipo de redes es la estrategia de arbitraje que determina cómo se resuelven los conflictos de acceso al medio. La solución más sencilla es utilizar técnicas basadas en contención, es decir, en que los nodos compitan para tener el acceso exclusivo al bus. Debido a la compartición del medio todos los nodos pueden monitorizar el estado del bus y detectar colisiones (situaciones

en las que dos o más nodos estén usando el bus al mismo tiempo). Cuando se detecta una colisión, los dispositivos causantes de la misma abortan la transmisión para intentarla posteriormente.

Esta solución es sencilla pero no es determinista, ya que no se puede predecir cuánto debe esperar un nodo para ganar el acceso al bus. Por lo tanto, el bus de contención no es el idóneo para soportar aplicaciones de tiempo real, por ejemplo. Para eliminar el comportamiento no determinista, aparece un enfoque alternativo que implica pasar un testigo (token) entre los nodos, de manera que el dispositivo que tiene en su poder el testigo tiene el acceso al bus. Cuando termina de transmitir sus datos, el testigo se pasa al siguiente dispositivo según algún esquema de prioridades predeterminado. Acotando el tiempo máximo de posesión del testigo, sí que se puede saber el tiempo que un nodo deberá esperar como máximo para conseguir el acceso al bus.

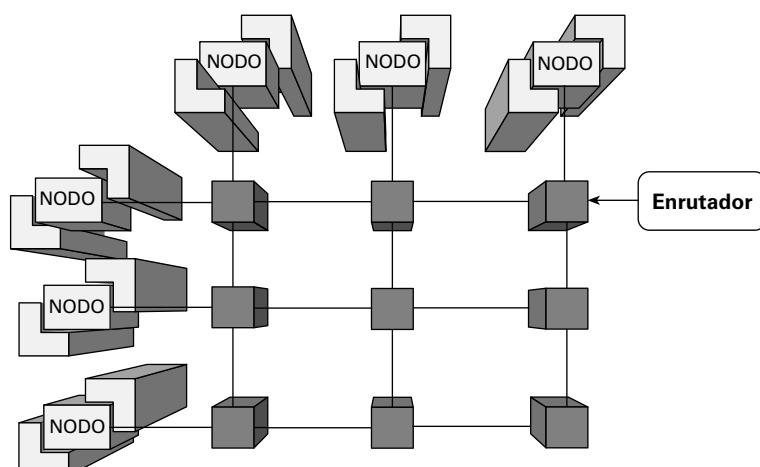
Sea cual sea la solución escogida para el arbitraje, este tipo de redes suelen presentar problemas de escalabilidad. Además suelen limitar el número de nodos que pueden conectarse a ellas y la distancia a la que pueden estar estos dispositivos.

Sin embargo, son redes de bajo coste que se utilizan casi siempre en las arquitecturas de memoria compartida, por lo que se han propuesto algunas soluciones como las redes multibus (existen varios medios compartidos al mismo nivel y todos los nodos se conectan a todos ellos, de manera que pueden utilizar el que esté libre en cada momento) o las redes jerárquicas (también basadas en la utilización de varios medios compartidos, pero no todos al mismo nivel de prioridad).

## 2. Redes conmutadas o indirectas

En este caso no existe un medio compartido, sino que los enrutadores que forman la red permiten que la información llegue del origen al destino. Son las redes típicas en arquitecturas de memoria distribuida y de nuevo existe una clasificación que distingue entre dos tipos de redes:

- **Redes Monoetapa.** Se pueden conectar dos nodos cualesquiera con un único salto. Suelen denominarse redes crossbar o de barras cruzadas, y son muy eficientes pero también muy caras. Permiten que cualquier nodo se comunique con otro en una única etapa y que existan conexiones simultáneas uno-a-uno siempre y cuando no sean con el mismo nodo destino (figura 5.9). Es decir, un nodo origen puede conectarse siempre con un nodo destino libre ya que cada pareja origen/destino tiene su propio camino independiente en la red.



**FIGURA 5.9**

Red conmutada monoetapa: barras cruzadas.

- **Redes Multietapa (MIN o Multistage Interconnection Network).** Si no se puede garantizar la anterior condición, para llegar de un nodo a otro puede que sea necesario más de un salto entre enrutadores. En este caso hay dos posibilidades:

— **Redes Bloqueantes.** No siempre es posible establecer una conexión entre un origen y un destino que estén libres ya que se comparten conexiones (figura 5.10). Normalmente hay un único camino posible para llegar de un nodo origen a otro destino (de esta forma se minimiza el número de enrutadores y de etapas en la red), por lo que si alguno de los enlaces que hay que atravesar está ocupado, no será posible establecer la conexión. Las redes más conocidas de este tipo son las omega y las de mariposa.

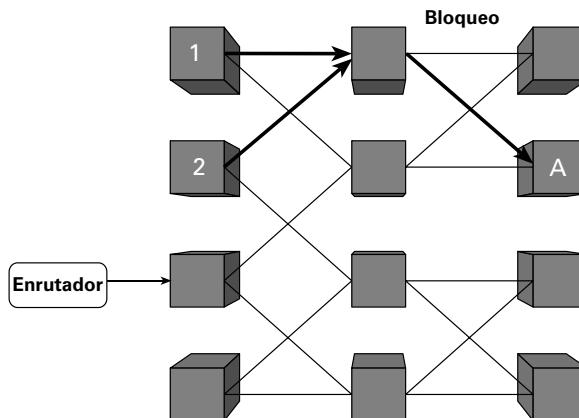


FIGURA 5.10

Red conmutada multietapa bloqueante.

— **Redes No Bloqueantes.** Cualquier origen puede conectarse con cualquier destino libre sin que esto afecte a otras conexiones que puedan establecerse a continuación (figura 5.11). Es decir, se comporta de manera similar a una red crossbar (permite cualquier combinación origen/destino) pero sin garantizar que la conexión pueda realizarse sólo con un salto, por lo tanto, con una latencia mayor. Las redes de este tipo más conocidas son las de Clos.

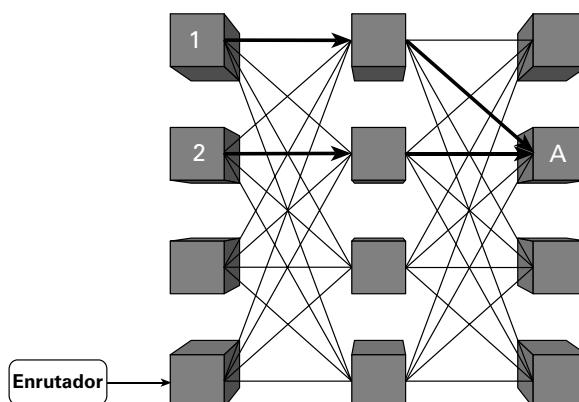


FIGURA 5.11

Red conmutada multietapa no bloqueante.

## 5.2.2. Técnicas de conmutación

Estas técnicas determinan cómo se transmiten los datos por la red desde el nodo origen hasta el nodo destino, y tienen una gran influencia en las prestaciones de la red. De hecho, puede decirse que en la actualidad, es el aspecto de los analizados en esta sección 5.2 que más repercute en el rendimiento de la red, ya que estas técnicas determinan el tamaño de la unidad de información que se asigna a los buffers de los enrutadores así como el tamaño de la unidad que se transfiere entre interfaces de red.

### 5.2.2.1. CONMUTACIÓN DE CIRCUITOS

Con esta técnica se reserva el camino completo desde el origen hasta el destino antes de comenzar la transferencia de información. Para ello, el interfaz origen envía una sonda (con tamaño de flit) con la información de encaminamiento que necesitan conocer los enrutadores.

Cuando la sonda llega al destino queda establecido un camino formado por un circuito de comunicación continuo. El destino envía una señal de reconocimiento a la fuente si tiene suficiente espacio para almacenar los datos que se le van a enviar. En cuanto el origen recibe la señal de reconocimiento empieza la transferencia.

Es muy importante comprender que con esta técnica todos los enlaces que componen el camino están reservados durante la transferencia completa, y que dado el coste que supone hacer esta reserva, siempre se transfiere el mensaje completo. Existen dos mecanismos para liberar el circuito reservado:

- La cola del mensaje va liberando los enlaces que lo constitúan.
- Lo libera el destino enviando una señal de reconocimiento cuando finaliza la transferencia.

Esta técnica de conmutación es muy fiable, su latencia es prácticamente constante y además no es necesario almacenar información en los enrutadores que componen el camino.

Pero pueden aparecer situaciones de interbloqueo debido a la necesidad de reservar los circuitos antes de comenzar con las transferencias de información (figura 5.12). Por ello se utiliza bastante poco en diseños reales.

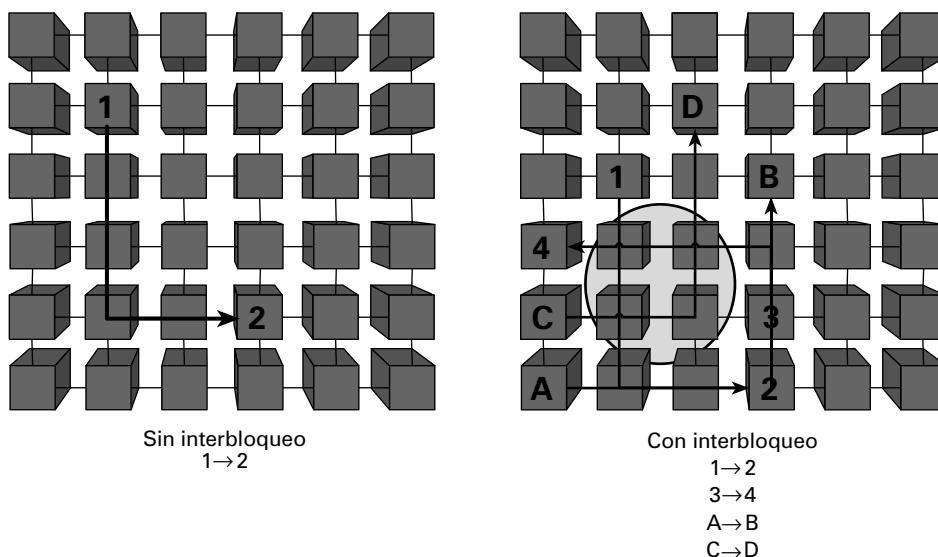


FIGURA 5.12

Conmutación de circuitos. Ejemplos sin y con interbloqueo.

### 5.2.2.2. CONMUTACIÓN DE PAQUETES

Esta técnica también se denomina de almacenamiento y reenvío, porque en ella cada enrutador recibe un paquete de información completo y lo almacena, antes de ejecutar el algoritmo de encaminamiento que le indicará cuál debe ser el siguiente salto del paquete por la red (figura 5.13).

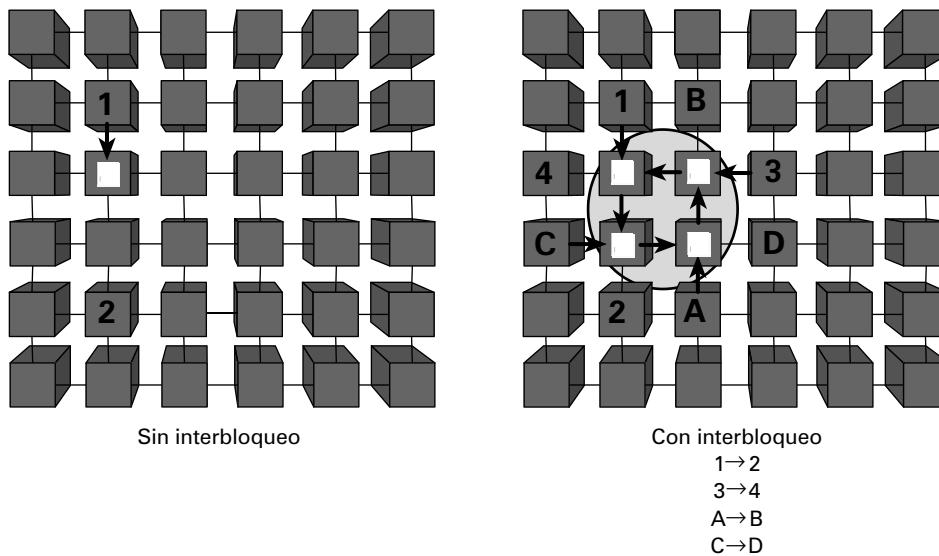


FIGURA 5.13

Comutación de paquetes. Ejemplos sin y con interbloqueo.

Por lo tanto, el mensaje se divide en paquetes de un tamaño máximo establecido en el interfaz de red del nodo origen. En cada enrutador, una vez ejecutado el algoritmo de encaminamiento, todo el paquete se reenvía al siguiente enrutador. Así que se asigna buffer y canal a un paquete y en un momento dado un paquete sólo ocupa un canal, no reserva ningún enlace.

Como cada paquete se transfiere de manera independiente desde el origen hasta el destino, el interfaz destino debe ser capaz de reconstruir la información original reensamblando todos los paquetes que le llegan, probablemente, desordenados.

Con esta técnica se evitan muchas situaciones de interbloqueo, pero pueden seguir produciéndose por falta de espacio en los enrutadores para almacenar los paquetes (figura 5.13). En general, el tamaño de los paquetes debe ser el menor posible para que no se exija una gran capacidad de almacenamiento a los enrutadores, sin influir negativamente en la latencia de comunicaciones. Esta técnica de conmutación se utiliza normalmente en redes LAN y SAN para arquitecturas de tipo cluster (memoria distribuida).

### 5.2.2.3. CONMUTACIÓN VIRTUAL CUT-THROUGH

Se trata de una solución intermedia entre las dos técnicas clásicas explicadas con anterioridad, conmutación de circuitos y de paquetes.

De hecho, funciona como la conmutación de paquetes pero permite que los paquetes avancen mientras los enlaces que necesitan atravesar estén libres, eliminando almacenamientos intermedios de los paquetes cuando son innecesarios. En este sentido se puede decir que funciona como la conmutación de circuitos, pero sin la necesidad de reservar el camino completo, se pueden realizar avances por caminos parciales. Cuando un paquete no puede avanzar más, se debe almacenar completo en un enrutador (figura 5.14).

Es decir, un paquete no bloqueado puede estar ocupando con sus transferencias varios canales de la red que formen su camino parcial en ese momento, por lo que también se trata de una técnica de camino

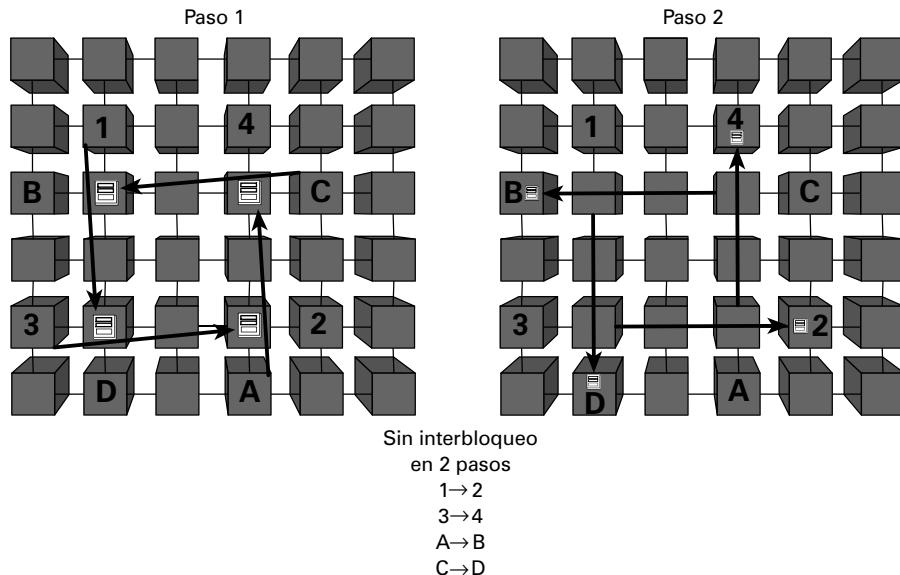


FIGURA 5.14

Comutación virtual cut-through.

segmentado. La latencia con esta técnica vuelve a ser prácticamente constante y en la actualidad se puede encontrar en el BlueGene de IBM, por ejemplo.

#### 5.2.2.4. CONMUTACIÓN VERMIFORME

Se trata de una técnica muy parecida a la de virtual cut-through pero exige menor capacidad de almacenamiento en los enruteadores. El paquete se divide en flits, de manera que en cuanto el primero de un paquete llega al enruteador, se puede comenzar a ejecutar el algoritmo de encaminamiento.

Terminada la ejecución de este algoritmo, se permite que avance el flit cabecera del paquete aunque los siguientes no hayan llegado todavía al enruteador. Estos flits, que irán llegando en orden, seguirán a la cabecera, que irá abriendo camino imitando el movimiento de un gusano (figura 5.15).

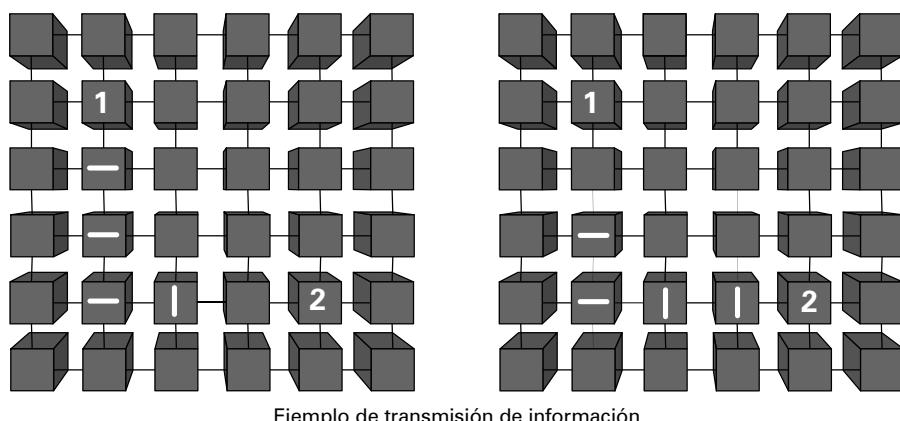


FIGURA 5.15

Comutación vermiforme.

Por eso con esta técnica se reduce la necesidad de almacenamiento en los enrutadores, que sólo necesitan capacidad para un flit. De hecho, se asignan tanto el buffer como el enlace a un flit. Es la técnica de commutación que utiliza Myrinet, una red muy extendida en la actualidad para clusters (memoria distribuida).

### 5.2.3. Técnicas de encaminamiento

Tanto si se utiliza commutación de paquetes como si se utilizan técnicas más recientes como virtual cut-through o vermiforme, es necesario determinar el camino que los paquetes van a seguir para llegar del origen al destino (en el caso de la commutación de circuitos el encaminamiento también se realiza, pero es tremadamente sencillo ya que se fija en el nodo origen cuando se realiza la reserva del circuito completo para la transferencia del mensaje).

Un algoritmo de encaminamiento debe determinar este camino asegurando la conectividad total (que cualquier nodo puede enviar un paquete a cualquier otro) y evitando en todo lo posible las situaciones de interbloqueo, en las que dos o más paquetes no pueden continuar su avance porque se bloquean esperando a que el otro o los otros liberen un recurso. Además, hay que intentar evitar que un paquete vague por la red ocupando recursos sin llegar nunca a su destino, hay que proporcionar tolerancia a fallos y procurar que exista una distribución equilibrada del tráfico en toda la red.

Para diseñar algoritmos de encaminamiento que permitan conseguir estos objetivos hay que tomar tres decisiones fundamentales:

1. Cómo se va a realizar la selección del camino entre todas las alternativas posibles. Existen dos maneras de realizar esta selección, de manera inconsciente o de manera adaptativa. En el primer caso se toma la decisión sin tener en cuenta para nada el estado de la red y de los componentes que la forman. Por lo tanto se puede utilizar un algoritmo aleatorio, rotatorio o determinístico (que siempre escoja el mismo camino para ir de un origen a un destino determinado) que escoja entre todas las opciones posibles. En el caso de un algoritmo adaptativo, por el contrario, para tomar esta decisión se tendrá en cuenta la información más actualizada que se tenga acerca del estado de los recursos que forman la red (grado de ocupación, alertas y errores, etc.). Además los algoritmos de encaminamiento pueden clasificarse en mínimos y no mínimos. En el primer caso, sólo se pueden realizar encaminamientos que acerquen los paquetes a su destino, mientras que en el segundo caso es posible alejarse del destino en alguno de los saltos.
2. Qué enrutador va a tomar esta decisión. En este caso existen tres alternativas diferentes. En primer lugar, puede ser el propio nodo origen el que incorpore en la cabecera del paquete información acerca del encaminamiento. Puede incorporar sólo alternativas y dejar que los enrutadores realicen la selección entre ellas (por ejemplo, si se utiliza un algoritmo adaptativo, ya que serán los enrutadores los que tengan la información más actualizada acerca del estado de la red). Pero también pueden incorporar las decisiones definitivas acerca del encaminamiento en el caso de algoritmos inconscientes. La segunda alternativa implica una decisión distribuida entre todos los enrutadores. En este caso la cabecera del paquete sólo incorpora la dirección destino, que será la que permita a los enrutadores tomar sus decisiones. Por lo tanto, el paquete tiene un tamaño menor que con la primera alternativa, pero por otro lado, se invierte más tiempo en tomar las decisiones de encaminamiento. Por último, existe la alternativa del encaminador centralizado, una unidad hardware encargada exclusivamente de tomar las decisiones de encaminamiento, por ejemplo, en las redes crossbar.
3. De qué manera se va a implementar el algoritmo. Hasta el momento, dos soluciones han dado buen resultado, las tablas de encaminamiento y las máquinas de estados finitos. Con cualquiera

de las dos técnicas se trata de averiguar las posibles alternativas de encaminamiento en función de la dirección del nodo destino y de tomar una decisión en base a los criterios de encaminamiento para escoger una de estas alternativas.

### 5.2.3. Técnicas de control de flujo

Estas técnicas garantizan que las unidades de información (mensajes, paquetes o flits) llegan al destino sin que exista solapamiento con unidades previas o posteriores. Además, aseguran que se produce una recepción sin errores, garantizan una calidad de servicio mínima en la red y gestionan los posibles conflictos por recursos (especialmente, por los recursos de almacenamiento en los enrutadores y por los enlaces).

Las técnicas de control de flujo se deben aplicar a diferentes niveles para cumplir con todas estas funcionalidades. Normalmente, se distingue entre control de flujo extremo a extremo, control de flujo de conmutación y control de flujo físico. Cada uno de estos niveles confía en los que están por debajo para realizar sus funciones.

El control de flujo físico garantiza que las unidades físicas de información o phits se reciben en el destino correctamente.

Para ello pueden utilizarse técnicas síncronas o asíncronas. En el caso de utilizar control de flujo síncrono, es una señal de reloj la que determina cuándo el nodo origen deposita nuevos phits en el enlace y cuándo el nodo destino muestrea el canal físico para recibirlos. Si las líneas que componen el enlace son muy largas y existe el peligro de que el origen y el destino se des-sincronicen, se recurre a las soluciones tradicionales; se envía la señal de reloj del origen al destino por una línea adicional o se codifica esta señal de reloj junto con los datos.

En el caso de utilizar un control de flujo asíncrono, se emplean señales de sincronización para que el origen indique al destino que hay un phit en el enlace y para que el destino confirme al origen que lo ha recibido correctamente y que está esperando la transmisión de un nuevo phit.

El control de flujo de conmutación garantiza que la unidad de información utilizada en la conmutación se transfiere correctamente entre enrutadores. Estas unidades son el paquete para técnicas de conmutación de paquetes o virtual cut-through, y el flit para la conmutación de circuitos (sonda de reserva del circuito) o vermiforme.

Las técnicas de control de flujo en este nivel garantizan que en el siguiente enrutador haya espacio de almacenamiento suficiente para recibir la unidad de información.

Si el control de flujo físico es asíncrono, la propia señalización de control envío/recepción entre origen y destino, permite realizar también el control de flujo de conmutación.

Por el contrario, si el control de flujo físico es síncrono, se deben utilizar técnicas de otro tipo para el control de flujo de conmutación. Las más conocidas son el control de flujo basado en créditos y el control de flujo STOP&GO.

En el caso de los créditos, se añade una señal de control que se envía desde el enrutador destino hacia el enrutador origen y que informa acerca de la disponibilidad de espacio de almacenamiento en el enrutador destino. Esta señal es la que concede crédito para el envío de la unidad de información.

Si se utilizan técnicas de STOP&GO, el control de flujo se basa en dos tipos de señales, la de GO, que indica al enrutador origen que pueden enviar unidades de información hacia el destino, y la de STOP, que indica que debe cesar el envío de información porque no queda espacio de almacenamiento suficiente en el enrutador destino. En el buffer de cada enrutador hay dos niveles de ocupación diferentes, el que hace que se active la señal de STOP (cuando sólo queda espacio para almacenar las unidades que en ese momento se están transmitiendo por el enlace) y el que hace que se active la señal de GO (cuando se calcula que en el tiempo que la señal de GO llega al enrutador origen se habrá liberado espacio suficiente en el buffer). Estas técnicas son especialmente interesantes en enlaces con ancho de

banda limitado, ya que ahorran la mayor parte de la señalización de control que supondría una gestión por créditos.

Por último, el control de flujo extremo a extremo garantiza que el mensaje o paquete se transmite correctamente desde la interfaz del nodo origen hasta la interfaz del nodo destino. En este caso se pueden utilizar también técnicas basadas en créditos o en STOP&GO, aunque existen multitud de propuestas específicas para diferentes redes y arquitecturas.

## 5.3 Diseño de arquitecturas de memoria compartida

En las arquitecturas de memoria compartida varios procesadores comparten una única memoria principal con la que se comunican por un único bus y a través de la cual se realiza la comunicación entre los procesadores utilizando instrucciones de carga y almacenamiento.

Para que la memoria compartida no se convierta en el cuello de botella del sistema son necesarias memorias caché de gran eficiencia, es decir, con una tasa de fallos lo menor posible. Así los procesadores recurrirán lo menos posible a la memoria principal compartida. En las arquitecturas de memoria compartida on-chip, en las que puede que se esté compartiendo un nivel de la memoria caché en lugar de la memoria principal, este principio se cumple igualmente, cuanto menos falle la caché de nivel 1 de los procesadores, menos recurrirán al nivel de la memoria caché que se comparte.

Las técnicas de diseño de procesadores estudiadas en los capítulos 1 y 3 y las de diseño jerarquía de memorias estudiadas en los capítulos 2 y 4 se utilizan para el diseño de este tipo de arquitecturas. Pero es necesario añadir nuevas técnicas que permitan resolver los problemas ya mencionados: coherencia, consistencia y sincronización.

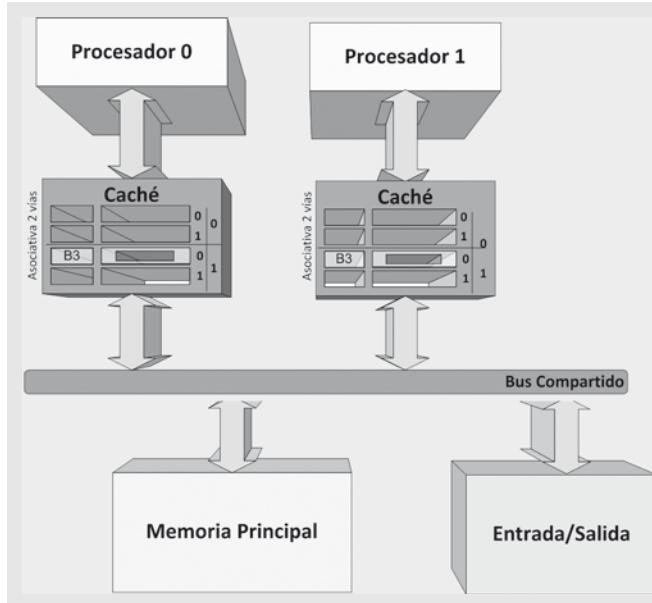
Primero se estudiará qué implican estos problemas para luego pasar a discutir cuáles de ellos se podrán resolver desde la propia arquitectura y cuáles se resolverán a un nivel superior. Este estudio se centrará en arquitecturas de memoria compartida tradicionales (homogéneas y compartiendo la memoria principal), pero la mayor parte de las soluciones presentadas pueden aplicarse directamente o adaptarse con facilidad a arquitecturas on-chip (aunque sean heterogéneas y compartan un nivel de la memoria caché).

El problema de la coherencia aparece en el momento en que existen diferentes copias del mismo dato en diferentes memorias del sistema. Por ejemplo, en el caso de un bloque de información que esté siendo utilizado por varios procesadores del sistema, se podrá encontrar copia de este mismo bloque en diferentes memorias caché de la arquitectura y en la memoria principal compartida. En el momento en el que estas copias sean diferentes entre sí, aparece una falta de coherencia que puede llevar a resultados equivocados en la ejecución de códigos, ya que hay que recordar que en este tipo de arquitecturas los procesadores se comunican de manera implícita a través de lecturas y escrituras en memoria. Esta comunicación no se realizará correctamente si no se garantiza que cuando un procesador lee una variable esta lectura le está devolviendo el último valor que se ha escrito en ella.

### Ejemplo 5.1

**Ejemplo del problema de coherencia en una arquitectura de memoria compartida.**

Supongamos que tenemos una arquitectura de memoria compartida con 2 procesadores, P0 y P1. Cada uno de estos procesadores tiene un único nivel de caché con escritura directa.



En este ejemplo tenemos a P0 y P1 ejecutando dos procesos que comparten una variable A ubicada en el bloque 3 de memoria.

Si los dos procesos realizan sólo lecturas de esta variable compartida, no surge ningún problema.

Pero, ¿qué ocurre si P1 escribe un nuevo valor para la variable A en su copia de B3? Al ser las cachés de escritura directa, este nuevo valor se actualizará tanto en la caché de P1 como en la memoria principal.

Pero la siguiente lectura que haga P0 del valor de A, lo hará de su copia de B3 en su caché, que tendrá almacenado el valor antiguo de A.

Este es el típico problema de coherencia que surge cuando se pueden almacenar en las memorias caché las variables compartidas y que debe resolverse de alguna manera para que los resultados de las aplicaciones sean los correctos.

En el caso de la consistencia, este problema aparece porque la existencia de varios procesadores en el sistema obliga a definir el orden en el que se materializan las operaciones en memoria principal.

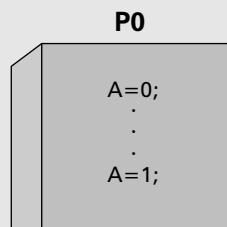
En sistemas monoprocesador este orden está claro, ya que siempre es el orden del programa, es decir, debe parecer que las instrucciones que acceden a memoria se han ejecutado en el orden secuencial en el que aparecen en el código. En realidad no tienen por qué ejecutarse en este orden (existen técnicas de ejecución fuera de orden como se estudió en el capítulo 3), pero sí tiene que parecer, por los resultados obtenidos, que se han ejecutado en este orden.

En el caso de una arquitectura con varios procesadores, hay que definir este orden. Es decir, hay que decidir cómo se intercalan las instrucciones de los códigos que se ejecutan en cada uno de los procesadores.

### Ejemplo 5.2

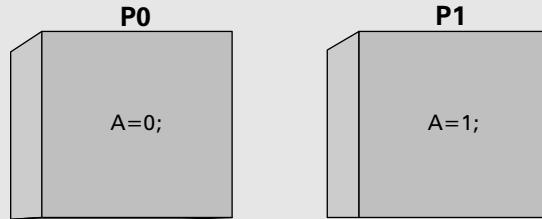
Ejemplo del problema de consistencia en una arquitectura de memoria compartida.

Supongamos que en el procesador P0 del ejemplo 5.1 aparece la siguiente secuencia de asignación de valores a la variable compartida A:



Sabemos que según el modelo de programación de las arquitecturas monoprocesador, se debe respetar el orden secuencial de estas asignaciones, de manera que el valor de la variable A que prevalece es A = 1.

Sin embargo, en la arquitectura con dos procesadores, si nos encontramos con esta secuencia de asignaciones para la variable compartida:



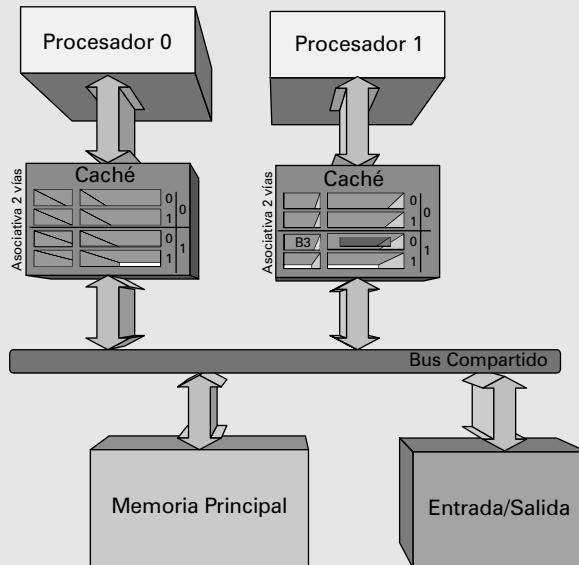
¿Cuál es el valor de la variable compartida que debe prevalecer? Esto es lo que tiene que definir el modelo de consistencia. Dependiendo del que se implemente en esta arquitectura la variable mantendrá el valor A = 0 o el valor A = 1.

Por último, el problema de la sincronización aparece, ya que la comunicación entre procesos se realiza a través de una memoria compartida. Para que esta comunicación sea correcta, es necesario sincronizar los accesos a las variables compartidas, tanto en comunicación uno a uno como en comunicaciones colectivas.

### Ejemplo 5.3

#### Ejemplo del problema de sincronización en una arquitectura de memoria compartida.

Supongamos en este caso que sólo el procesador P1 tiene copia de la variable compartida A en su memoria caché porque está acumulando en ella el resultado de una operación matemática. Cuando termine esta operación, el proceso que se está ejecutando en P0 tiene que leer su valor, modificarlo e imprimirlo en pantalla para que el usuario visualice el resultado obtenido.



Aunque se haya definido un modelo de consistencia, esto no resuelve el problema planteado, ya que en este caso necesitamos algún mecanismo que asegure al proceso que se ejecuta en P0 cuando realice la lectura de la variable compartida A que el procesador P1 ya ha terminado de trabajar con esta variable y que se está leyendo de memoria principal el resultado final.

### 5.3.1. Soluciones para el problema de la coherencia

Este problema se puede resolver exclusivamente desde la propia arquitectura, añadiendo controladores de coherencia a las memorias cachés que están en el último nivel de la jerarquía de memoria que no está compartido. Es decir, si se comparte la memoria principal, se añaden estos controladores a las cachés de nivel 2 de todos los procesadores. Si se comparte el nivel 2 de caché, se añaden a las cachés de nivel 1.

Estos controladores de coherencia se basan en una idea muy sencilla: pueden espiar en todo momento lo que están haciendo el resto de procesadores del sistema ya que existe un medio compartido de comunicaciones, un bus. Por eso se basan en protocolos de coherencia que se denominan de espionaje o Snoopy (figura 5.26).

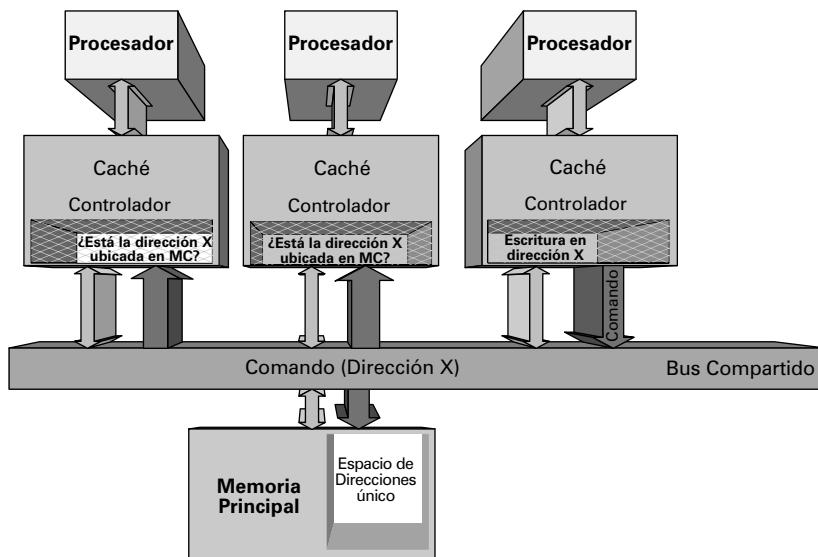


FIGURA 5.16

Protocolo de coherencia basado en espionaje con un nivel de caché.

Para diseñar un protocolo de este tipo es necesario especificar:

- Los posibles estados de un bloque alojado en un determinado marco de la caché de un procesador.
- Los comandos del protocolo, es decir, las órdenes o peticiones que los controladores envían por el bus para que sean vistas por otros controladores.
- El diagrama de transición de estados de los bloques en caché, con los eventos que pueden provocar estas transiciones y las acciones que hay que llevar a cabo ante determinados eventos, transiciones o comandos.

Los estados y comandos pueden variar mucho de unos protocolos a otros, por lo que es difícil proponer una notación general. Sin embargo, para la definición de eventos, en este libro se va a diferenciar siempre entre dos tipos, los locales y los globales.

Los eventos locales son aquellos que se producen por acciones que lleva a cabo el procesador asociado a la propia caché y a su controlador. Los eventos globales, por el contrario, se producen cuando

se observan en el bus compartido comandos enviados por los controladores de coherencia de cachés remotas.

Los eventos locales pueden ser, por tanto, de cuatro tipos:

- **PRHit.** Petición local de lectura con acierto en caché.
- **PRMiss.** Petición local de lectura con fallo en caché.
- **PWHit.** Petición local de escritura con acierto en caché.
- **PWMiss.** Petición local de escritura con fallo en caché.

En el caso de los eventos globales, siempre se utilizará la notación Bus para identificarlos, seguida del nombre del comando que se observa. Habrá siempre un tipo de evento global por cada comando del protocolo. Estos eventos globales tendrán lugar cuando un controlador de coherencia detecte que un determinado comando que otro controlador ha volcado en el bus se refiere a una dirección de memoria que está ubicada en ese momento en la memoria caché local.

Por ejemplo, **BusRMiss(X)** sería el evento en el que se observa en el bus que otro procesador ha volcado un comando de fallo de lectura por la dirección de memoria X, que está ubicada en la memoria caché local.

Además, hay que tener en cuenta que existen dos alternativas para diseñar estos protocolos, la invalidación y la actualización.

En el caso de los protocolos de invalidación, se basan en asegurar que un procesador tiene acceso exclusivo a un bloque de datos antes de escribir en él. Para ello, cuando un procesador desea escribir sobre un bloque, antes envía un comando de invalidación por el bus que invalida el resto de las copias en otras cachés del sistema. A partir de ese momento no quedan más copias válidas de ese bloque en otras cachés y los futuros accesos sobre el bloque modificado serán en exclusiva. Un bloque inválido es equivalente a un bloque no presente en caché, cuando se solicita produce un fallo.

Al producirse este fallo pueden ocurrir dos cosas. Si las cachés son de post-escritura, la única copia correcta del dato está en la caché que lo tiene en exclusiva. En ese caso, el controlador de caché monitorea el bus para detectar solicitudes del bloque modificado y enviarlo actualizado a los procesadores que lo necesiten o a la memoria principal para que sea ésta la que se encargue de resolver el fallo con el dato ya actualizado (lo que ocurre normalmente).

Si las cachés son de escritura directa habrá una copia correcta de la información en la memoria principal tras la escritura, así que esta monitorización no es necesaria y es la memoria principal la que resuelve el fallo como ocurre normalmente.

La actualización se basa en que todos los datos están siempre actualizados en las diferentes cachés. Para ello, cuando se escribe en un bloque en un procesador determinado se difunden por el bus las modificaciones realizadas a todas las cachés que tienen una copia de este bloque, y éstas lo actualizan.

La actualización es necesaria sólo en caso de datos compartidos, es decir que tengan múltiples copias en diferentes cachés. Por ello se hace una clara distinción entre datos compartidos y locales. Esto permite evitar actualizaciones innecesarias y reducir las necesidades de ancho de banda para la actualización.

Si las cachés son de escritura directa, cada actualización de una escritura servirá también para modificar el contenido de la memoria principal. Si por el contrario, las cachés son de post-escritura, las actualizaciones sólo se realizarán en otras cachés y la memoria principal sólo se actualizará en los reemplazamientos, como es habitual.

Veamos la diferencia entre invalidación y actualización con dos ejemplos sencillos. Para ello vamos a diseñar dos protocolos de coherencia de caché, uno de invalidación y otro de actualización, ambos de tres estados, ya que son los más sencillos que pueden proponerse. En ambos casos vamos a suponer que las cachés son de post-escritura.

**Ejemplo 5.4**

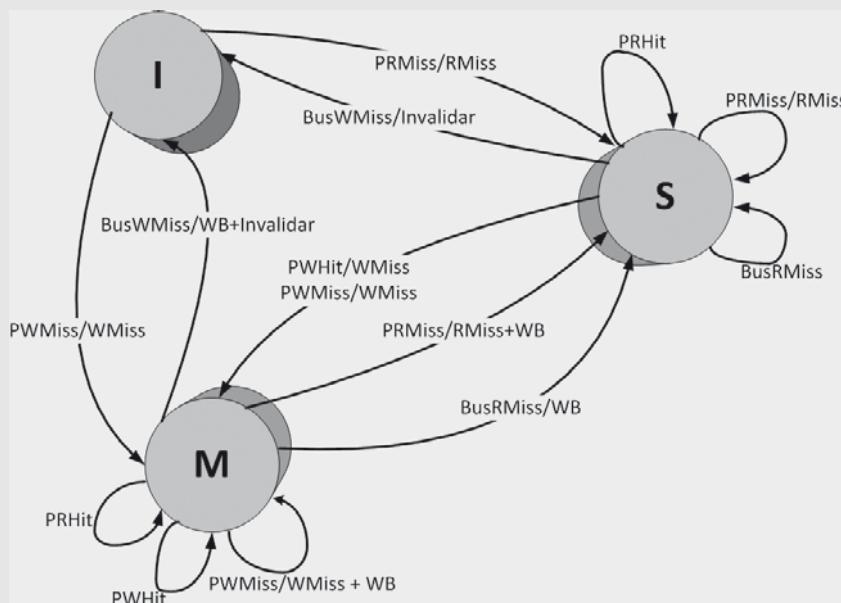
Protocolo de coherencia de caché basado en espionaje, de 3 estados y con invalidación.  
Cachés de post-escritura.

En este ejemplo tenemos un protocolo que define tres estados posibles para los bloques ubicados en caché:

- Inválido (I). El bloque no está en la caché y si lo ha estado en algún momento, ha sido invalidado, por lo que cuando el procesador lo solicita para leer o para escribir se produce un fallo.
- Compartido o Shared (S). El bloque está en la caché y es coherente con la memoria principal. Puede que existan otras copias en estas mismas condiciones en las cachés de otros procesadores.
- Modificado (M). El bloque está en la caché y no es coherente con la memoria principal. Esto significa que se ha realizado al menos una escritura sobre él y que por lo tanto, es la única copia en las cachés del sistema (la caché local tiene el bloque en exclusiva).

Este protocolo MSI utiliza dos comandos para mantener la coherencia en el sistema, RMiss (fallo de lectura) y WMiss (fallo de escritura + invalidación implícita). Por último, se contemplan dos acciones, la primera denominada WB implica el volcado de un bloque completo al bus compartido para actualizar su contenido en la memoria principal, la segunda es una acción Invalidar que implica marcar como inválido un bloque de la caché local.

El diagrama de transición de estados para este protocolo es el siguiente:



Analicemos las transiciones que se producen desde cada uno de los estados:

| ESTADO ORIGEN DE LA TRANSICIÓN | ESTADO DESTINO DE LA TRANSICIÓN | EVENTO QUE LA PROVOCÀ | EXPLICACIÓN                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------------------------|---------------------------------|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| I                              | S                               | PRMiss                | El procesador local solicita leer un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal con un comando RMiss que resuelva este fallo de lectura. El resto de procesadores sufren un evento global BusRMiss si ven este comando en el bus compartido y les afecta.<br>La copia del bloque que se recibe puede ser la única del sistema o puede estar compartida. |

| ESTADO ORIGEN DE LA TRANSICIÓN | ESTADO DESTINO DE LA TRANSICIÓN | EVENTO QUE LA PROVOCÁ | EXPLICACIÓN                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------------------------|---------------------------------|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                | M                               | PWMiss                | <p>El procesador local solicita escribir un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal con un comando WMiss que resuelva este fallo de escritura.</p> <p>La copia que se recibe del bloque es la única ubicada en caché de toda la arquitectura.</p>                                                                                                                                                                                                                       |
| S                              | S                               | PRHit                 | <p>El procesador local solicita leer un bloque que ya está ubicado en la caché, por lo que se produce un acierto y el estado de este bloque no se modifica.</p>                                                                                                                                                                                                                                                                                                                                                                  |
|                                | S                               | PRMiss                | <p>El procesador local solicita leer un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal con un comando RMiss que resuelva este fallo de lectura.</p> <p>La copia del bloque que se recibe puede ser la única del sistema o puede estar compartida.</p> <p>Esta copia reemplaza a la de un bloque que también estaba en estado compartido (S), por lo que la etiqueta asociada al marco escogido para el emplazamiento no se modifica.</p>                                       |
|                                | M                               | PWHit                 | <p>El procesador local solicita escribir un bloque que ya está ubicado en la caché, por lo que se produce un acierto.</p> <p>Pero se vuela un comando WMiss en el bus compartido que sirve como orden de invalidación de este bloque para sus posibles copias en otras cachés.</p> <p>La copia del bloque en caché es exclusiva.</p>                                                                                                                                                                                             |
|                                | M                               | PWMiss                | <p>El procesador local solicita escribir un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal con un comando WMiss que resuelva este fallo de escritura. Este comando sirve al mismo tiempo como orden de invalidación de este bloque para sus posibles copias en otras cachés.</p> <p>La copia que se recibe del bloque es exclusiva, no hay más copias en las cachés del sistema.</p>                                                                                           |
|                                | S                               | BusRMiss              | <p>Se observa en el bus compartido un fallo de lectura de otro procesador por un bloque que está ubicado en la caché local. Su estado no se modifica.</p>                                                                                                                                                                                                                                                                                                                                                                        |
|                                | I                               | BusWMiss              | <p>Se observa en el bus compartido un fallo de escritura de otro procesador por un bloque que está ubicado en la caché local. Este comando lleva implícita una orden de invalidación ya que en los protocolos de invalidación las escrituras se hacen en copias exclusivas. Se invalida la copia local del bloque.</p>                                                                                                                                                                                                           |
| M                              | M                               | PRHit                 | <p>El procesador local solicita leer un bloque que ya está ubicado en la caché, por lo que se produce un acierto y el estado de este bloque no se modifica.</p>                                                                                                                                                                                                                                                                                                                                                                  |
|                                | S                               | PRMiss                | <p>El procesador local solicita leer un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal con un comando RMiss que resuelva este fallo de lectura.</p> <p>La copia del bloque que se recibe puede ser la única del sistema o puede estar compartida.</p> <p>Esta copia reemplaza a la de un bloque que estaba en estado modificado (M), por lo que antes de reemplazarlo hay que volcar las modificaciones realizadas a memoria principal (post-escritura) con una acción WB.</p> |

| ESTADO ORIGEN DE LA TRANSICIÓN | ESTADO DESTINO DE LA TRANSICIÓN | EVENTO QUE LA PROVOCÁ | EXPLICACIÓN                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------------------|---------------------------------|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                | M                               | PWHit                 | El procesador local solicita escribir un bloque que ya está ubicado en la caché y en exclusiva, por lo que se produce un acierto y el estado de este bloque no se modifica.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|                                | M                               | PWMiss                | <p>El procesador local solicita escribir un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal con un comando WMiss que resuelva este fallo de escritura. Este comando sirve al mismo tiempo como orden de invalidación de este bloque para sus posibles copias en otras cachés.</p> <p>La copia que se recibe del bloque es exclusiva, no hay más copias en las cachés del sistema.</p> <p>Esta copia reemplaza a la de un bloque que también estaba en estado modificado (M), por lo que antes de reemplazarlo hay que volcar las modificaciones realizadas a memoria principal (post-escritura) con una acción WB.</p> |
|                                | S                               | BusRMiss              | <p>Se observa en el bus compartido un fallo de lectura de otro procesador por un bloque que está ubicado en la caché local y no es coherente con memoria principal.</p> <p>Se realiza una actualización de la memoria principal con una acción de WB para que se pueda resolver el fallo y se cambia de estado al bloque puesto que ya no es copia exclusiva y pasa a ser coherente con memoria principal.</p>                                                                                                                                                                                                                                                          |
|                                | I                               | BusWMiss              | <p>Se observa en el bus compartido un fallo de escritura de otro procesador por un bloque que está ubicado en la caché local y no es coherente con memoria principal.</p> <p>Se realiza una actualización de la memoria principal con una acción de WB para que se pueda resolver el fallo y se cambia de estado al bloque puesto que el WMiss de otro procesador lleva una orden de invalidación implícita. Es decir, se invalida la copia local del bloque.</p>                                                                                                                                                                                                       |

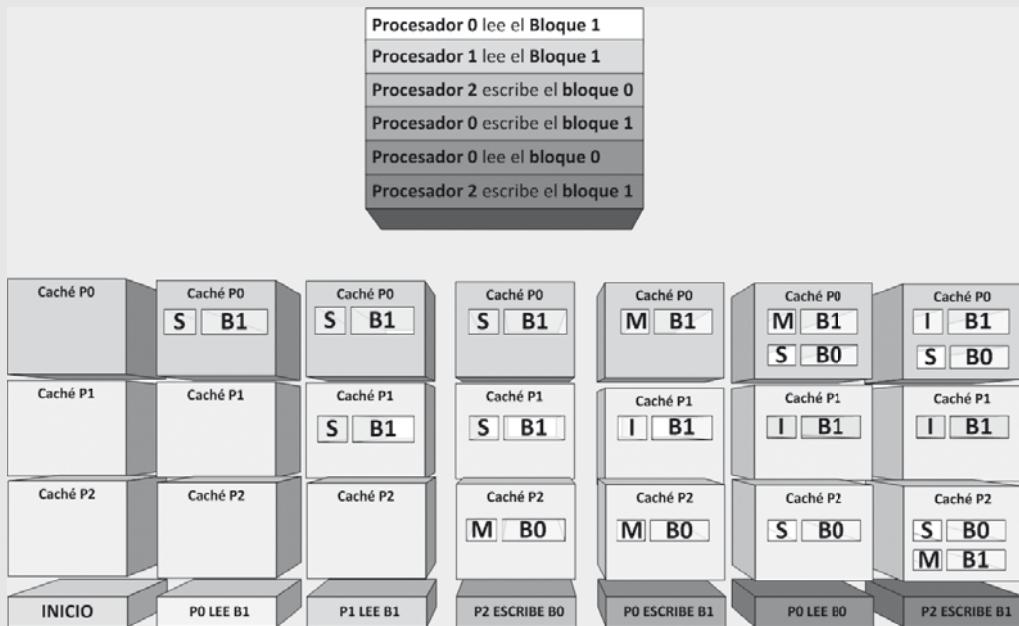
### Ejemplo 5.5

Aplicación práctica del protocolo del ejemplo 5.4.

Para comprender mejor el funcionamiento de este protocolo, vamos a plantear un caso práctico en el que se utilizará para mantener la coherencia de una arquitectura de memoria compartida.

Supongamos que tenemos una arquitectura de este tipo con tres procesadores (P0, P1 y P2) cuyas cachés están inicialmente vacías. ¿Cómo evolucionan los contenidos de las cachés y el protocolo si se ejecuta la siguiente secuencia de lecturas y escrituras en los bloques 0 y 1 de MP?

\* Nota: Las lecturas y escrituras se realizan con instrucciones de load y store a palabras concretas, en este caso estamos simplificando la manera de expresar el patrón de accesos a memoria porque lo único que nos importa es en qué bloques se encuentran esas palabras y si se solicita una lectura o una escritura.



| Lectura/escritura | Evento local         | Comandos y acciones                                                                                                                                                                                   |
|-------------------|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P0 lee B1         | Fallo de lectura     | P0 vuelca un comando RMiss al bus.<br>MP resuelve el fallo de lectura.                                                                                                                                |
| P1 lee B1         | Fallo de lectura     | P1 vuelca un comando RMiss al bus.<br>P0 detecta este comando (evento BusRMiss) pero no modifica el estado de su copia.<br>MP resuelve el fallo de lectura.                                           |
| P2 escribe B0     | Fallo de escritura   | P2 vuelca un comando WMiss al bus.<br>MP resuelve el fallo de escritura.                                                                                                                              |
| P0 escribe B1     | Acierto de escritura | P0 vuelva un comando WMiss al bus.<br>P1 detecta este comando (evento BusWMiss) e invalida su copia de B1.<br>MP resuelve el fallo de escritura (en este caso innecesario).                           |
| P0 lee B0         | Fallo de lectura     | P0 vuelca un comando RMiss al bus.<br>P2 detecta este comando (evento BusRMiss) y hace un WB de B0 a memoria principal, modificando además el estado de su copia.<br>MP resuelve el fallo de lectura. |
| P2 escribe B1     | Fallo de escritura   | P2 vuelca un comando WMiss al bus.<br>P0 detecta este comando (evento BusWMiss) y hace un WB de B1 a memoria principal, invalidando después su copia.<br>MP resuelve el fallo de escritura.           |

Con este ejemplo se puede observar que este sencillo protocolo de tres estados y dos comandos resuelve el problema de la coherencia.

Obviamente, esta sencillez hace que su comprensión e implementación sea muy sencilla pero también presenta algunos inconvenientes, especialmente dos:

- Cuando un procesador tiene un acierto local de escritura envía un comando WMiss por el bus compartido ya que este comando lleva una orden de invalidación implícita para el resto de cachés. El problema es que la memoria principal responde a este comando resolviendo un fallo que realmente no se ha producido, lo que supone un tráfico innecesario en el bus compartido (un bloque completo en cada acierto de escritura de un procesador). La manera de evitar esto sería añadir un comando de invalidación explícito, Inv, independiente del comando de fallo de escritura WMiss.
- Aunque resolvamos este problema, siempre que un procesador solicite escribir en un determinado bloque enviará una orden de invalidación para este bloque por el bus compartido (sea con WMiss o con Inv si lo hemos añadido como comando) ya que no tiene forma de saber si su copia es la única del sistema o si por el contrario, existen copias del mismo bloque en otras cachés. Si su copia es única, este comando no sirve para nada, por lo que de nuevo tendríamos tráfico innecesario en el bus compartido. La manera de evitar esto sería añadir un cuarto estado, Exclusivo (E), que es para los bloques que son copia única en el sistema y son coherentes con memoria principal. De esta manera se diferencia con facilidad en qué situaciones son necesarias las invalidaciones (cuando el bloque está compartido y hay que invalidar el resto de copias) y en qué situaciones no (cuando la copia del bloque es única y se puede pasar a escribir sobre él directamente). Estos protocolos de 4 estados suelen denominarse MESI y son ampliamente utilizados.

### Ejemplo 5.6

Protocolo de coherencia de caché basado en espionaje, de 3 estados y con actualización.  
Cachés de post-escritura.

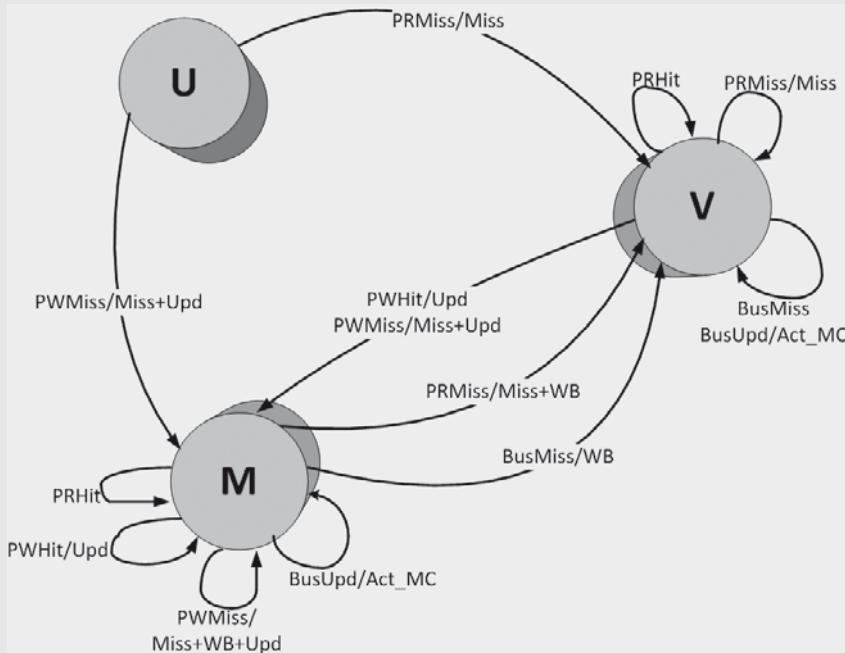
En este ejemplo tenemos de nuevo un protocolo que define tres estados posibles para los bloques ubicados en caché:

- Estado inicial o uncached (U). El bloque no está en la caché, por lo que cuando el procesador lo solicita para leer o para escribir se produce un fallo.
- Válido (V). El bloque está en la caché y no ha sido modificado mediante escrituras locales. Puede que existan otras copias en las cachés de otros procesadores, todas ellas coherentes entre sí.
- Modificado (M). El bloque está en la caché y no es coherente con la memoria principal porque el procesador local ha realizado al menos una escritura sobre él. Puede que existan otras copias en las cachés de otros procesadores, todas ellas coherentes entre sí.

\*Nota: Con esta definición es posible que un bloque esté en estado Modificado en más de una memoria caché del sistema.

Este protocolo utiliza dos comandos para mantener la coherencia en el sistema, Miss (fallo) y Upd (actualización de la palabra modificada). Por último, se contemplan dos acciones, WB (volcado de un bloque completo al bus compartido para actualizar su contenido en la memoria principal) y ActMC (actualización de la memoria caché local con los contenidos de un comando Upd en el bus compartido).

El diagrama de transición de estados para este protocolo es el siguiente:



Analicemos las transiciones que se producen desde cada uno de los estados:

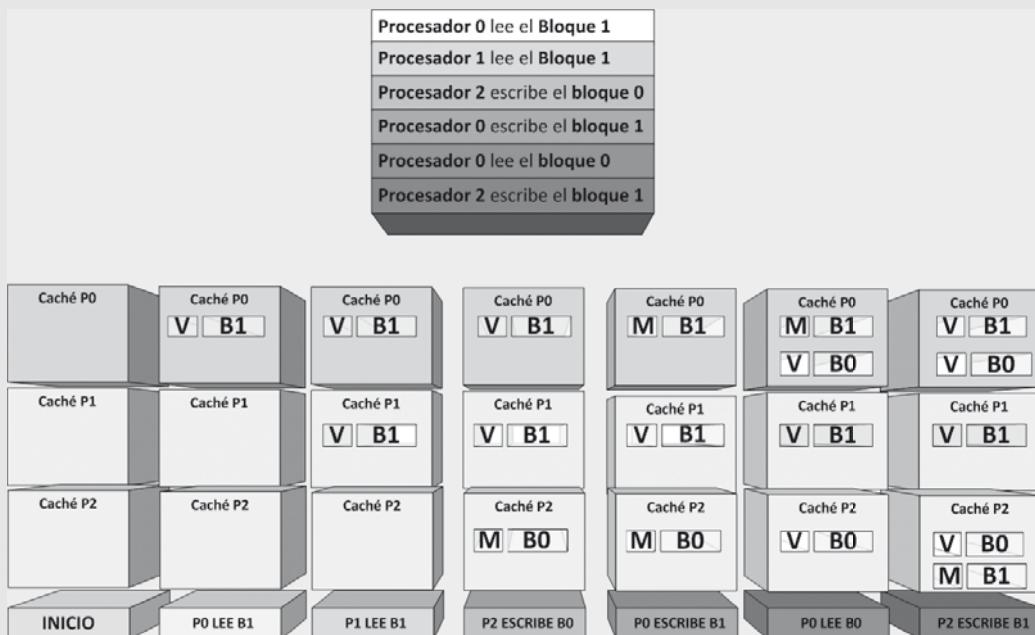
| ESTADO DE ORIGEN DE LA TRANSICIÓN | ESTADO DESTINO DE LA TRANSICIÓN | EVENTO QUE LA PROVOCABA | EXPLICACIÓN                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------------------------------|---------------------------------|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| U                                 | V                               | PRMiss                  | El procesador local solicita leer un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal, con un comando Miss, que resuelva este fallo de lectura.<br>La copia del bloque que se recibe puede ser la única del sistema o puede estar compartida.                                                                                                                                                                            |
|                                   | M                               | PWMiss                  | El procesador local solicita escribir un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal, con un comando Miss, que resuelva este fallo de escritura.<br>La copia del bloque que se recibe puede ser la única del sistema o puede estar compartida.<br>Cuando se realiza la escritura localmente se envía con un comando Upd el nuevo valor de la palabra escrita por el bus compartido.                                 |
| V                                 | V                               | PRHit                   | El procesador local solicita leer un bloque que ya está ubicado en la caché, por lo que se produce un acierto y el estado de este bloque no se modifica.                                                                                                                                                                                                                                                                                                                 |
|                                   | V                               | PRMiss                  | El procesador local solicita leer un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal, con un comando Miss, que resuelva este fallo de lectura.<br>La copia del bloque que se recibe puede ser la única del sistema o puede estar compartida.<br>Esta copia reemplaza a la de un bloque que también estaba en estado válido (V), por lo que la etiqueta asociada al marco escogido para el emplazamiento no se modifica. |

| ESTADO DE ORIGEN DE LA TRANSICIÓN | ESTADO DESTINO DE LA TRANSICIÓN | EVENTO QUE LA PROVOCÁ | EXPLICACIÓN                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------|---------------------------------|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                   | M                               | PWHit                 | El procesador local solicita escribir un bloque que ya está ubicado en la caché, por lo que se produce un acierto.<br>Cuando se realiza la escritura localmente se envía con un comando Upd el nuevo valor de la palabra escrita por el bus compartido.                                                                                                                                                                                                                                                                                                               |
|                                   | M                               | PWMiss                | El procesador local solicita escribir un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal, con un comando Miss, que resuelva este fallo de escritura.<br>La copia del bloque que se recibe puede ser la única del sistema o puede estar compartida.<br>Cuando se realiza la escritura localmente se envía con un comando Upd el nuevo valor de la palabra escrita por el bus compartido.                                                                                                                              |
|                                   | S                               | BusMiss               | Se observa en el bus compartido un fallo de otro procesador por un bloque que está ubicado en la caché local. Su estado no se modifica.                                                                                                                                                                                                                                                                                                                                                                                                                               |
|                                   | S                               | BusUpd                | Se detecta en el bus compartido una actualización que afecta a un bloque ubicado en la caché. Se realiza la acción ActMC para actualizar el contenido de este bloque con la palabra escrita en otro procesador.                                                                                                                                                                                                                                                                                                                                                       |
| M                                 | M                               | PRHit                 | El procesador local solicita leer un bloque que ya está ubicado en la caché, por lo que se produce un acierto y el estado de este bloque no se modifica.                                                                                                                                                                                                                                                                                                                                                                                                              |
|                                   | V                               | PRMiss                | El procesador local solicita leer un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal, con un comando Miss, que resuelva este fallo de lectura.<br>La copia del bloque que se recibe puede ser la única del sistema o puede estar compartida.<br>Esta copia reemplaza a la de un bloque que estaba en estado modificado (M), por lo que antes de reemplazarlo hay que volcar las modificaciones realizadas a memoria principal (post-escritura) con una acción WB.                                                    |
|                                   | M                               | PWHit                 | El procesador local solicita escribir un bloque que ya está ubicado en la caché, por lo que se produce un acierto y el estado de este bloque no se modifica.<br>Cuando se realiza la escritura localmente se envía con un comando Upd el nuevo valor de la palabra escrita por el bus compartido.                                                                                                                                                                                                                                                                     |
|                                   | M                               | PWMiss                | El procesador local solicita escribir un bloque que no está ubicado en la caché, se produce un fallo y se solicita a la memoria principal con un comando Miss que resuelva este fallo de escritura.<br>Esta copia reemplaza a la de un bloque que también estaba en estado modificado (M), por lo que antes de reemplazarlo hay que volcar las modificaciones realizadas a memoria principal (post-escritura) con una acción WB.<br>Cuando se realiza la escritura localmente se envía con un comando Upd el nuevo valor de la palabra escrita por el bus compartido. |
|                                   | V                               | BusMiss               | Se observa en el bus compartido un fallo de otro procesador por un bloque que está ubicado en la caché local y no es coherente con memoria principal.<br>Se realiza una actualización de la memoria principal con una acción de WB para que se pueda resolver el fallo y se cambia de estado al bloque puesto que pasa a ser coherente con memoria principal.                                                                                                                                                                                                         |
|                                   | M                               | BusUpd                | Se detecta en el bus compartido una actualización que afecta a un bloque ubicado en la caché. Se realiza la acción ActMC para actualizar el contenido de este bloque con la palabra escrita en otro procesador.                                                                                                                                                                                                                                                                                                                                                       |

**Ejemplo 5.7****Aplicación práctica del protocolo del ejemplo 5.6.**

Para comprender mejor el funcionamiento de este protocolo, vamos a plantear un caso práctico en el que se utiliza para mantener la coherencia de una arquitectura de memoria compartida.

Supongamos que tenemos la misma arquitectura que en el caso práctico que hemos empleado para ilustrar el protocolo basado en invalidación y el mismo patrón de lecturas y escrituras en memoria.



| Lectura/escritura | Evento local         | Comandos y acciones                                                                                                                                                                                                                                                         |
|-------------------|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P0 lee B1         | Fallo de lectura     | P0 vuela un comando Miss al bus.<br>MP resuelve el fallo de lectura.                                                                                                                                                                                                        |
| P1 lee B1         | Fallo de lectura     | P1 vuela un comando Miss al bus.<br>P0 detecta este comando (evento BusMiss) pero no modifica el estado de su copia.<br>MP resuelve el fallo de lectura.                                                                                                                    |
| P2 escribe B0     | Fallo de escritura   | P2 vuela un comando Miss al bus.<br>MP resuelve el fallo de escritura.<br>P2 vuela un comando Upd al bus con la palabra que ha escrito localmente.                                                                                                                          |
| P0 escribe B1     | Acierto de escritura | P0 vuela un comando Upd al bus con la palabra que ha escrito localmente y modifica el estado de su copia del bloque.<br>P1 detecta este comando (BusUpd) y realiza una acción ActMC para actualizar el contenido de su memoria caché con la palabra incluida en el comando. |
| P0 lee B0         | Fallo de lectura     | P0 vuela un comando Miss al bus.<br>P2 detecta este comando (evento BusMiss) y hace un WB de B0 a memoria principal, modificando además el estado de su copia.<br>MP resuelve el fallo de lectura.                                                                          |

| Lectura/escritura | Evento local       | Comandos y acciones                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------------|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P2 escribe B1     | Fallo de escritura | <p>P2 vuela un comando Miss al bus.</p> <p>P0 detecta este comando (evento BusMiss) y hace un WB de B1 a memoria principal, modificando además el estado de su copia.</p> <p>MP resuelve el fallo de escritura.</p> <p>P2 vuela un comando Upd al bus con la palabra que ha escrito localmente.</p> <p>P0 y P1 detectan este comando (BusUpd) y realizan una acción ActMC para actualizar el contenido de su memoria caché con la palabra incluida en el comando.</p> |

De nuevo tenemos un sencillo protocolo de tres estados y dos comandos que resuelve el problema de la coherencia.

¿Qué inconvenientes podemos encontrar a la solución explicada en este caso? Nos encontramos un problema similar que en el caso de invalidación; todos los procesadores que realizan una escritura local envían un comando de actualización por el bus. Pero lo hacen sin saber si existe alguna otra copia del bloque en el sistema, porque si no la hay, la actualización es completamente innecesaria.

De nuevo la solución está en añadir un cuarto estado Exclusivo (E), en el que el bloque es coherente con la memoria principal y es la única copia en una caché del sistema.

Gracias a este cuarto estado, sólo se enviarán actualizaciones cuando sea necesario notificar las escrituras locales al resto de copias que haya en otras cachés de la arquitectura. Pero si no existe ninguna otra copia, no se enviará esta actualización, evitando así una sobrecarga innecesaria en el bus compartido.

El otro inconveniente está en que, tal y como se ha definido el funcionamiento del protocolo, todos los procesadores que tengan su bloque en estado modificado harán un WB a memoria principal con un evento BusMiss, es decir, actualizarán la memoria para que pueda resolver el fallo con los datos correctos.

Pero todos estos volcados son innecesarios, basta con que uno de los procesadores haga el WB, ya que todas las cachés tienen exactamente la misma copia de la información.

La manera de resolver este problema es hacer que sólo el último procesador que ha escrito sobre un bloque determinado lo marque como Modificado, mientras que el que ocupaba ese puesto hasta el momento pase a estado Válido para que no responda con un WB cuando se produzca un BusMiss. Bastaría con que un procesador con un bloque en estado Modificado lo pase a Válido cuando observe un BusUpd que afecte a ese bloque (esto significa que otro procesador ha escrito este bloque con posterioridad a él).

Comparando ambas alternativas, invalidación y actualización, cabe señalar que:

- Múltiples escrituras consecutivas del mismo procesador sobre la misma palabra o sobre palabras consecutivas del mismo bloque sin lecturas intermedias, requieren múltiples operaciones de actualización pero sólo una de invalidación.
- La actualización requiere una operación por palabra escrita, la invalidación una por bloque.
- El retardo entre escritura y lectura es menor en actualización pues el dato ya está actualizado en todas las cachés. Con el protocolo de invalidación, para que un procesador escriba en un bloque, el resto de copias se deben invalidar. Por lo tanto, las lecturas posteriores siempre provocan un fallo que hay que resolver antes de poder hacer la lectura con éxito.

Por todo esto, y teniendo en cuenta el gran consumo de ancho de banda del bus que se produce con las actualizaciones, se suelen utilizar más los protocolos de invalidación. Pero en aplicaciones con ciertos patrones de acceso a memoria, el aumento de la tasa de fallos que producen las invalidaciones puede llegar a degradar bastante el rendimiento de la arquitectura, no se trata de una solución perfecta. En algunos casos se diseñan protocolos de coherencia mixtos, que aplican la invalidación o la actualización según la situación para intentar aprovechar las ventajas de ambos tipos de solución.

**Ejemplo 5.8****Comparación de protocolos de coherencia basados en invalidación y en actualización.**

En los ejemplos 5.5 y 5.7 que hemos utilizado para comprender mejor el funcionamiento de los protocolos de invalidación y actualización de tres estados, hemos visto que con el protocolo de invalidación se generan tres comandos de invalidación (dos de ellos son en realidad peticiones de resolución de fallo de tipo WMiss que llevan además la invalidación implícita, y el otro es un WMiss que se envía sólo para invalidar) mientras que con el protocolo de actualización se han generado tres comandos de actualización (uno por cada escritura realizada) que implican el tráfico de tres palabras por el bus compartido.

Si después de la escritura que P2 realiza sobre el bloque 0 se hubieran realizado 255 escrituras más sobre ese mismo bloque por parte de ese procesador (supongamos que en el proceso que se ejecuta en P2 se escriben consecutivamente los 256 elementos de un vector, todos en el bloque 0 de memoria), el protocolo de invalidación seguiría generando tres comandos de invalidación. Sin embargo el protocolo de actualización generaría en total 258 comandos de actualización, uno por cada palabra escrita en el código, lo que supondría un tráfico de 258 palabras por el bus compartido en lugar de las tres palabras que teníamos inicialmente.

Ahora supongamos una nueva modificación en el patrón de accesos a memoria, incluyamos justo al final que P0 lee B1. ¿Qué ocurre en este caso? Si hemos utilizado el protocolo de invalidación, el estado de B1 en la caché de P0 es Inválido, por lo que habrá que resolver un fallo de lectura para poder realizar este acceso. Esto conlleva traer un bloque completo desde memoria principal, supongamos que de 512 palabras.

Sin embargo, si hemos utilizado un protocolo de actualización, esta nueva lectura no provocará ningún fallo y nos ahorraremos el tráfico por el bus compartido generado para la resolución del fallo.

Por lo tanto, que el rendimiento de un tipo de protocolo u otro sea mejor dependerá del patrón de accesos a memoria y del tamaño de bloque.

Cuando las jerarquías de memoria incluyen sólo un nivel de caché, cualquier transacción por el bus implica una comparación de etiquetas que puede interferir con accesos de los procesadores a sus cachés.

Hay que tener en cuenta que si cada procesador genera T transacciones de bus por segundo y hay N procesadores en el sistema, cada procesador deberá examinar  $T^*N$  transacciones por segundo para aplicar el protocolo de coherencia.

Cada uno de estos exámenes implica, como mínimo, buscar en la caché local para comprobar si la dirección involucrada en la transacción se encuentra ubicada en la caché y, si por lo tanto, hay que reaccionar ante la transacción espia en el bus.

Para evitar estas interferencias del protocolo de coherencia con los procesadores se pueden duplicar las etiquetas de la memoria caché. Esto implica la actualización de los dos juegos de etiquetas pero así sólo hay conflicto entre el controlador de caché y el procesador si se produce un acceso simultáneo al mismo bloque desde ambos componentes.

Este problema queda resuelto en casi todos los sistemas actuales, que incorporan caches multinivel en sus jerarquías. De esta manera el procesador trabaja la mayor parte del tiempo con el nivel de caché más cercano y la actividad de monitorización se hace sobre otro nivel.

Normalmente las cachés que llevan a cabo las tareas de espionaje son las de nivel 2, que están conectadas al bus común y la memoria principal compartida (figura 5.17). Pero entonces será necesario completar los protocolos de coherencia, sean de invalidación o de actualización, con algún tipo de mecanismo que permita involucrar también en la solución al problema a las cachés de nivel 1.

La solución más utilizada implica añadir a las cachés de nivel 1 las etiquetas con el estado de los bloques y utilizar una política de escritura directa en esta caché. Así, cualquier modificación en el estado de un bloque en la caché de nivel 1 se propagará inmediatamente a la caché de nivel 2 y al resto de procesadores del sistema a través de la implementación del protocolo de coherencia. Por otro lado, como el principio de inclusión debe estar garantizado en la jerarquía de memoria, cualquier cambio que se produzca en la caché de nivel 2 se deberá reflejar en la caché de nivel 1 inmediatamente, ya que ésta debe contener siempre un subconjunto del contenido de la caché de nivel 2.

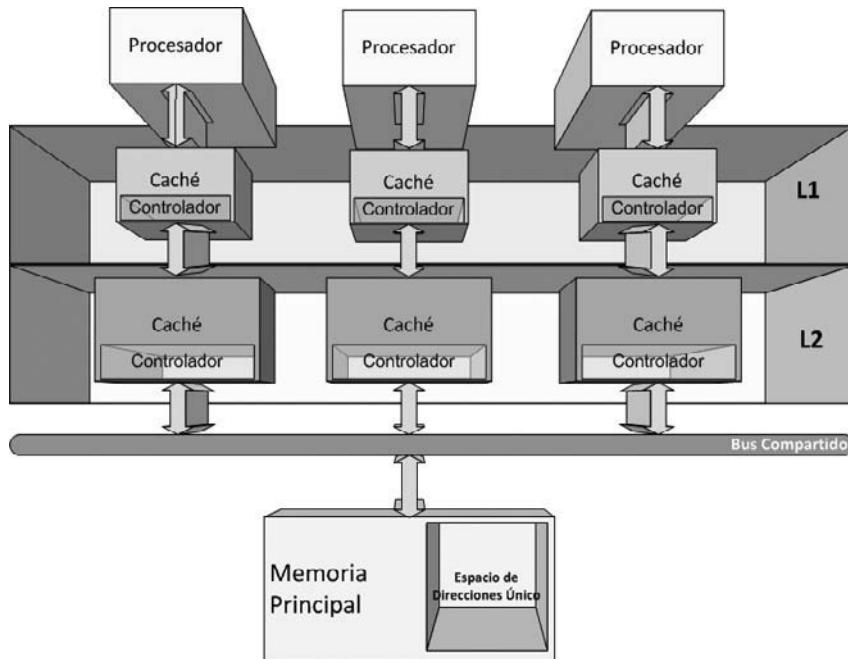


FIGURA 5.17

Protocolo de coherencia basado en espionaje con dos niveles de caché.

Por último, cabe mencionar que en la mayor parte de los protocolos de coherencia hay que resolver problemas de interbloqueo e inanición. Por ejemplo, ¿qué ocurre si dos procesadores vuelcan un comando de escritura en exclusiva al mismo tiempo en el bus? Será el arbitraje del bus el que decida cuál de los comandos se transmitirá primero. Esto hará que el procesador cuyo comando quede en segundo lugar tenga que invalidar el bloque en su memoria caché y no pueda realizar la escritura, sino que tendrá que enviar un comando de fallo de escritura por el bloque que acaba de invalidar y sobre el que quería escribir.

Para manejar este tipo de situaciones muchos protocolos terminan introduciendo estados intermedios Compartido/Modificado, de manera que un bloque pasa a este estado cuando se hace la petición de escritura en exclusiva y dependiendo del resultado de esta solicitud, se pasa al estado Modificado o no.

Una vez discutidos estos problemas prácticos de implementación, hay que señalar que la principal desventaja de los protocolos de coherencia de espionaje, sea cual sea la opción de diseño escogida, está en su falta de escalabilidad. En sistemas con un gran número de procesadores el tiempo en el que puede resolverse un fallo de caché acaba limitado siempre por los tiempos relacionados con las tareas de espionaje y comprobación, en lugar de con la latencia del bus y de la memoria principal.

### 5.3.2. Soluciones para el problema de la consistencia

Este problema se resuelve definiendo un modelo de consistencia para completar el repertorio de instrucciones de la arquitectura de memoria compartida. Es decir, el problema de la consistencia no se resuelve desde el hardware, sino que se resuelve definiendo un modelo que ordene las referencias a la memoria compartida de los diferentes procesadores.

El modelo de consistencia más sencillo es el de consistencia secuencial, que impone un orden absoluto a todas las referencias a memoria que realizan todos los procesadores. Con un modelo de consis-

tencia secuencial es como si se establecieran turnos para acceder a la memoria compartida (los accesos son atómicos, se realizan de uno en uno), entrelazando las referencias a memoria de los distintos procesadores como si se tratara de un código compilado con multithreading para un sistema monoprocesador.

Dentro de cada procesador, las referencias se ordenan con el orden de programa tradicional, es decir, debe parecer que los accesos a memoria se han realizado en el orden secuencial en el que aparecen en el código. Las referencias realizadas por diferentes procesadores se ordenarán de la forma que especifique el usuario o desarrollador mediante mecanismos de sincronización explícitos, el modelo de consistencia secuencial no propone ninguna forma concreta de establecer los turnos para el acceso a memoria.

Aunque este modelo de consistencia es muy sencillo de comprender e intuitivo para un usuario o desarrollador, las limitaciones que implica imponer un orden total a todas las referencias a memoria son demasiado fuertes, impidiendo utilizar muchas de las optimizaciones que incorporan los compiladores y procesadores actuales, que tienden a no respetar el orden de programa durante la ejecución de los procesos (reordenan instrucciones, eliminan resultados intermedios, etc).

El modelo de consistencia secuencial exige unas características concretas en la arquitectura hardware que lo implementa:

1. Las referencias a memoria realizadas por diferentes procesos deben realizarse de manera atómica. Esto exige que se serialicen las escrituras realizadas a una misma dirección de memoria y que todas las escrituras, sean a la misma o a diferentes direcciones, se vean en el mismo orden en todos los procesadores que componen la arquitectura. Para cumplir con estas exigencias, es necesario que la red incluida en la arquitectura entregue los mensajes en los nodos destino exactamente en el mismo orden en el que se enviaron desde el nodo origen y que se prohíba leer un nuevo valor escrito por un procesador hasta que todas las cachés con copia de este valor hayan recibido todas las invalidaciones o actualizaciones del protocolo de coherencia.
2. Las referencias a memoria realizadas dentro de un proceso deben realizarse en el orden de programa, por lo que no se permite ejecución fuera de orden. Esto implica que los procesadores no pueden utilizar ninguna técnica de aumento de prestaciones que desordene la ejecución de las instrucciones y de nuevo, que la red que incluya la arquitectura garantice que todos los mensajes que se envían por ella llegan a sus destinos exactamente en el mismo orden en el que se han enviado.

Por lo tanto, para utilizar un modelo de consistencia secuencial será necesario que los procesadores, la red y los controladores de coherencia de las cachés, así como el compilador, cumplan con unos requisitos determinados, por norma general, muy difíciles de cumplir.

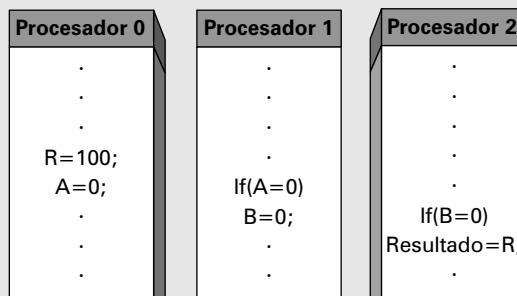
Es por esto que normalmente se utilizan modelos relajados de consistencia. En este caso se exige al usuario o desarrollador que etiquete de manera explícita las referencias a memoria que deben ser ordenadas, rebajando mucho las exigencias que la solución del problema de consistencia impone al hardware y al compilador, ya que así no es necesario mantener la atomicidad de los accesos a memoria o el orden de programa dentro de cada procesador.

Normalmente estos modelos relajan:

1. La atomicidad. Algunos modelos permiten que un procesador lea el valor escrito por otro procesador en una posición de memoria antes de que esta escritura sea visible por los demás procesadores.
2. El orden de programa. Algunos modelos permiten que los accesos a memoria de un único procesador no se realicen en el orden secuencial del programa.

**Ejemplo 5.9****Ejemplo de consistencia secuencial.**

Supongamos que en una arquitectura de memoria compartida con tres procesadores (P0, P1 y P2) se realizan las siguientes operaciones:



\* Nota: No se han incluido en los códigos los mecanismos de sincronización necesarios para acceder a las variables compartidas. Suponemos de momento que este problema está resuelto, ya veremos en la sección siguiente las alternativas que existen.

La pregunta en este caso es, ¿podemos estar seguros con un modelo de consistencia secuencial de que en la variable Resultado quedará almacenado el valor 100?

Si analizamos lo que implica el modelo de consistencia secuencial, que se respeta el orden de programa dentro de cada procesador y que la atomicidad garantiza que todas las escrituras se vean al mismo tiempo en todos los procesadores, podemos deducir que sí, que Resultado=100.

Eso es así porque P0 escribe R=100 y A=0 justo en este orden, entonces P1 ve A=0 y hace B=0 y por último cuando P2 ve B=0, hace Resultado=R=100.

Obviamente, si a P2 llega antes el valor B=0 que el valor R=100, el resultado ya no sería el esperado. Esto podría ocurrir principalmente por dos motivos:

- Por las diferentes latencias de comunicaciones entre unos procesadores y otros de la arquitectura.
- Porque la propagación de una escritura en memoria principal a todas las memorias caché que tienen copia del bloque modificado no son atómicas casi nunca.

Por eso para garantizar la consistencia secuencial es necesario también que la red incluida en la arquitectura entregue los mensajes en los nodos destino exactamente en el mismo orden en el que se enviaron desde el nodo origen y que se prohíba leer un nuevo valor escrito por un procesador hasta que todas las cachés con copia de este valor hayan recibido todas las invalidaciones o actualizaciones del protocolo de coherencia.

Aún cumpliendo estas condiciones, podrían surgir multitud de "imprevistos" similares que con el modelo de consistencia secuencial no nos garantizarían un Resultado=100.

**Ejemplo 5.10****Modelo de ordenación débil para consistencia.**

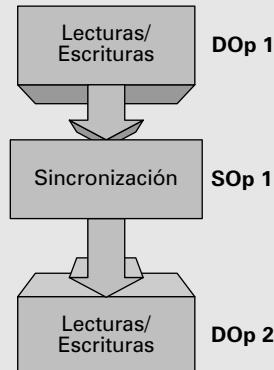
Este modelo es un ejemplo típico de los modelos relajados de consistencia que intentan reducir todas las restricciones impuestas por el modelo de consistencia secuencial transfiriéndole parte de la responsabilidad al usuario o desarrollador. Se basa en mantener el orden de los accesos a memoria sólo cuando existe un código explícito de sincronización que así lo indique.

Según este modelo hay dos clases de accesos a memoria: las operaciones de datos (DOp) y las operaciones de sincronización (SOp). El desarrollador debe etiquetar como operaciones de sincronización aquellas en las que es necesario mantener un orden concreto, el resto de operaciones serán de datos.

Si SOp es una operación de sincronización, el modelo de ordenación débil garantiza:

- Todas las DOp que aparecen antes en el orden de programa se han completado antes de ejecutar la SOp.
- Esta SOp se completa antes que todas las DOp posteriores en el orden de programa.

En el ejemplo de la figura, lo que nos garantiza el modelo de ordenación débil es que todas las lecturas y escrituras del bloque DOp1 finalizarán antes de la ejecución SOp1. Y al mismo tiempo, que esta sincronización se completará siempre antes de realizar las operaciones lectura y escritura del bloque DOp2. No hay ninguna otra garantía, es decir, no hace falta que se cumpla el orden de programa para las DOp del bloque 1, si hubiera otro bloque con SOp sus accesos a memoria podrían reordenarse e intercalarse con el bloque SOp1, etc.



### 5.3.3. Soluciones para el problema de la sincronización

Como se ha mencionado con anterioridad, en las arquitecturas de memoria compartida la comunicación entre procesos se realiza de manera explícita a través de las variables compartidas. Por ello es necesario sincronizar el acceso a estas variables compartidas.

Aunque casi todas las arquitecturas incorporan mejoras en el hardware que ayudan a incrementar el rendimiento de la sincronización y de la comunicación entre procesos (registros explícitos para comunicación, líneas especiales en el bus compartido), de nuevo la solución al problema no viene dada desde el hardware, sino desde el sistema operativo o la programación explícita.

Sin embargo, estas soluciones se basan en instrucciones o primitivas de sincronización que debe ofrecer el repertorio de instrucciones de la arquitectura. Las primitivas atómicas más habituales en los repertorios de instrucciones son las siguientes:

- **Test&Set.** Esta primitiva permite leer una posición de la memoria compartida  $m(x)$ , volcar su contenido a un registro del procesador local R y modificar el valor de la posición de memoria  $m(x)$  haciendo que valga 1. Por tanto se pueden hacer de manera atómica las operaciones  $R=m(x)$  y  $m(x)=1$ .
- **Swap.** Esta primitiva permite intercambiar el contenido de una posición de memoria  $m(x)$  y un registro del procesador local R. Es decir, se completan de manera atómica las operaciones  $m(x)=R$  y  $R=m(x)$ .
- **Compare&Swap.** En este caso el intercambio entre un registro y una posición de memoria viene determinado por el contenido de otro registro. Es decir, si  $m(x)=R1$ , entonces se hace el intercambio  $R2=m(x)$  y  $m(x)=R2$ .
- **Fetch&Op.** Esta primitiva permite volcar el contenido de una posición de memoria a un registro del procesador local y realizar una operación Op entre este contenido y una variable a. Después, se

escribe el resultado de esta operación en la misma posición de memoria. Por lo tanto, se realizan de manera atómica las operaciones  $m(x)=m(x)$  Op a y  $R=m(x)$ .

- **Load Linked&Store Conditional:** La instrucción LL carga una posición de memoria  $m(x)$  en un registro R y reserva el bloque de cache en el que está la posición  $m(x)$ . La instrucción SC intenta actualizar el contenido de la posición de memoria  $m(x)$ . La actualización sólo tiene éxito si se mantiene la reserva en la caché. La reserva se pierde si se recibe una orden de invalidación de otro procesador, si se produce un reemplazamiento de ese bloque en la caché o si se produce un cambio de contexto. Si la actualización no tiene éxito porque la reserva se ha perdido, no se produce el Store.

Con estas primitivas se puede realizar sincronización mediante exclusión mutua, es decir, se puede garantizar que sólo un proceso puede acceder a la memoria compartida en un momento dado (o a cualquier otro recurso compartido).

Para llegar a esta exclusión mutua, suelen utilizarse cerros basados en las primitivas de sincronización. Sea cual sea la forma en la que se implementa el cerrojo, siempre se basa en dos acciones.

La primera, el cierre del cerrojo o lock, con la que un proceso intenta adquirir el derecho a utilizar el recurso compartido. Por lo tanto, si varios procesos intentan cerrar el cerrojo al mismo tiempo, sólo uno de ellos lo debe conseguir. Los demás deben quedar a la espera de utilizar el recurso compartido cuando les llegue su turno.

La segunda, la apertura de cerrojo o unlock, con la que se libera el recurso compartido. Si ningún proceso estaba esperando por él, el próximo en hacer un lock, podrá pasar a utilizarlo directamente. Si había procesos esperando por el recurso, uno de ellos podrá hacer el lock.

### Ejemplo 5.11

#### Ejemplo de cerrojo simple.

Veamos primero un ejemplo de utilización de cerrojo para conseguir exclusión mutua en el acceso a una variable compartida.

Supongamos que estamos acumulando en una variable acum la suma de todos los elementos de un vector de longitud K:

```
for(i=0; i<k; i++)
 acum = acum + vector[i];
```

Si ejecutamos este mismo código en una arquitectura de memoria compartida, a cada procesador le corresponde realizar la acumulación parcial de una parte de los elementos del vector (desde el índice inicial hasta el índice final). Después se hace la suma de todos los acumuladores parciales en la variable global acum:

```
for(i=inicial; i<final; i++)
 parcial = parcial + vector[i];
lock (cerrojo)
 acum = acum + parcial;
unlock (cerrojo)
```

Sección Crítica

Para garantizar la exclusión mutua en el acceso a la variable compartida acum se ha utilizado un cerrojo, que permite que se serialicen las sumas de las acumulaciones parciales realizadas por los diferentes procesadores.

La implementación de las funciones lock y unlock para el manejo del cerrojo dependerá de las primitivas de sincronización que estén disponibles en la arquitectura.

Por ejemplo, la función de lock se puede implementar con Test&Set:

```
while (Test & Set (cerrojo) == 1)
{}
```

Pero también con Compare&Swap:

```
Aux=1;
do
 compare & swap (0, aux, cerrojo);
while(aux == 1)
 compare & swap (0, aux, cerrojo)
 if (cerrojo == 0)
 temp = cerrojo;
 cerrojo = aux;
 aux = temp;
```

O con Fetch&Op, escogiendo como operación el OR:

```
while (Fetch & OR(cerrojo,1) == 1)
{}
```

Además de la utilización de cerrojos para conseguir la exclusión mutua, existen otros mecanismos para la sincronización y comunicación entre procesos. Por ejemplo, se pueden utilizar barreras para la sincronización global, pero estas soluciones escapan del alcance de este libro.

## 5.4 Diseño de arquitecturas de memoria compartida-distribuida

Como ya se ha mencionado con anterioridad, este tipo de arquitecturas permiten mejorar la escalabilidad de las de memoria compartida al eliminar el cuello de botella que suponen la memoria principal compartida y el bus.

Este bus se sustituye por una red de comunicación más sofisticada que permite un mayor ancho de banda, de manera que la memoria se distribuye entre los nodos, pero se tiene un único espacio de direcciones.

Los problemas que surgen al trabajar con este tipo de arquitecturas son los mismos que con las de memoria compartida, ya que existe un único espacio de direcciones físicas compartido: coherencia, consistencia y sincronización.

La consistencia y la sincronización se solucionan con las mismas alternativas que en el caso de las arquitecturas de memoria distribuida (teniendo en cuenta que la red de comunicaciones ya no es un bus y esto puede introducir algo más de complejidad), pero la solución para el problema de la coherencia es completamente diferente que en estas arquitecturas, ya que al no existir un bus de medio compartido no puede utilizarse como herramienta el espionaje.

#### 5.4.1. Soluciones para el problema de la coherencia

En el caso de tener un único espacio de direcciones distribuido sobre diferentes memorias físicas, la solución más sencilla para resolver el problema de la coherencia es marcar los datos compartidos para no llevarlos a las cachés de los diferentes procesadores. Obviamente esta solución presenta dos desventajas importantes. La primera, puede llegar a ser muy complicado que el compilador o el programador marquen estos datos compartidos que deben ser siempre accedidos en memoria principal. La segunda, el rendimiento de los accesos a memoria empeora drásticamente para los datos compartidos, ya que no se puede aprovechar la localidad espacial como se hace habitualmente y es como si no existiera una jerarquía de memoria.

La solución son los protocolos de coherencia basados en directorio, ya que los de snoopy no pueden utilizarse con las redes de comunicaciones utilizadas en este tipo de arquitecturas (no son de medio compartido).

El directorio es, en primera instancia, una estructura centralizada que mantiene el estado de todos los bloques que contienen las memorias principales del sistema (figura 5.18).

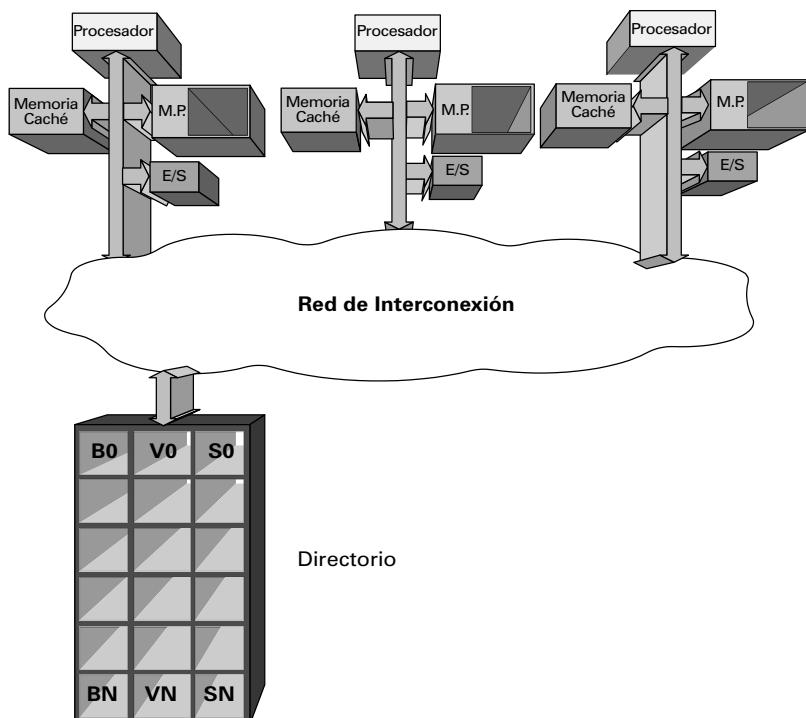


FIGURA 5.18

Protocolo de coherencia basado en directorio centralizado.

Es decir, se asocia una entrada en el directorio a cada bloque de memoria principal y se mantiene actualizado su estado en cada caché del sistema. Es importante que se mantenga información sobre qué cachés tienen copias de cada bloque (vector de compartidos, V) y si lo han modificado (estado del bloque, S).

El vector de compartidos suele contener tantos elementos como nodos tiene el sistema, e indica qué nodos tienen una copia compartida de cada bloque en su caché poniendo un 1 en la posición que corresponde a ese nodo en el vector. Cuando se produce una escritura en un bloque en estado compartido sólo es necesario enviar un mensaje de invalidación a los procesadores indicados en este vector.

Hay que señalar que los protocolos basados en directorio siempre funcionan mediante invalidación, ya que un mecanismo de actualización resultaría demasiado costoso con una red como la que incluyen las arquitecturas de memoria compartida-distribuida (haría falta un mensaje nuevo para cada procesador con copia de un bloque en su caché por cada palabra que se modificara de éste).

Para evitar que el directorio se convierta en un nuevo cuello de botella, la mayor parte de las arquitecturas de memoria compartida-distribuida actuales distribuyen el directorio entre todos los nodos del sistema (figura 5.19). Normalmente a cada nodo le corresponden las entradas del directorio que almacenan el estado de los bloques ubicados en su memoria principal. Hay que recordar que en estas arquitecturas el espacio de direcciones está dividido de forma estática entre los procesadores y por lo tanto esta división es conocida por todos ellos.

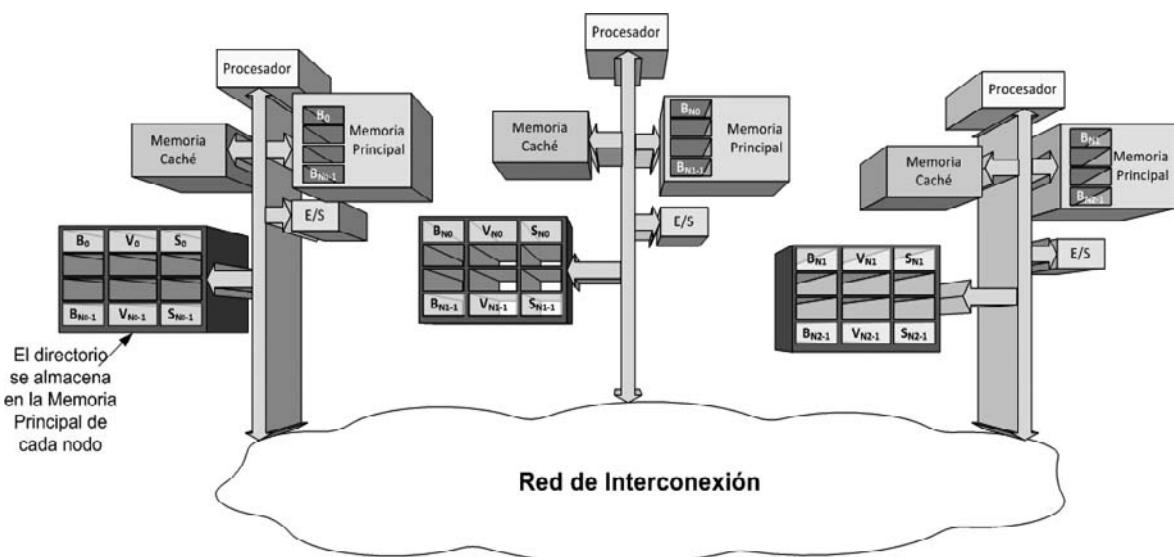


FIGURA 5.19

Protocolo de coherencia basado en directorio distribuido.

Cuando el directorio está distribuido se suele utilizar la siguiente nomenclatura:

- **Nodo local.** Origina la solicitud de lectura o escritura de un dato.
- **Nodo propietario.** Nodo en el que residen el dato (en la memoria principal) y su estado (en la entrada del directorio que corresponda).

- **Nodo remoto.** Nodo que tiene copias adicionales del dato en su memoria caché, en estado exclusivo o compartido.

Los tres nodos pueden coincidir, o bien dos de ellos, y el protocolo no varía salvo en que los mensajes generados son locales (no hace falta que salgan a la red de comunicaciones para enviarse).

Cabe señalar que para que este tipo de protocolos funcionen se debe cumplir la hipótesis de consistencia secuencial, es decir, que todos los mensajes se reciben en el mismo orden en el que se envían.

De nuevo es necesario añadir a la arquitectura controladores de coherencia para resolver el problema de la coherencia desde el hardware. Pero en este caso hay que diseñar dos tipos de controladores diferentes, los de las memorias cachés y los de los directorios.

Por lo tanto, para diseñar un protocolo de este tipo es necesario especificar:

- Los posibles estados de un bloque alojado en un determinado marco de la caché de un procesador, desde el punto de vista de la propia caché y desde el punto de vista del directorio.
- Los mensajes del protocolo, es decir, las órdenes o peticiones que los controladores envían por la red a otros controladores mediante primitivas de envío Send. Es necesario especificar el tipo de mensaje, su contenido, y los nodos origen y destino del mismo.
- El diagrama de transición de estados de los bloques en caché y en el directorio, con los eventos que pueden provocar estas transiciones y las acciones que hay que llevar a cabo ante determinados eventos, transiciones o comandos.

De nuevo existen multitud de alternativas para la implementación de los protocolos de coherencia, por lo que sigue siendo difícil proponer una notación general. Sin embargo, para la definición de eventos, se puede mantener la diferenciación entre eventos locales y globales.

Los eventos locales son exactamente los mismos que en el caso de arquitecturas de memoria compartida.

En el caso de los eventos globales, siempre se utiliza en este libro la notación Recv(mensaje) para identificarlos. Es decir, habrá siempre un tipo de evento global por cada tipo de mensaje del protocolo.

Por ejemplo, **Recv(Inv)** sería el evento en el que se recibe en el nodo local un mensaje de invalidación desde el nodo propietario. Aunque en algunos casos estas comunicaciones sean locales, siempre existe comunicación explícita para que la implementación del protocolo sea más sencilla.

El comportamiento del controlador de la memoria caché es muy similar al del diseñado para arquitecturas de memoria compartida, pero los eventos globales pasan de ser comandos observados o espíados en el bus, a ser recepciones de mensajes por la red.

En cuanto al controlador de un directorio, en general puede recibir tres tipos de mensajes:

- Fallo de lectura.
- Fallo de escritura.
- Post-escritura.

Y la recepción de estos mensajes puede causar:

- Actualización del estado de un bloque o del vector de compartidos en el directorio.
- Envío de mensajes en respuesta a solicitudes (para resolver un fallo, para realizar una invalidación etc).

En la tabla 5.1 se observa un conjunto de mensajes típico para la implementación de un protocolo de coherencia de caché sencillo basado en directorio.

**TABLA 5.1**

Conjunto de mensajes para la implementación de un protocolo de coherencia de caché basado en directorio distribuido.

| Etiqueta         | Origen                 | Destino                | Contenido    | Significado                                                                             |
|------------------|------------------------|------------------------|--------------|-----------------------------------------------------------------------------------------|
| <b>RMiss</b>     | Caché Local            | Directorio Propietario | P, A         | El nodo P tiene un fallo de lectura por la dirección A.                                 |
| <b>WMiss</b>     | Caché Local            | Directorio Propietario | P, A         | El nodo P tiene un fallo de escritura por la dirección A.                               |
| <b>Inv</b>       | Directorio Propietario | Caché Remota           | A            | Invalidación del bloque que contiene la dirección A.                                    |
| <b>Fetch</b>     | Directorio Propietario | Caché Remota           | A            | Orden de post-escritura del bloque que contiene la dirección A.                         |
| <b>Fetch+Inv</b> | Directorio Propietario | Caché Remota           | A            | Orden de post-escritura seguida de Invalidación del bloque que contiene la dirección A. |
| <b>Reply</b>     | Directorio Propietario | Caché Local            | Bloque       | Envío de un bloque para resolver un fallo.                                              |
| <b>WB</b>        | Caché Remota           | Directorio Propietario | P, A, Bloque | El nodo P realiza una post-escritura del bloque que contiene la dirección A.            |

### **Ejemplo 5.12**

Protocolo de coherencia de caché basado en directorio distribuido, de 3 estados en caché y en directorio con invalidación.

En este ejemplo tenemos un protocolo que define tres estados posibles para los bloques ubicados en caché:

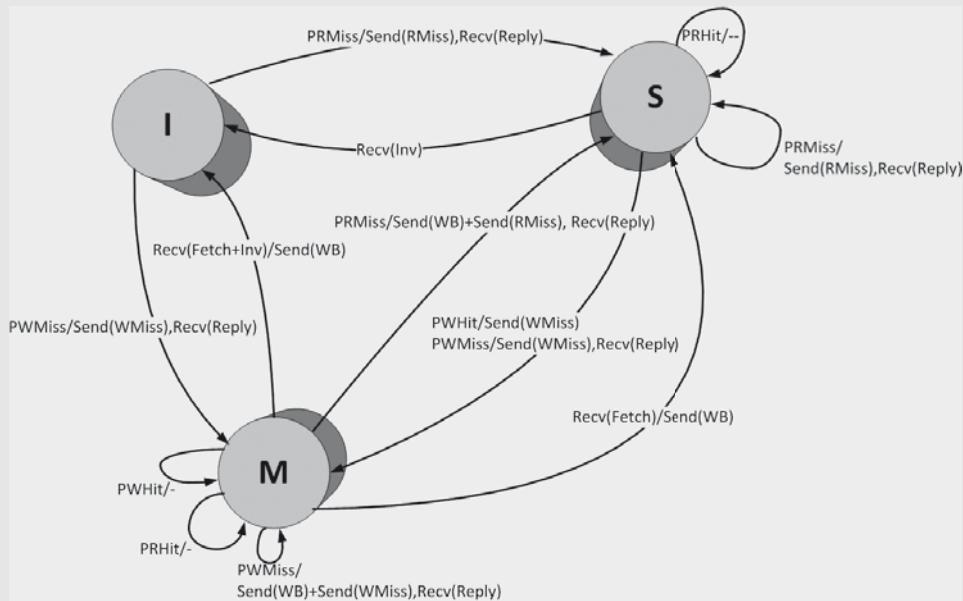
- Inválido (I). El bloque no está en la caché y si lo ha estado en algún momento, ha sido invalidado, por lo que cuando el procesador lo solicita para leer o para escribir se produce un fallo.
- Compartido o Shared (S). El bloque está en la caché y es coherente con la memoria principal. Puede que existan otras copias en estas mismas condiciones en las cachés de otros procesadores.
- Modificado (M). El bloque está en la caché y no es coherente con la memoria principal. Esto significa que se ha realizado al menos una escritura sobre él y que por lo tanto, es la única copia en las cachés del sistema.

Es decir, son los mismos estados que ya utilizamos en el protocolo MSI basado en espionaje (ejemplo 5.4).

En cuanto a los estados que puede tener un bloque desde el punto de vista del directorio son tres muy similares:

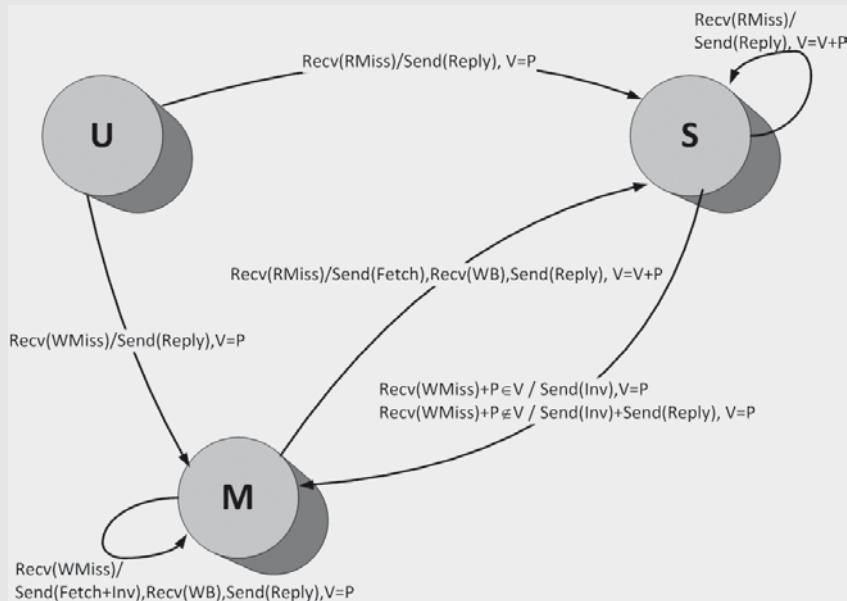
- Uncached o no cacheado (U). El bloque no está en ninguna caché del sistema.
- Compartido o Shared (S). El bloque está en una o más cachés y es coherente con la memoria principal.
- Modificado (M). El bloque está en una única caché y no es coherente con la memoria principal. Esto significa que se ha realizado al menos una escritura sobre él y que por lo tanto, si había otras copias de este bloque en otras cachés, se habrán invalidado.

Los mensajes que utiliza este protocolo son los que hemos visto en la tabla 5.1. El diagrama de transición de estados desde el punto de vista de las cachés para este protocolo es el siguiente:



No es necesario analizar las transiciones de este diagrama en profundidad ya que son las mismas que en el protocolo MSI basado en espionaje del ejemplo 5.4. La principal diferencia es que en este caso los eventos globales no consisten en detectar un comando en el bus compartido sino en recibir un mensaje del directorio.

La novedad está más bien en el diagrama de transición de estados desde el punto de vista del directorio:



Para simplificar el diagrama sólo se ha puesto una vez el envío del mensaje de invalidación (Send(Inv)) por parte del directorio, pero se envía un mensaje de este tipo a cada nodo que aparece en el vector de compartidos del bloque involucrado.

También para simplificar el diagrama y que se comprenda mejor, no se han incluido los mensajes que pueden llegar al directorio cuando las distintas cachés del sistema remplazan sus bloques sucios (sería un evento Recv(WB) espontáneo, sin que haya habido un mensaje de Fetch o de Fetch+Inv previamente). En este caso, el mensaje de WB lleva un bloque en su interior, por lo que lo único que tiene que hacer el directorio es actualizar el vector de compartidos del bloque haciendo  $V=0$ , y pasar del estado M al estado U. En el caso de reemplazamientos de bloques que no han sido modificados, se utiliza el mensaje con etiqueta WB pero vacío (no hace falta actualizar el bloque en la memoria principal), esto significa que la caché ha remplazado un bloque limpio. Por lo que el directorio actualiza el vector de compartidos y deja al bloque en el estado S, a no ser que el procesador del que llegue el mensaje fuera el único con copia del bloque, por lo que se pasa también al estado U ya que  $V=0$ .

Analicemos en este caso las transiciones que se producen desde cada uno de los estados:

| ESTADO ORIGEN DE LA TRANSICIÓN | ESTADO DESTINO DE LA TRANSICIÓN | EVENTO QUE LA PROVOCABA               | EXPLICACIÓN                                                                                                                                                                                                                                                                                       |
|--------------------------------|---------------------------------|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| U                              | S                               | Recv(RMiss)                           | El directorio envía copia del bloque solicitado al procesador local y pone un 1 en el vector de compartidos en la posición que corresponde.                                                                                                                                                       |
|                                | M                               | Recv(WMiss)                           | El directorio envía copia del bloque solicitado al procesador local y pone un 1 en el vector de compartidos en la posición que corresponde.                                                                                                                                                       |
| S                              | S                               | Recv(RMiss)                           | El directorio envía copia del bloque solicitado al procesador local y añade un 1 en el vector de compartidos en la posición que corresponde.                                                                                                                                                      |
|                                | M                               | Recv(WMiss) y P comparte el bloque    | El directorio envía un mensaje Inv a todos las cachés remotas que tienen copia del bloque en el que el procesador local va a escribir. Actualiza el vector de compartidos dejando un único 1 en la posición del procesador que va a realizar la escritura.                                        |
|                                | M                               | Recv(WMiss) y P no comparte el bloque | El directorio envía un mensaje de invalidación a todas las cachés remotas que tienen copia del bloque en el que el procesador local va a escribir.<br>El directorio envía copia del bloque solicitado al procesador local y pone un 1 en el vector de compartidos en la posición que corresponde. |
| M                              | S                               | Recv(RMiss)                           | El directorio envía un mensaje de Fetch a la caché remota que tiene el bloque modificado.<br>Cuando recibe el WB desde esta caché, el directorio envía copia del bloque solicitado al procesador local y pone un 1 en el vector de compartidos en la posición que corresponde.                    |
|                                | M                               | Recv(WMiss)                           | El directorio envía un mensaje de Fetch+Inv a la caché remota que tiene el bloque modificado.<br>Cuando recibe el WB desde esta caché, el directorio envía copia del bloque solicitado al procesador local y pone un único 1 en el vector de compartidos en la posición que corresponde.          |

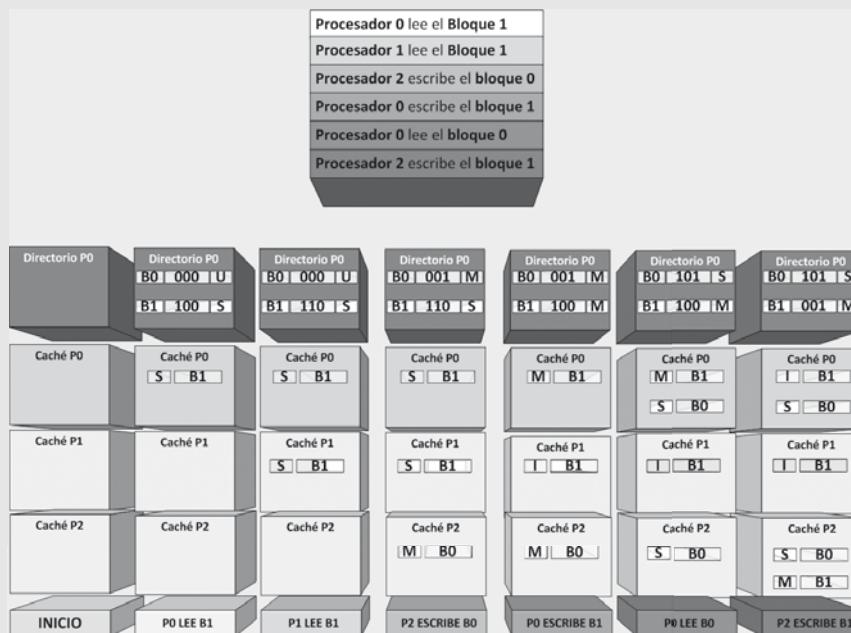
Se puede observar que con el conjunto de etiquetas que manejamos en este ejemplo, cuando un procesador que tiene un acierto de escritura quiere notificar al directorio que va a realizar una escritura sobre ese bloque que ya tiene, lo hace con un mensaje WMiss (ya que no tenemos disponible el WHit).

Esta notificación es imprescindible para que el directorio invalide el resto de posibles copias de este bloque en el sistema. Pero como puede comprobar, mediante el vector de compartidos, que aunque el mensaje recibido es un WMiss, el procesador local ya tiene una copia del bloque en su caché, no le responde con un Reply innecesario.

**Ejemplo 5.13****Aplicación práctica del protocolo del ejemplo 5.12.**

Para comprender mejor el funcionamiento de este protocolo, vamos a plantear un caso práctico en el que se utilizará para mantener la coherencia de una arquitectura de memoria compartida-distribuida.

Supongamos que tenemos una arquitectura de este tipo con tres procesadores (P0, P1 y P2) cuyas cachés están inicialmente vacías. ¿Cómo evolucionan los contenidos de las cachés, de los directorios y el protocolo si se ejecuta la siguiente secuencia de lecturas y escrituras en los bloques 0 y 1 de MP? Supongamos que ambos bloques se encuentran ubicados en la memoria principal del P0 y que el directorio está distribuido, de manera que sólo nos interesa la evolución del directorio de P0.



\* Nota: Las lecturas y escrituras se realizan con instrucciones de load y store a palabras concretas, en este caso estamos simplificando la manera de expresar el patrón de accesos a memoria porque lo único que nos importa es en qué bloques se encuentran esas palabras y si se solicita una lectura o una escritura.

En la siguiente tabla, los mensajes se especificarán con un sencillo formato (etiqueta, dirección de bloque, destino), dónde DIR\_P0 es el directorio de P0 y MC\_P0 es la memoria caché del procesador 0, por ejemplo.

| Lectura/escritura | Evento local         | Mensajes                                                      |
|-------------------|----------------------|---------------------------------------------------------------|
| P0 lee B1         | Fallo de lectura     | MC_P0: Send (RMiss,B1,DIR_P0)<br>DIR_P0: Send(Reply,B1,MC_P0) |
| P1 lee B1         | Fallo de lectura     | MC_P1: Send (RMiss,B1,DIR_P0)<br>DIR_P0: Send(Reply,B1,MC_P1) |
| P2 escribe B0     | Fallo de escritura   | MC_P2: Send (WMiss,B0,DIR_P0)<br>DIR_P0: Send(Reply,B0,MC_P2) |
| P0 escribe B1     | Acierto de escritura | MC_P0: Send (WMiss,B1,DIR_P0)<br>DIR_P0: Send(Inv,B1,MC_P1)   |

| Lectura/escritura | Evento local       | Mensajes                                                                                                                       |
|-------------------|--------------------|--------------------------------------------------------------------------------------------------------------------------------|
| P0 lee B0         | Fallo de lectura   | MC_P0: Send (RMiss,B0,DIR_P0)<br>DIR_P0: Send(Fetch,B0,MC_P2)<br>MC_P2: Send(WB,B0,DIR_P0)<br>DIR_P0: Send(Reply,B0,MC_P0)     |
| P2 escribe B1     | Fallo de escritura | MC_P2: Send (WMiss,B1,DIR_P0)<br>DIR_P0: Send(Fetch+Inv,B1,MC_P0)<br>MC_P0: Send(WB,B1,DIR_P0)<br>DIR_P0: Send(Reply,B1,MC_P2) |

En algunos casos los mensajes son entre la memoria caché y el directorio de un mismo procesador, por lo que no llegan a salir a la red sino que se transfieren directamente a través de la memoria del nodo local. Pero el formato es exactamente el mismo en todos los casos, sea necesario salir a la red o no.

En el caso en el que P0 escribe B1 y tiene un acierto de escritura se puede observar que el mensaje con etiqueta WMiss que envía al directorio es sólo para anunciar que va a escribir en el bloque, en realidad no necesita que se le resuelva el fallo. Y de hecho el directorio no devuelve un mensaje de Reply con el bloque porque P0 aparece en el vector de compartidos y eso significa que ya tiene copia de este bloque. Por tanto, el directorio, simplemente, responde enviando una invalidación a todos los procesadores que tienen también copia de él, en este caso, P1.

Con este ejemplo se puede observar que este sencillo protocolo de tres estados resuelve el problema de la coherencia también en el caso de las arquitecturas de memoria compartida-distribuida.

La estructura de directorio que se ha explicado hasta ahora, a pesar de estar distribuida, puede presentar problemas de escalabilidad. Hay que tener en cuenta que el vector de compartidos de cada bloque necesita un bit por cada nodo del sistema (es lo que se llama un vector de compartidos completo), por lo que el directorio puede llegar a ocupar un espacio inadmisible en la memoria principal si la arquitectura se escala hasta un número determinado de nodos.

Para evitar este problema se han propuesto dos soluciones:

- **Vector de compartidos por grupos.** En este caso, cada bit del vector de compartidos representa a un grupo de nodos del sistema. Con que uno de los nodos de un grupo tenga copia de un bloque en su caché, la posición correspondiente del vector de compartidos en el directorio se pondrá a 1. Obviamente, el problema de esta implementación es que cuando se invalida un bloque, se debe enviar el mensaje de invalidación a todos los nodos que forman un grupo, ya que el directorio no puede saber cuáles de ellos tienen copia del bloque en su caché y cuáles no, sólo sabe que existe alguna caché de ese grupo de nodos que tiene copia y por lo tanto, está obligado a enviar la invalidación a todos ellos.
- **Directorio limitado.** Esta alternativa limita el número de cachés que pueden tener copia de un bloque simultáneamente. En este caso, lo que almacena el vector de compartidos es el identificador de los nodos que en cada momento tienen copia del bloque. Si llega una falla de lectura de un nodo por un determinado bloque y ya se ha llegado al límite de copias en el sistema para ese bloque, el directorio deberá decidir con algún algoritmo de selección qué copia debe invalidarse para poder enviar el bloque al nodo que acaba de tener el fallo sin sobrepasar así el número máximo de copias en caché permitidas en el sistema.

Por lo tanto, ambas alternativas resuelven el problema del espacio que ocupa el directorio en la memoria principal para sistemas con un gran número de nodos, pero sacrifican a cambio el ancho de banda de la red de interconexión de la arquitectura, ya que generan tráfico adicional (mensajes de invalidación extra) respecto del caso en el que se utiliza el vector de compartidos sencillo.

Por último, hay que señalar que los problemas de implementación que este tipo de protocolos presentan en la práctica, sea cual sea la estructura del directorio, están relacionados con el cumplimiento de la hipótesis de consistencia secuencial para la red de comunicaciones (y eso escapa a los objetivos de este libro) y con la existencia de más un nivel de caché. En este último caso la solución es la misma que para las arquitecturas de memoria compartida.

## 5.5 Diseño de arquitecturas de memoria distribuida

Como ya se ha mencionado con anterioridad, una arquitectura de memoria distribuida se compone de un conjunto de nodos de cómputo completos e independientes, conectados entre sí mediante una red de comunicaciones. Por eso este tipo de sistema se suele denominar multicomputador, como se ha comentado con anterioridad.

Cuanto más distribuidos y menos acoplados estén los recursos de cómputo que componen la arquitectura, más importancia cobran los sistemas operativos, el middleware, los modelos de programación y cualquier herramienta que contribuya a la integración de estos recursos. Por eso la comprensión en profundidad de este tipo de arquitecturas no se puede hacer desde el hardware exclusivamente.

Hoy en día se distinguen dos grandes tipos de arquitecturas de memoria distribuida, los clusters y los grids. Aunque existen otros paradigmas de computación distribuida como Peer2Peer o Cloud, que no están todavía muy extendidos más que en ciertas aplicaciones muy concretas.

### 5.5.1. Clusters

Un cluster es una arquitectura de memoria distribuida compuesta por un conjunto de nodos de cómputo independientes y conectados por una red controlada de altas prestaciones.

Este tipo de sistemas han aparecido de manera natural en empresas, universidades y centros de investigación, debido principalmente a:

- Los grandes avances en las prestaciones de los microprocesadores de consumo habitual.
- A la aparición de redes de comunicación y protocolos de alto rendimiento.
- Al desarrollo de herramientas software que facilitan su configuración y administración.

Se puede distinguir entre distintos tipos de clusters. Primero se pueden clasificar en Dedicados o No dedicados atendiendo al tipo de utilización que se haga de los recursos de cómputo que forman el cluster. En un cluster dedicado estos recursos están disponibles por completo el 100% del tiempo para la ejecución de las tareas que se lancen en el sistema. Sin embargo en un cluster no dedicado, algunos recursos estarán disponibles parcial o totalmente sólo en ciertos instantes de tiempo para el cluster. Por ejemplo, cuando se pretende aprovechar los ciclos de CPU no utilizados de las computadoras personales de los trabajadores de una organización, se puede estar ejecutando una tarea del cluster en el 100% de los recursos durante la noche, pero en cuanto uno de los trabajadores se siente delante de su computadora para trabajar, la cantidad de recursos disponible para el cluster disminuirá. Este tipo de arquitecturas se denominan en ocasiones NOW (Network Of Workstations), ya que en realidad no son más que redes de ordenadores que sólo en determinados momentos trabajan como una única arquitectura.

Atendiendo a las características de los nodos de cómputo que forman el cluster, encontramos sistemas homogéneos o heterogéneos. Los primeros clusters eran sistemas homogéneos en los que todos los nodos eran idénticos y la comunicación entre dos nodos cualesquiera del sistema suponía exactamente el mismo tiempo. En un sistema de este tipo puede aparecer heterogeneidad debida a dos factores:

- A las diferentes características hardware de las computadoras que componen el sistema (procesador, memoria, etc).
- A las diferentes características de las redes de comunicaciones que permiten la comunicación entre estos nodos en lo que se refiere a latencia y a ancho de banda.

La evolución natural de los clusters ha sido hacia sistemas heterogéneos. Por ejemplo, muchos clusters que se emplean actualmente en empresas e instituciones científicas no se han formado con equipos que se hayan adquirido específicamente para crear los clusters, sino que se ha aprovechado hardware disponible, que en la mayor parte de los casos es heterogéneo. También es típico que se adquieran equipos para diseñar clusters homogéneos pero que con el paso del tiempo se incorporen al sistema nuevos nodos o que se actualicen los nodos ya existentes. De nuevo en la mayor parte de los casos las ampliaciones o actualizaciones llevan a la aparición de heterogeneidad en el sistema.

Por lo tanto, la mayor parte de los clusters que existen en la actualidad son heterogéneos o terminarán siéndolo.

Por último, se puede distinguir entre clusters propietarios, diseñados por un fabricante concreto, o clusters Beowulf. En este caso, los nodos de proceso son PCs y utilizan un sistema operativo Linux. Muchas veces se considera un multicomputador casero que puede aprovechar componentes comerciales, baratos y potentes y recursos anticuados o infrautilizados que de otra manera no se utilizarían.

Sea cual sea el tipo de cluster diseñado, estos sistemas presentan una serie de ventajas si se comparan con las arquitecturas de memoria compartida o compartida-distribuida:

- Relación coste/prestaciones. Los clusters obtienen excelentes prestaciones utilizando hardware comercial cuya utilización está muy extendida, por lo que los precios de los componentes de la arquitectura suelen ser muy bajos.
- Escalabilidad. Este tipo de arquitecturas permite configurar sistemas con gran variedad de tamaños, desde unos pocos nodos hasta miles de ellos, sin que esto se refleje demasiado negativamente en sus prestaciones.
- Disponibilidad. Cuando se dispone del software adecuado, un fallo en un nodo o incluso en varios nodos del cluster no implica un fallo general de toda la arquitectura.
- Flexibilidad. Los clusters permiten realizar modificaciones en su configuración de manera muy sencilla, e incluso se pueden hacer estos cambios de manera dinámica, agregando o eliminando nodos del sistema en tiempo de ejecución.
- Aprovechamiento de los recursos. Esta flexibilidad lleva directamente a un mejor aprovechamiento de los recursos disponibles, ya que se pueden agregar nodos al cluster cuando sea necesario por su carga de trabajo, y liberarlos cuando estén infrautilizados.
- Crecimiento progresivo. Cuando el sistema se queda obsoleto o es necesario aumentar su capacidad de cómputo, basta con sustituir los nodos más antiguos por otros más potentes o con añadir nuevos nodos al sistema. Esta característica también se denomina en ocasiones escalabilidad incremental.
- Rápida respuesta a los avances tecnológicos. Esta posibilidad de crecimiento progresivo facilita la rápida incorporación de la tecnología más moderna al sistema.

Además, como no existe una memoria compartida ni un espacio de direcciones compartido, este tipo de arquitecturas no presentan los problemas de coherencia, consistencia y sincronización antes estudiados. Pero todavía existen algunas desventajas en la utilización de este tipo de sistemas.

Primero hay que tener en cuenta que la utilización de memoria distribuida, y por lo tanto, de comunicación explícita entre procesos, añade dificultad a la utilización eficiente de la red de comunicaciones de la arquitectura, que puede llegar a convertirse en el cuello de botella del sistema.

Además, en este tipo de arquitecturas existe todavía una carencia de estándares que dificulta la migración de cierto tipo de aplicaciones a un esquema de memoria distribuida. El único aspecto en el que existe un acuerdo global en toda la comunidad de usuarios de este tipo de sistemas es en la utilización de MPI (Message Passing Interface) como librería de paso de mensajes. Este es el único estándar (y no lo es de manera estricta, sólo *de facto*) que existe para este tipo de arquitecturas.

Por último, hay que analizar de manera algo más extensa los problemas de falta de imagen de sistema único y de equilibrio de carga así como las soluciones más extendidas para su resolución.

#### 5.5.1.1. SOLUCIONES PARA LA FALTA DE IMAGEN DE SISTEMA ÚNICO

Para solucionar el problema de la falta de imagen de sistema único es necesario instalar un middleware sobre el sistema operativo local de cada uno de los nodos que componen el cluster o parchear/modificar estos sistemas operativos locales para que funcionen como un sistema operativo distribuido (figura 5.20).

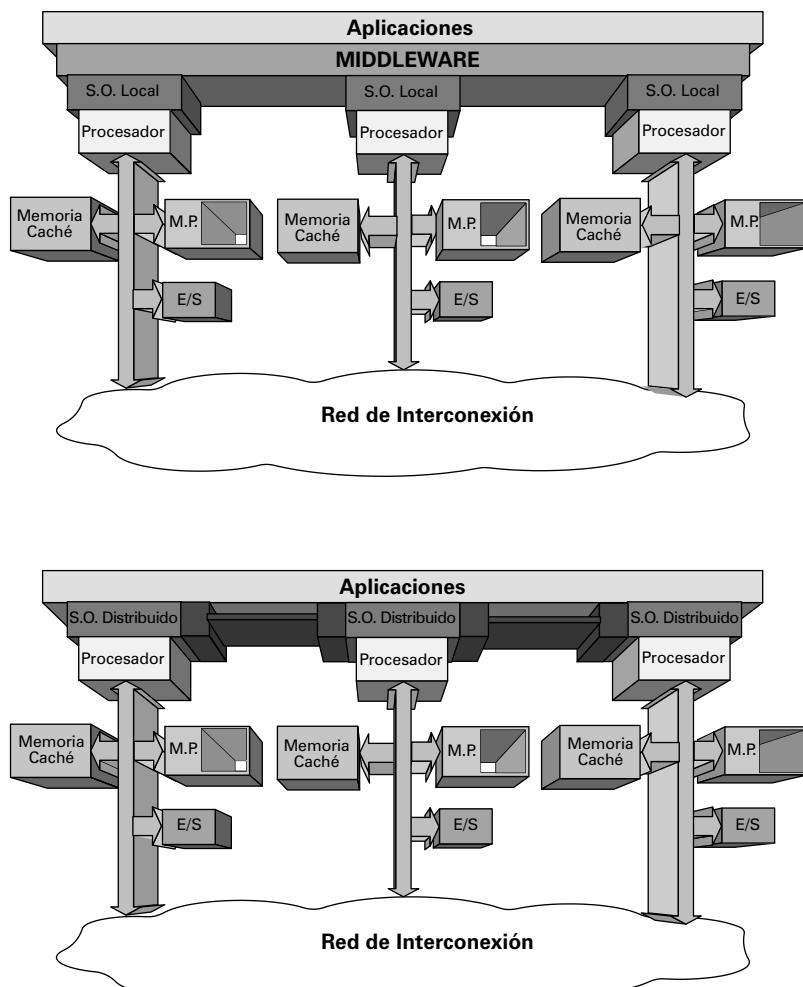


FIGURA 5.20

Soluciones para el problema de imagen de sistema único.

En cualquiera de los dos casos, la intención es proporcionar una capa intermedia entre los usuarios y sus aplicaciones, y la arquitectura, que proporcione los siguientes servicios:

- Espacio de procesos único. Para que los identificadores de proceso sean únicos en todo el sistema y esto facilite las comunicaciones entre procesos, la creación y terminación de procesos independientemente del nodo en el que se lance o ejecuten, etc
- Sistema de planificación de trabajos único. Para que el usuario pueda lanzar sus trabajos en el sistema sin preocuparse de en qué nodo o nodos se van a ejecutar realmente.
- Sistema de ficheros único. Para que el usuario perciba que existe una única jerarquía de ficheros bajo el mismo directorio raíz.
- Espacio de E/S único. Para que cualquier nodo del cluster pueda acceder a cualquier periférico o dispositivo de almacenamiento como si estuviera conectado localmente, incluso sin conocer su ubicación física.
- Interfaz de usuario único. De esta forma todos los usuarios del cluster tendrán un interfaz común de acceso sea cual sea el punto desde el que están accediendo al cluster.

Además, la imagen de sistema único se favorece si existe un punto de entrada único a la arquitectura y un punto de control y gestión del cluster también único.

De todas formas todavía no existe ninguna herramienta que permita, tanto al administrador como a los usuarios, ver al cluster como un sistema de cómputo único fuertemente acoplado.

#### **5.5.1.2. SOLUCIONES PARA EL EQUILIBRIO DE CARGA**

El equilibrio de la carga de trabajo entre los nodos que componen un cluster, es uno de los factores determinantes en el rendimiento del sistema. Para optimizar la utilización de los recursos disponibles hay que evitar situaciones en las que la llegada aleatoria de tareas al cluster provoque que unos nodos del sistema estén sobrecargados mientras que otros estén infrautilizados o incluso completamente desocupados, ya que estas situaciones implican graves penalizaciones en el tiempo de respuesta del sistema (que vendrá determinado por el tiempo de respuesta del nodo más lento).

De nuevo la solución para este problema no viene dada desde la propia arquitectura o hardware sino que involucra a niveles superiores. Los algoritmos de equilibrio de carga tienen como objetivo repartir las tareas que se ejecutan en el sistema de manera que cada nodo ejecute una carga proporcional a su capacidad de cómputo y se optimice la utilización de los recursos.

Este tipo de algoritmos pueden programarse a medida para un determinado sistema o incluso para una determinada combinación sistema-aplicación, aunque existen soluciones generales que en muchos casos ya se incorporan al middleware o sistema operativo mencionados en la sección anterior.

#### **5.5.2. Grids**

Un Grid está formado por un conjunto de recursos distribuidos geográficamente y conectados mediante una red.

Este tipo de arquitecturas son la extensión del concepto del cluster heterogéneo, permitiendo conectar todo tipo de recursos, que no tienen por qué estar ubicados en el mismo espacio geográfico, normalmente mediante una red de comunicaciones de área extensa (es decir, en un entorno mucho menos controlado y de menores prestaciones que en el caso de un cluster).

Las arquitecturas Grid permiten la creación de organizaciones virtuales (VO) mediante la compartición de recursos entre diferentes empresas, entidades y organismos (figura 5.21). Por lo tanto se trata de sistemas muy dinámicos cuya composición y configuración puede variar en cualquier momento.

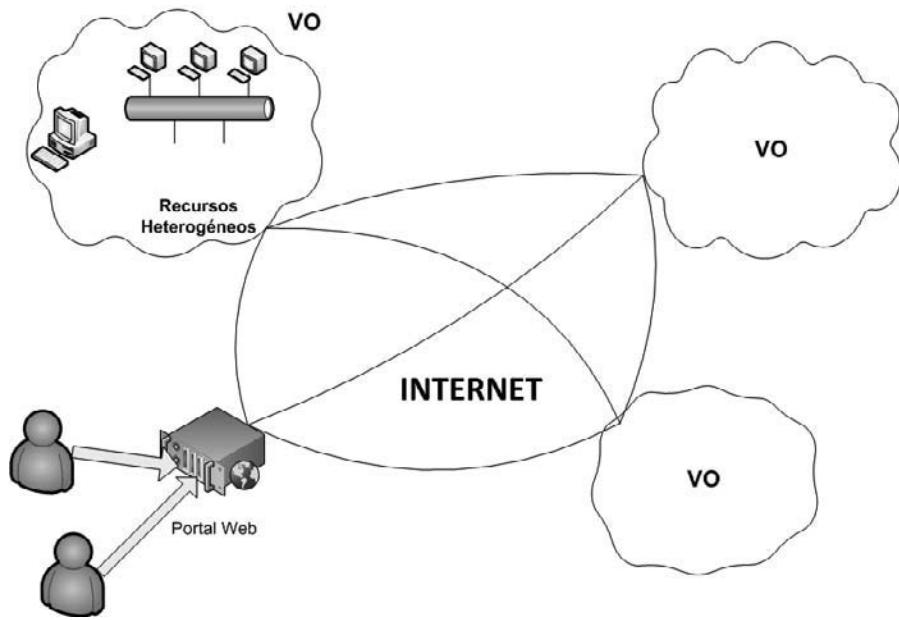


FIGURA 5.21

Arquitectura Grid típica.

Los problemas que plantean estos sistemas todavía no están resueltos del todo, por lo que se encuentran casi exclusivamente en entornos académicos, aunque parece que es el futuro de la computación distribuida o que llevará a nuevos tipos de arquitecturas que lo serán (como los sistemas Cloud).

Los principales problemas que plantean este tipo de arquitecturas están relacionados con los siguientes aspectos:

- Descubrimiento de servicios.
- Gestión de recursos.
- Comunicaciones.
- Seguridad.
- Tolerancia a fallos.
- Portabilidad de aplicaciones.

Cuando se habla de un Grid hoy en día, es casi inevitable hablar de Globus, herramienta que se ha convertido prácticamente en el estándar para tecnología Grid (de hecho, estándar *de facto*) ya que proporciona una infraestructura tecnológica robusta y muy completa para resolver todos estos problemas.

Este middleware ha sido programado por una comunidad de expertos y es de código abierto. Se trata de un conjunto de servicios (de hecho, actualmente se basa en Web Services) y librerías que dan soporte a los administradores y usuarios de arquitecturas Grid. No resuelven los problemas antes mencionados, pero sí que proporcionan la mayor parte de las herramientas necesarias para hacerlo.

## Resumen de decisiones de diseño de sistemas multiprocesador y multicomputador

### DISEÑO DE REDES DENTRO DE ARQUITECTURAS

| Decisión                                                | Alternativas                                               | Decisiones asociadas                                                                             |
|---------------------------------------------------------|------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| <b>Topología</b>                                        | Estática                                                   |                                                                                                  |
|                                                         | Dinámica                                                   | De medio compartido o conmutadas (monoetapa, multietapa bloqueantes o multietapa no bloqueantes) |
| <b>Tipo de técnica de conmutación</b>                   | Circuitos<br>Paquetes<br>Virtual cut-through<br>Vermiforme |                                                                                                  |
| <b>Tipo de decisión de encaminamiento</b>               | Inconsciente                                               | Aleatoria, rotatoria o determinista                                                              |
|                                                         | Adaptativa                                                 | Algoritmo de adaptación                                                                          |
| <b>Responsabilidad de la decisión de encaminamiento</b> | En origen                                                  | Fija o con capacidad de adaptación                                                               |
|                                                         | Decisión distribuida                                       |                                                                                                  |
|                                                         | Decisión centralizada                                      |                                                                                                  |
| <b>Implementación del encaminamiento</b>                | Tablas<br>Máquinas de estados                              |                                                                                                  |
| <b>Técnicas de control de flujo</b>                     |                                                            |                                                                                                  |

### DISEÑO DE ARQUITECTURAS DE MEMORIA COMPARTIDA

| Decisión                                                                       | Alternativas                  | Decisiones asociadas                                                               |
|--------------------------------------------------------------------------------|-------------------------------|------------------------------------------------------------------------------------|
| <b>Solución al problema de coherencia con protocolos de espionaje (snoopy)</b> | Invalidación<br>Actualización | Protocolo (estados, comandos y diagrama de transición de estados) e implementación |
| <b>Modelo de consistencia</b>                                                  | Secuencial<br>Relajado        |                                                                                    |
| <b>Solución al problema de sincronización</b>                                  | Cerrojos<br>Barreras<br>Otros |                                                                                    |

### DISEÑO DE ARQUITECTURAS DE MEMORIA COMPARTIDA-DISTRIBUIDA

| Decisión                                                                                       | Alternativas                                      | Decisiones asociadas                                                               |
|------------------------------------------------------------------------------------------------|---------------------------------------------------|------------------------------------------------------------------------------------|
| <b>Solución al problema de coherencia con protocolos de invalidación basados en directorio</b> | Directorio centralizado<br>Directorio distribuido | Protocolo (estados, comandos y diagrama de transición de estados) e implementación |

| Decisión                                                                                          | Alternativas                  | Decisiones asociadas |
|---------------------------------------------------------------------------------------------------|-------------------------------|----------------------|
| <b>Modelo de consistencia</b>                                                                     | Secuencial<br>Relajado        |                      |
| <b>Solución al problema de sincronización</b>                                                     | Cerrojos<br>Barreras<br>Otros |                      |
| <b>Sistema operativo o middleware que da soporte a la compartición del espacio de direcciones</b> |                               |                      |

#### DISEÑO DE ARQUITECTURAS DE MEMORIA DISTRIBUIDA

| Decisión                                                                                     | Alternativas                                                                                              | Decisiones asociadas |
|----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|----------------------|
| <b>Diseño hardware del sistema: tipo de nodos (procesador, memoria, disco, SO), red, etc</b> | Sistemas homogéneos/<br>heterogéneos<br>Sistemas propietarios/abiertos<br>Sistemas dedicados/no dedicados |                      |
| <b>Solución al problema de imagen de sistema único</b>                                       | Sistema operativo<br>Middleware                                                                           |                      |
| <b>Gestión de recursos, equilibrio de carga y planificación</b>                              |                                                                                                           |                      |
| <b>Tolerancia a fallos</b>                                                                   |                                                                                                           |                      |
| <b>Seguridad</b>                                                                             |                                                                                                           |                      |

## BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS

- ANDREWS, G. R. (1999): *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison Wesley.
- BUYYA, R. (1999): *High Performance Cluster Computing* (volúmenes 1 y 2), Prentice Hall.
- CULLER, D., SINGH, J. P. & GUPTA, A. (1998): *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann.
- DALLY, W. J. & TOWLES, B. P. (2004): *Principles and Practices of Interconnection Networks*, Morgan Kaufmann.
- DUATO, J.; YALAMANCHILI, S. & NI, L. (2002): *Interconnection Networks*, Morgan Kaufmann.
- EL-REWINI, H. & ABD-EL-BARR, M. (2005): *Advanced Computer Architecture and Parallel Processing*, Wiley.
- HENNESSY, J. L. & PATTERSON, D. A. (2007): *Computer Architecture: A Quantitative Approach* (4.<sup>a</sup> ed.), Morgan Kaufmann.
- HWANG, K. (1992): *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw Hill.
- MAGOULES, F.; PAN, J.; TAN, K. A. & KUMAR, A. (2009): *Introduction to Grid Computing*, CRC Press.
- ORTEGA, J.; ANGUITA, M. & PRIETO, A. (2005): *Arquitectura de Computadores*, Thomson.
- PFISTER, G. (1997): *In search of clusters* (2.<sup>a</sup> ed.), Prentice Hall.
- SHIVA, S. G. (2005): *Advanced Computer Architectures*, CRC Press.

## PROBLEMAS

- 5.1.** Una empresa que distribuye software para gestión empresarial comprueba que una función concreta de este software tarda 5 segundos en ejecutarse sobre un determinado procesador. Si el 68% de esta función puede paralelizarse en una arquitectura de memoria compartida con cuatro núcleos de este mismo procesador, ¿cuál es el tiempo de ejecución mínimo que puede conseguirse para esta función utilizando la arquitectura de memoria compartida?
- 5.2.** Una aplicación de cómputo científico tarda 10 segundos en leer sus operandos, 120 segundos en realizar operaciones matemáticas completamente paralelizables sobre una arquitectura de memoria compartida y 6 segundos en escribir sus resultados. ¿Cuál es la fracción serie de esta aplicación? ¿Cuál es el speedup máximo que podríamos conseguir paralelizando esta aplicación?
- 5.3.** Se diseña un procesador con un único nivel de memoria caché con tasa de fallos del 6% y tamaño de bloque de 256 palabras de 64 bits cada una. Este procesador funciona con una frecuencia de reloj de 1 GHz y realiza en media 1.25 accesos a memoria en cada ciclo de reloj. Se desea diseñar una arquitectura de memoria compartida con estos procesadores. ¿Cuál es el ancho de banda mínimo que debe proporcionar el bus compartido de esta arquitectura si se quieren incluir 4 procesadores?
- 5.4.** En el problema 5.3, ¿qué ocurre si se duplica el tamaño del bloque de la caché? ¿Cuál debería ser la tasa de fallos de la memoria caché de cada procesador para que se puedan conectar hasta 8 procesadores si se vuelve al tamaño de bloque del problema 5.3 y al ancho de banda obtenido en este problema?
- 5.5.** Se diseña una arquitectura de memoria compartida de 4 procesadores con cachés de post-escritura que utiliza un protocolo de coherencia de caché snoopy con invalidación de 4 estados:
- Inválido. Bloque inválido, no presente en la memoria caché.
  - Exclusivo. Única copia del sistema, coherente con memoria principal.
  - Compartido. Varias copias en el sistema, coherentes entre sí y con la memoria principal.
  - Modificado. Única copia válida del sistema, no coherente con la memoria principal.

Para el funcionamiento correcto de este protocolo se utiliza una señal S (Shared) en el bus compartido que sirve para distinguir los datos compartidos de los privados. A esta señal, que funciona mediante un OR cableado, están conectados todos los controladores de caché de manera que cada vez que observan un evento en el bus relacionado con un determinado bloque, vuelcan un 1 si tienen copia de dicho bloque y un 0 si no la tienen.

Los comandos de este protocolo son Miss e Inv, y las acciones con WB (volcado de bloque a memoria principal) e Invalidar (invalidación en memoria caché local).

- a) Deducir el diagrama de transición de estados que muestra la evolución de los bloques en una memoria caché del sistema según este protocolo.
- b) Mostrar la evolución de las memorias cachés implicadas, los eventos y acciones que se producen y los comandos que se envían por el bus cuando se produce la siguiente secuencia de lectura y escrituras.
  - P0 lee del bloque 0.
  - P3 lee del bloque 2.
  - P1 lee del bloque 0.
  - P2 escribe en el bloque 1.

- P0 escribe en el bloque 1.
- P2 lee el bloque 0.
- P3 escribe en el bloque 2.
- P0 escribe en el bloque 0.

Suponer que el estado inicial de las cachés es vacío y utilizar una tabla de este tipo:

| Evento | Comandos/Acciones | Estado de las cachés |    |    |
|--------|-------------------|----------------------|----|----|
|        | Pi:<br>Pj:        | Pi                   | Pj | Pk |
|        |                   |                      |    |    |

- 5.6. Repetir el problema 5.5 si los fallos de caché, en lugar de resolverlos siempre la memoria principal, los resuelven otras memorias caché siempre que sea posible.
- 5.7. En la siguiente tabla se detallan los tiempos que supone en cada una de las implementaciones de los dos problemas anteriores resolver un fallo desde la memoria principal ( $T_{falloMP}$ ), desde otra memoria caché ( $T_{falloMC}$ ), invalidar un bloque ( $T_{INV}$ ) y hacer un Writeback de un bloque a memoria principal ( $T_{WB}$ ). La implementación 1 corresponde al caso en el que los fallos se resuelven siempre desde la memoria principal y la implementación 2 al caso en el que se pueden resolver desde otra memoria caché siempre y cuando sea posible. Para el ejemplo concreto de código que se ha utilizado en ambos problemas, discutir acerca de la conveniencia de una y otra implementación. ¿Se pueden generalizar las conclusiones obtenidas?

|               | Implementación 1 | Implementación 2 |
|---------------|------------------|------------------|
| $T_{falloMP}$ | 100 ns           | 100 ns           |
| $T_{falloMC}$ | 140 ns           | 30 ns            |
| $T_{INV}$     | 10 ns            | 10 ns            |
| $T_{WB}$      | 60 ns            | 120 ns           |

- 5.8. Se diseña una arquitectura de memoria compartida de 4 procesadores con cachés híbridas de escritura directa y post-escritura que utiliza un protocolo de coherencia de caché snoopy con invalidación de 4 estados:

- Inválido. Bloque invalidado.
- Válido. Coherente con MP y con otras copias posibles en MC.
- Reservado. El bloque se ha escrito solamente una vez, y es coherente con MP. Es decir, cuando se escribe sólo una vez sobre un bloque, se utiliza escritura directa para actualizar la MP.
- Sucio. El bloque se ha escrito dos o más veces y es la única copia válida del sistema.

Los comandos de este protocolo son RMiss y WMiss, y las acciones con WB (volcado de bloque a memoria principal), ED (escritura directa de una única palabra a memoria principal) e Invali-dar (invalidación en memoria caché local).

- a) Deducir el diagrama de transición de estados que muestra la evolución de los bloques en una memoria caché del sistema según este protocolo.

- b) Mostrar la evolución de las memorias cachés implicadas, los eventos y acciones que se producen y los comandos que se envían por el bus cuando se produce la siguiente secuencia de lectura y escrituras:
- P0 lee el bloque 3.
  - P1 escribe en el bloque 3.
  - P2 escribe en el bloque 4.
  - P1 escribe en el bloque 3.
  - P2 escribe en el bloque 3.
  - P3 lee el bloque 4.

Suponer que el estado inicial de las cachés es el siguiente:

P0: Vacío.

P1: Bloque 3 válido, bloque 4 reservado.

P2: Bloque 3 válido.

P3: Vacío.

Utilizar una tabla de este tipo:

| Evento | Comandos/Acciones | Estado de las cachés |    |    |
|--------|-------------------|----------------------|----|----|
|        |                   | Pi                   | Pj | Pk |
|        | Pi:<br>Pj:        |                      |    |    |
|        |                   |                      |    |    |

- 5.9. Se diseña una arquitectura de memoria compartida de 4 procesadores con cachés de post-escritura que utiliza un protocolo de coherencia de caché snoopy con actualización de 4 estados:

- Estado inicial o uncached (U). El bloque no está en la caché, por lo que cuando el procesador lo solicita para leer o para escribir se produce un fallo.
- Exclusivo (E). El bloque está en la caché y no ha sido modificado mediante escrituras locales. Es la única copia en el sistema.
- Válido (V). El bloque está en la caché y no ha sido modificado mediante escrituras locales. Existen otras copias en las cachés de otros procesadores, todas ellas coherentes entre sí.
- Modificado (M). El bloque está en la caché y no es coherente con la memoria principal porque el procesador local ha realizado al menos una escritura sobre él. Puede que existan otras copias en las cachés de otros procesadores, todas ellas coherentes entre sí.

Para el funcionamiento correcto de este protocolo se utiliza una señal S (Shared) en el bus compartido que sirve para distinguir los datos compartidos de los privados. A esta señal, que funciona mediante un OR cableado, están conectados todos los controladores de caché de manera que cada vez que observan un evento en el bus relacionado con un determinado bloque, vuelcan un 1 si tienen copia de dicho bloque y un 0 si no la tienen.

Los comandos de este protocolo son RMiss, WMiss y Upd, y las acciones con WB (volcado de bloque a memoria principal) y Actualizar\_MC (actualización local con los contenidos de un comando Upd).

- a) Deducir el diagrama de transición de estados que muestra la evolución de los bloques en una memoria caché del sistema según este protocolo.
- b) Mostrar la evolución de las memorias cachés implicadas, los eventos y acciones que se producen y los comandos que se envían por el bus cuando se produce la misma secuencia de

lecturas y escrituras que en el problema 5.5. Suponer también que el estado inicial de las cachés es vacío y utilizar el mismo tipo de tabla y nomenclatura.

- c) Comparar para este ejemplo el rendimiento de los protocolos de invalidación y actualización. ¿Se pueden generalizar las conclusiones extraídas de esta comparación?
- 5.10.** Un sistema de memoria compartida-distribuida con 4 procesadores y cachés privadas asociadas a cada procesador, utiliza un protocolo de directorio de invalidación para garantizar la coherencia de las cachés. Este protocolo se basa en la utilización de tres estados para el directorio y para las memorias caché:
- Inválido (I). Bloque invalidado. Este estado en el directorio se denomina No Cacheado (NC), es decir, cuando no hay ninguna caché en el sistema con copia del bloque.
  - Compartido (C). Una o varias copias del bloque coherentes entre sí y con la memoria principal.
  - Exclusiva (E). Una única copia en el sistema, no coherente con memoria principal.

Las cachés poseen el siguiente estado inicial:

P0: Vacío.

P1: Bloque 3 compartido, bloque 1 exclusivo.

P2: Bloques 0 y 3 compartidos.

P3: Vacío.

Además se pueden enviar mensajes con el formato (etiqueta, bloque, destino), siendo las etiquetas posibles RMiss (fallo de lectura), WHit (acierto de escritura), WMiss (fallo de escritura), WB (volcado de bloque u orden de volcado de bloque) e Inv (invalidación). El campo bloque indica qué bloque está implicado en la operación y el campo destino indica a qué procesador se le envía el mensaje.

Si los bloques 0, 1, 2 y 3 se encuentran en la memoria principal del procesador P0, su directorio en la situación inicial es:

|    |      |    |
|----|------|----|
| B0 | 0010 | C  |
| B1 | 0100 | E  |
| B2 | 0000 | NC |
| B3 | 0110 | C  |

Mostrar en una tabla como la siguiente la evolución de las cachés y del directorio de P0, y todos los mensajes que se envían, con las siguientes lecturas y escrituras:

- P3 lee del bloque 3.
- P1 lee del bloque 1.
- P2 escribe en el bloque 3.
- P3 lee del bloque 1.
- P1 escribe en el bloque 3.

| Evento | Mensajes | Directorio | Estado de las cachés |    |    |
|--------|----------|------------|----------------------|----|----|
|        |          |            | Pi                   | Pj | Pk |
|        |          |            |                      |    |    |

- 5.11.** Se diseña una arquitectura de memoria compartida-distribuida de 8 procesadores que utiliza un protocolo de coherencia de cachés basado en directorio. Este protocolos utiliza los siguientes 4 estados para los bloques en las memorias caché y en el directorio:

- Inválido. Bloque inválido.
- Exclusivo. Única copia del sistema, coherente con memoria principal.
- Compartido. Varias copias en el sistema, coherentes entre sí y con la memoria principal.
- Modificado. Única copia válida del sistema, no coherente con la memoria principal.

Los mensajes que se pueden enviar con este protocolo tienen el formato (etiqueta, bloque, destino). El campo bloque indica qué bloque está implicado en la operación y el campo destino indica a qué procesador se le envía el mensaje.

- a) Utilizando el conjunto de etiquetas propuesto en este libro para los protocolos basados en directorio distribuido como punto de partida, definir el conjunto de etiquetas necesario para implementar este protocolo.
- b) Deducir el diagrama de transición de estados que muestra la evolución de los bloques en el directorio del sistema según este protocolo.
- c) Mostrar la evolución del directorio y de las cachés con la secuencia de lecturas y escrituras del problema 5.5. Suponer que el propietario de todos los bloques involucrados en las operaciones de lectura y escritura es P0 y que las cachés están vacías inicialmente.

| Evento | Mensajes | Directorio | Estado de las cachés |    |    |
|--------|----------|------------|----------------------|----|----|
|        |          |            | Pi                   | Pj | Pk |
|        |          |            |                      |    |    |

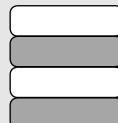
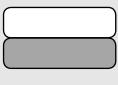
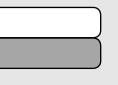
- 5.12.** Se diseña una arquitectura de memoria compartida-distribuida de 8 procesadores que utiliza un protocolo de coherencia de cachés basado en directorio. Este protocolos utiliza los siguientes 4 estados:

- Inválido. Bloque inválido.
- Exclusivo. Única copia del sistema, coherente con memoria principal.
- Compartido. Varias copias en el sistema, coherentes entre sí y con la memoria principal.
- Modificado. Única copia válida del sistema, no coherente con la memoria principal.

Los mensajes que se pueden enviar con este protocolo tienen el formato (etiqueta, bloque, destino). El campo bloque indica qué bloque está implicado en la operación y el campo destino indica a qué procesador se le envía el mensaje.

- a) Definir el conjunto de etiquetas posibles para este protocolo si el directorio está centralizado.
- b) Si el directorio se almacena en la memoria del procesador P7, mostrar la evolución del directorio y de las cachés con la secuencia de lecturas y escrituras del problema 5.5. Suponer

que el propietario de todos los bloques involucrados en las operaciones de lectura y escritura es P0 y que las cachés están vacías inicialmente.

| Evento | Mensajes | Directorio | Estado de las cachés                                                              |                                                                                     |                                                                                     |
|--------|----------|------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|        |          |            | Pi                                                                                | Pj                                                                                  | Pk                                                                                  |
|        |          |            |  |  |  |

## AUTOEVALUACIÓN

1. ¿Qué diferencias hay entre las arquitecturas de memoria compartida y las de memoria compartida-distribuida?
2. ¿Qué significa que en las arquitecturas de memoria compartida la comunicación entre procesos es implícita?
3. ¿Cuál es la principal característica de una red crossbar?
4. ¿En qué consiste la conmutación virtual cut-through y cuáles son sus ventajas?
5. ¿Cómo se resuelve el problema de la coherencia en arquitecturas de memoria compartida?
6. ¿Qué diferencias hay entre los protocolos de coherencia basados en invalidación y basados en actualización?
7. ¿En qué consiste el problema de la consistencia y qué alternativas hay para resolverlo?
8. Cuando se estudian protocolos de coherencia, ¿qué es un directorio y para qué se utiliza?
9. ¿Qué diferencias hay entre un cluster y un grid?
10. ¿En qué consiste la falta de imagen de sistema único y cómo se puede resolver este problema?

# 6

---

# Evaluación de prestaciones

---

## Contenidos

- 6.1. Definición de métricas de rendimiento
- 6.2. Métricas de rendimiento sencillas
- 6.3. Evaluación y comparación de rendimiento
- 6.4. Técnicas de medida y benchmarks
- 6.5. Métricas de rendimiento complejas

La necesidad de definir métricas de rendimiento surge en multitud de contextos relacionados con la arquitectura de computadoras como comparar diferentes alternativas de diseño, predecir el impacto que va a tener una mejora, encontrar el conjunto de parámetros que hace que un diseño consiga su rendimiento máximo o establecer las alternativas de mejora de una arquitectura.

De hecho a lo largo de todo este libro ya se han empleado métricas y expresiones relacionadas con la evaluación de prestaciones para tomar decisiones de diseño, comparar alternativas y cuantificar la ganancia obtenida con ciertas mejoras.

En este capítulo se resumen los aspectos básicos de la evaluación de rendimiento de arquitecturas de computadoras. Por tanto, se proporcionan unas guías y recomendaciones para escoger métricas de rendimiento adecuadas y se definen métricas sencillas que se puedan emplear en contextos generales de evaluación. También se estudia cómo realizar la interpretación de estos resultados y cómo comparar unos con otros, introduciendo las leyes de Amdhal y Gustafson y su utilización en diferentes escenarios.

Además, se presentan las diferentes técnicas de medida que pueden emplearse para obtener los valores de las métricas escogidas en sistemas reales y se estudian las diferentes aplicaciones que se pueden ejecutar en un sistema para realizar estas medidas de rendimiento, proporcionando así una introducción al concepto de benchmark.

Por último se discuten métricas de rendimiento más sofisticadas que permitan la evaluación de arquitecturas compuestas por más un procesador como las estudiadas en el capítulo 5 de este libro. En esta última parte del capítulo se estudian las diferentes definiciones de speedup y métricas como la eficiencia o la escalabilidad.

## 6.1 Definición de métricas de rendimiento

Hasta este momento, en este libro se han estudiado el funcionamiento y el diseño de los principales componentes que se encuentran en una arquitectura monoprocesador y de las arquitecturas multiprocesador y multicamputador. Sin embargo, como se ha mencionado en la introducción de este capítulo, y como se anunció en el capítulo 1, no es suficiente con adquirir estos conocimientos, también es necesario saber cómo van a evolucionar estas arquitecturas, cómo se pueden mejorar, cuánto mejor son unas que otras, etc.

En todos estos contextos en los que puede ser necesario llevar a cabo la evaluación de rendimiento de una arquitectura, existen ciertas características que siempre es deseable que la métrica cumpla y que permitirán que la evaluación de rendimiento de una arquitectura sea más completa y fiable. Las más importantes son:

- **Facilidad de medida.** Es deseable que la métrica de rendimiento sea fácil de medir para que sea ampliamente utilizada y así existan resultados con los que comparar el obtenido. Además, la facilidad de medida hará más difícil que se cometan errores en la evaluación de rendimiento.
- **Repetibilidad.** Una métrica de rendimiento es repetible si siempre que se mide sobre la misma arquitectura en las mismas condiciones se obtiene el mismo valor (exceptuando, obviamente, la variabilidad típica que se produce en la experimentación). Es decir, se trata de una métrica determinista.
- **Fiabilidad.** Una métrica de rendimiento es fiable si el valor obtenido para la arquitectura A es mayor que para la arquitectura B y el rendimiento de la arquitectura A en el aspecto que cuantifica la métrica, es siempre mayor para esta arquitectura.
- **Consistencia.** Una métrica es consistente si su definición, su significado y sus unidades de medida no varían de unas arquitecturas a otras.
- **Linealidad.** Una métrica es lineal si su relación con el aspecto de rendimiento que cuantifica es lineal, es decir, si por ejemplo al duplicarse el rendimiento se duplica el valor de la métrica. Este tipo

de relación hace mucho más fácil la interpretación de los resultados de las medidas. Si al hacer una mejora en el diseño de un procesador su rendimiento se duplica pero el valor de la métrica pasa de 0.8 a 0.9, probablemente no interpretemos bien los resultados de la evaluación de rendimiento, sin embargo si pasa de valer 2 a valer 4, es mucho más sencillo interpretarlos correctamente.

Por supuesto, también es deseable que el coste de las medidas sea bajo, que la métrica no dependa de intereses comerciales, que esté estandarizada, etc.

Pero obviamente es muy complicado conseguir una métrica de rendimiento que cumpla todas estas características y aún así siga siendo fácil de medir, desgraciadamente la mayor parte de las veces hay que sacrificar alguna de estas propiedades al escoger las métricas que se van a emplear en una evaluación de rendimiento.

## 6.2 Métricas de rendimiento sencillas

Habitualmente se utilizan como métricas de rendimiento aspectos que no cumplen las características mencionadas en la sección anterior, como puede ser la frecuencia de reloj de un procesador, o los MIPS (millones de instrucciones por segundo) o MFLOPS (millones de instrucciones en coma flotante por segundo) que puede ejecutar.

$$\text{MIPS} = \frac{I}{t_{CPU} \cdot 10^6} = \frac{1}{CPI \cdot T \cdot 10^6}$$

$$\text{MFLOPS} = \frac{\text{Operaciones en FP}}{t_{CPU} \cdot 10^6}$$

Tal y como se ha estudiado en el capítulo 1 de este libro, el tiempo que un determinado código tarda en ejecutarse en un procesador es inversamente proporcional a su frecuencia de funcionamiento. Pero, según la ecuación de prestaciones del procesador, también depende del CPI de la arquitectura y del número de instrucciones del código. Es decir, un procesador A con una frecuencia de reloj mayor que otro procesador B no necesariamente ejecutará un código más rápido que B. Por lo tanto, la frecuencia de reloj como métrica de rendimiento no cumple con una de las características más importantes de las enumeradas en la sección anterior: la fiabilidad. Y además tampoco es lineal. Sólo se puede utilizar para realizar una buena evaluación de rendimiento si se están comparando procesadores con exactamente el mismo diseño (mismo repertorio de instrucciones, mismo CPI) y que sólo varían en la frecuencia de reloj.

Pero su facilidad de medida, repetibilidad y consistencia hace que se use en muchos casos como métrica de rendimiento general, también en comparaciones de diseños diferentes, de ahí que algunos fabricantes de procesadores intenten utilizar técnicas de diseño que aumenten al máximo la frecuencia de reloj (por ejemplo, la hipersegmentación, que permite que el procesador se segmente con un alto número de etapas que puede alcanzar fácilmente el valor de 30) a costa de aumentar también el CPI, sabiendo que en muchos casos se identificará una frecuencia de reloj alta con un rendimiento también alto.

Más o menos lo mismo ocurre con los MIPS y los MFLOPS, utilizados en muchos casos como métrica de rendimiento del procesador a pesar de ser métricas no fiables, no lineales y además, inconsistentes. Hay que tener en cuenta que dos procesadores con repertorios de instrucciones diferentes pueden llegar a realizar cantidades de trabajo muy diferentes con una única instrucción (por ejemplo, si se compara un repertorio CISC con uno RISC), por lo que de nuevo un valor mayor para los MIPS o los MFLOPS no implica necesariamente un rendimiento mayor.

Lo mismo ocurre con métricas típicas para la memoria y el sistema de E/S (la latencia y el ancho de banda), que se pueden utilizar cuando se comparan alternativas con exactamente el mismo diseño, pero que no son fiables en el resto de los casos, cuando se comparan sistemas de muy diferentes características.

Entonces, ¿qué métricas sencillas y generales se pueden utilizar para evaluar el rendimiento de una arquitectura de manera fiable? En la tabla 6.1 se muestra un resumen de las que se han utilizado a lo largo de este libro para evaluar el rendimiento de procesadores, jerarquías de memoria y sistemas de E/S.

Existen otras métricas de rendimiento sencillas, menos generales, pero que también pueden ser fiables en muchos casos. Es el caso del consumo de potencia o del tiempo de ejecución (también denominado tiempo de respuesta), que no sólo tiene en cuenta el tiempo de CPU de una determinada tarea sino todo el tiempo que transcurre desde que ésta comienza su ejecución hasta que finaliza. Es decir, este tiempo incluye los accesos a memoria y las transacciones de E/S, las esperas por los recursos en el caso de sistemas operativos multitarea, el tiempo que el sistema operativo necesita para realizar sus tareas, etc. En ocasiones, desde el punto de vista del administrador, se emplea como métrica la productividad del sistema, inversamente relacionada con los tiempos de ejecución.

**TABLA 6.1**  
Métricas de rendimiento sencillas y fiables.

| Componente evaluado  | Métrica                          | Significado                                                                                 | Expresión                                                                         |
|----------------------|----------------------------------|---------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| Procesador           | Tiempo de CPU                    | Tiempo que un código con $I$ instrucciones ensamblador tarda en ejecutarse en el procesador | $t_{CPU} = I \cdot CPI \cdot T$                                                   |
| Jerarquía de memoria | Tiempo medio de acceso a memoria | Tiempo que el procesador tarda en media en realizar un acceso a memoria                     | $t_{MEM} = T_{accesoMC} + TF \cdot pF$                                            |
| Bus                  | Ancho de banda                   | Cantidad de información que puede transferir el bus por unidad de tiempo (normalmente B/s)  | $BW = \text{ancho de datos} \cdot f \cdot n.\text{o de transferencias por ciclo}$ |

En el caso de la evaluación de los dispositivos de E/S, las métricas suelen ser particulares para cada tipo de dispositivo, a continuación se dan algunos ejemplos:

- Dispositivos de almacenamiento. Capacidad, rpm (revoluciones por minuto), tiempos de búsqueda, tiempos de lectura y escritura, velocidad de grabación. Si se trata de un RAID, además los valores del MTTF y del MTTDL que se estudiaron en el capítulo 4.
- Monitores. Resolución, tamaño de la diagonal, contraste.
- Dispositivos de impresión. Resolución, número de páginas por minuto.
- Altavoces. Potencia nominal, impedancia.

## 6.3 Evaluación y comparación de rendimiento

La evaluación de rendimiento de un sistema no suele tener significado por sí misma, en casi todos los casos adquiere este significado comparando los valores obtenidos para las métricas escogidas con otros anteriores del mismo sistema o con los obtenidos para otros sistemas.

Por eso es muy habitual utilizar cocientes para comparar el rendimiento de dos sistemas A y B o de un mismo sistema con dos diferentes configuraciones A y B, de manera que si M es la métrica de rendimiento escogida, y cuanto mayor sea su valor, mayor es el rendimiento, este cociente es:

$$\frac{M_B}{M_A}$$

Y si este cociente es igual a 1, el rendimiento de las dos alternativas es el mismo, si es mayor que 1, el rendimiento de la alternativa B es mayor que el de la A, y si el cociente es menor que 1, el rendimiento de B es menor que el de A. Además, este cociente indica cuánto mayor o menor es el rendimiento en cada caso.

En el caso de utilizar métricas relacionadas con el tiempo de CPU o de ejecución, es al contrario, ya que cuanto mayor es este tiempo peor es el rendimiento. Y en muchos casos en lugar de utilizar medidas individuales o aisladas, se utilizan medias aritméticas o ponderadas.

### Ejemplo 6.1

#### Comparación de rendimiento entre diferentes configuraciones de una misma computadora.

Se desea evaluar el rendimiento de tres configuraciones diferentes de una misma computadora: C1, C2 y C3. Para ello se descargan tres aplicaciones de prueba de una conocida página de internet dedicada a la evaluación de rendimiento: PA (intensiva en uso del procesador), PB (intensiva en uso memoria) y PC (intensiva en E/S). Los tiempos de ejecución que se miden para estas tres aplicaciones son, en segundos:

|                                             | C1    | C2    | C3    |
|---------------------------------------------|-------|-------|-------|
| PA                                          | 100   | 110   | 130   |
| PB                                          | 50    | 45    | 40    |
| PC                                          | 22    | 15    | 12    |
| Media aritmética para las tres aplicaciones | 57.33 | 56.67 | 60.67 |

Con estos resultados se puede comparar el rendimiento de las tres configuraciones estudiadas con frases del tipo "La configuración 1 es N veces más rápida que la configuración 2". Por ejemplo, la configuración C1 es 1.3 veces más rápida (o un 30% más rápida) que la configuración C3 en la ejecución de la aplicación intensiva en uso del procesador PA (comparando los 100 segundos de C1 con los 130 de C3). Sin embargo, la configuración C3 es 1.25 veces más rápida (o un 25% más rápida) que la configuración C1 en la ejecución de la aplicación intensiva en uso de memoria PB (comparando los 50 segundos de C1 con los 40 de C3).

Pero si con estos resultados tuviéramos que escoger una configuración que obtuviera el mejor rendimiento global, podemos utilizar las medias aritméticas de los tres tiempos. Así sabemos que la configuración C2 es la mejor, un 1.01 más rápida que la C1 y un 1.07 más rápida que la C3.

Si nos damos cuenta de que el 90% del tiempo el sistema está ejecutando aplicaciones que hacen un uso intensivo del procesador, el 8% del tiempo de la memoria y el 2% restante de E/S, podríamos utilizar esta información para calcular una media ponderada de los tiempos (en lugar de aritmética) y escoger así la mejor configuración posible dando más peso a la que obtiene un mejor rendimiento en las aplicaciones cuya ejecución es más frecuente.

|                                                                  | C1    | C2    | C3     |
|------------------------------------------------------------------|-------|-------|--------|
| PA                                                               | 100   | 110   | 130    |
| PB                                                               | 50    | 45    | 40     |
| PC                                                               | 22    | 15    | 12     |
| <b>Media aritmética para las tres aplicaciones</b>               | 57.33 | 56.67 | 60.67  |
| <b>Media ponderada<br/>(0.9 · tPA + 0.08 · tPB + 0.02 · tPC)</b> | 94.44 | 102.9 | 120.44 |

Con esta información acerca de la frecuencia de ejecución de las aplicaciones escogeríamos la configuración C1, que es 1.09 veces más rápida que la C2 y 1.28 veces más rápida que la C3.

Este método para escoger la mejor configuración utilizando la media ponderada es mucho más fiable que el anterior, el problema es que no siempre se tiene información acerca de las frecuencias de ejecución de las aplicaciones para ponderar la media, por lo que se tiene que recurrir a la aritmética, que supone que todas las aplicaciones se ejecutan con la misma frecuencia.

La ganancia, aceleración o speedup que se puede obtener realizando una modificación a una arquitectura está muy relacionada con este tipo de cocientes, sólo hay que tener en cuenta que se define como un cociente de tiempos de CPU o de ejecución y cuanto mayores son estos tiempos, peor es el rendimiento.

Como ya se estudió en el capítulo 1, se define el speedup como:

$$S = \frac{t_{sin\ mejor} }{t_{con\ mejor}}$$

De manera que hay tres alternativas:

- S=1: La modificación no afecta al rendimiento de la arquitectura.
- S<1: La modificación ha empeorado el rendimiento de la arquitectura.
- S>1: La modificación ha mejorado realmente el rendimiento de la arquitectura.

### Ejemplo 6.2

#### Cálculo del speedup.

Un diseñador de procesadores debe decidir si aumentar el número de etapas de segmentación de su nuevo diseño. Para tomar esta decisión evalúa que aumentar de 4 a 6 etapas le permite aumentar la frecuencia de reloj del procesador en un 15%, pero que el CPI pasa de valer 1 a valer 1.23.

Por lo tanto, para saber si aumentar el número de etapas supone una mejora o no, calcula el speedup que obtendría mediante la ecuación de prestaciones del procesador:

$$\text{Procesador con 4 etapas} \left\{ \begin{array}{l} I \\ CPI = 1 \\ f \end{array} \right.$$

$$\text{Procesador con 6 etapas} \left\{ \begin{array}{l} I \\ CPI = 1.23 \\ 1.15 f \end{array} \right.$$

Ya que el número de instrucciones que componen un código no cambia porque no se modifica el repertorio de instrucciones. Por tanto:

$$S = \frac{t_{4 \text{ etapas}}}{t_{6 \text{ etapas}}} = \frac{(I \cdot CPI / f)_{4 \text{ etapas}}}{(I \cdot CPI / f)_{6 \text{ etapas}}} = \frac{1/f}{1.23/1.15f} = 0.93 < 1$$

Por lo que aumentar el número de etapas de segmentación del procesador no supondría una mejora.

### Ejemplo 6.3

#### Análisis coste/prestaciones.

El mismo diseñador que el del ejemplo 6.2 se plantea modificar el diseño del procesador incluyendo un divisor en coma flotante, ya que hasta el momento las divisiones se emulaban por software.

Esta modificación implica disminuir el número de instrucciones que se ejecutan en un 10% (porque se incluye en el repertorio una instrucción específica para realizar las divisiones) y aumentar el periodo de reloj del procesador de 0.3 a 0.32 ns, por tanto:

$$\begin{aligned} \text{Procesador sin divisor en FP} &\left\{ \begin{array}{l} I \\ CPI \\ T = 0.3 \text{ ns} \end{array} \right. \\ \text{Procesador con divisor en FP} &\left\{ \begin{array}{l} 0.9 \cdot I \\ CPI \\ T = 0.32 \text{ ns} \end{array} \right. \end{aligned}$$

Ya que el CPI empeoraba ligeramente (las divisiones en coma flotante son poco frecuentes), pero se ha conseguido evitar con alguna pequeña modificación. Por tanto:

$$S = \frac{t_{\text{sin divisor}}}{t_{\text{con divisor}}} = \frac{(I \cdot CPI \cdot T) \text{ sin divisor}}{(I \cdot CPI \cdot T) \text{ con divisor}} = \frac{I \cdot 0.3}{0.9 \cdot I \cdot 0.32} = 1.04 > 1$$

Tras comprobar que la modificación evaluada es efectivamente una mejora, el diseñador se pregunta cuánto le compensa incrementar el coste del diseño del procesador para conseguir esta mejora en el rendimiento.

Como  $S=1.04$ , es decir, la mejora es de un 4%, como mucho convendría incrementar el coste del diseño en un 4%. Si el incremento del coste es superior, no compensaría realizar la mejora.

### 6.3.1. Ley de Amdhal y Ley de Gustafson

La ley de Amdhal permite calcular la ganancia obtenida para una tarea (speedup) en un determinado sistema con una modificación concreta. Esta ganancia está siempre limitada por la fracción de tiempo que puede utilizarse la mejora realizada en el sistema (si es que finalmente la modificación resulta ser una mejora).

Según esta ley, el speedup depende siempre de dos factores. El primero es la fracción del tiempo de ejecución que se puede utilizar la mejora ( $F$ ) y el segundo es la ganancia que se introduce con la mejora

durante esta fracción de tiempo ( $G$ ). Hay que tener en cuenta que se puede calcular el tiempo de ejecución de la tarea con la mejora a partir de estas dos magnitudes y del tiempo de ejecución antiguo, lo que permite obtener el speedup según la ley de Amdhal.

$$t_{\text{antes de la mejora}} = t$$

$$t_{\text{después de la mejora}} = (1 - F) \cdot t + \frac{F \cdot t}{G}$$

$$S = \frac{t_{\text{antes de la mejora}}}{t_{\text{después de la mejora}}} = \frac{1}{(1 - F) + \frac{F}{G}}$$

#### Ejemplo 6.4

##### Aplicación de la ley de Amdhal.

En un determinado procesador las instrucciones enteras se ejecutan en 1 ciclo de reloj, mientras que las de coma flotante necesitan 5 ciclos de reloj para completarse.

En media, las aplicaciones que se ejecutan en este procesador invierten un 30% del tiempo en ejecutar operaciones en coma flotante. Y nos gustaría saber si desde el punto de vista coste-prestaciones sería interesante rediseñar el procesador para que las operaciones de coma flotante fueran 10 veces más rápidas a costa de duplicar el coste total del procesador.

Para tomar esta decisión basta con saber si el speedup que se obtiene realizando esta modificación en el diseño del procesador es de 2, ya que así la ganancia obtenida con la mejora sería comparable a la inversión realizada.

Si se utiliza la ley de Amdhal, tenemos que  $F=0.3$  (la mejora puede utilizarse un 30% del tiempo) y  $G=10$  (la ganancia en la parte mejorada, en este caso la ejecución de las instrucciones en coma flotante, es 10) por lo que:

$$S = \frac{t_{\text{antes de la mejora}}}{t_{\text{después de la mejora}}} = \frac{1}{(1 - F) + \frac{F}{G}} = \frac{1}{(1 - 0.3) + \frac{0.3}{10}} = 1.37 < 2$$

Por lo que se mejoraría el rendimiento global del procesador en un 37%, y este es, más o menos, el incremento en el coste del diseño que se considera aceptable, en principio sólo por esta mejora no compensa duplicar su coste.

#### Ejemplo 6.5

##### Aplicación de la ley de Amdhal con más de una mejora.

Nos estamos planteando realizar dos modificaciones en el diseño de un procesador:

- Mejorar la ALU de enteros de manera que el CPI de las instrucciones aritmético-lógicas con enteros pase de 1 a 0.8.
- Mejorar el coprocesador en coma flotante para que las instrucciones en coma flotante se ejecuten al doble de velocidad.

Si este procesador ejecuta el 45% del tiempo instrucciones aritmético-lógicas con enteros y el 10% del tiempo instrucciones en coma flotante, el speedup que se podría obtener con cada una de estas mejoras es:

$$\text{Mejora para instrucciones enteras} \begin{cases} F = 0.45 \\ G = \frac{CPI_{ant}}{CPI_{nuevo}} = \frac{1}{0.8} = 1.25 \end{cases}$$

Porque no cambian ni el número de instrucciones ni la frecuencia de reloj. Entonces:

$$S = \frac{t_{\text{antes de la mejora}}}{t_{\text{después de la mejora}}} = \frac{1}{(1-F) + \frac{F}{G}} = \frac{1}{(1-0.45) + \frac{0.45}{1.25}} = 1.1$$

$$\text{Mejora para instrucciones FP} \begin{cases} F = 0.1 \\ G = 2 \end{cases}$$

$$S = \frac{t_{\text{antes de la mejora}}}{t_{\text{después de la mejora}}} = \frac{1}{(1-F) + \frac{F}{G}} = \frac{1}{(1-0.1) + \frac{0.1}{2}} = 1.05$$

Como ninguna de las dos mejoras sale demasiado cara y el speedup obtenido con cada una de ellas no es espectacular, nos planteamos incorporar ambas al diseño. Pero en este caso, ¿cuál sería el speedup? La utilización de las dos mejoras no se solapa en el tiempo, o se utiliza una o se utiliza la otra (las instrucciones de enteros no utilizan el hardware para coma flotante y las de coma flotante no utilizan la ALU de enteros), por lo que se puede plantear de nuevo la ley de Amdhal, pero en este caso generalizando para varias mejoras:

$$t_{\text{antes de la mejora}} = t$$

$$t_{\text{después de la mejora}} = (1 - \sum F) \cdot t + \sum \frac{F}{G} \cdot t$$

$$S = \frac{t_{\text{antes de la mejora}}}{t_{\text{después de la mejora}}} = \frac{1}{(1 - \sum F) + \sum \frac{F}{G}}$$

Y esta expresión se puede utilizar siempre y cuando la utilización de las mejoras no se solape, es decir, que cuando se esté utilizando una, no se pueda utilizar ninguna de las otras.

En nuestro ejemplo, aplicando las dos mejoras al diseño del procesador tendríamos un speedup de:

$$S = \frac{t_{\text{antes de la mejora}}}{t_{\text{después de la mejora}}} = \frac{1}{(1 - \sum F) + \sum \frac{F}{G}} = \frac{1}{(1 - 0.45 - 0.1) + \frac{0.45}{1.25} + \frac{0.1}{2}} = 1.16$$

Cuando la ley de Amdhal se aplica a arquitecturas multiprocesador y multicomputador, establece un límite máximo para la ganancia que se puede obtener paralelizando una aplicación sobre N procesadores. Este límite lo fija la fracción serie o secuencial de la aplicación, 1-F, que en este caso es la fracción de tiempo que no se puede utilizar la mejora que implica la paralelización:

$$S = \frac{t_{\text{antes de la mejora}}}{t_{\text{después de la mejora}}} = \frac{t_{\text{secuencial}}}{t_{\text{paralelo}}} = \frac{1}{(1-F) + \frac{F}{N}}$$

$$S_{\max(N \rightarrow \infty)} = \frac{1}{1-F}$$

Sin embargo, este planteamiento se puede considerar algo pesimista, ya que en muchos casos, al aumentar el número de procesadores sobre los que se ejecuta una aplicación también aumenta el tamaño de la parte paralela del programa, sin que aumente la fracción serie. Por ejemplo, es típico aumentar la resolución espacio-temporal del problema, realizando cálculos para un número mayor de puntos. De estas consideraciones surge la ley de Gustafson, que al contrario que la ley de Amdhal, no supone que el tamaño del problema es fijo.

Según esta ley, si sobre  $N$  procesadores el tiempo de ejecución de un programa es la suma del tiempo que tarda en ejecutarse la fracción serie y la fracción paralela, al pasar a ejecutar este mismo programa sobre un único procesador y no poder aprovechar el paralelismo, la fracción paralela tardará  $N$  veces más en ejecutarse:

$$t_{\text{antes de la mejora}} = (1 - F) \cdot t + N \cdot F \cdot t$$

$$t_{\text{después de la mejora}} = t = (1 - F) \cdot t + F \cdot t$$

$$S = \frac{t_{\text{antes de la mejora}}}{t_{\text{después de la mejora}}} = (1 - F) + N \cdot F$$

Por lo que el speedup máximo no está limitado, en principio, por la fracción serie del programa:

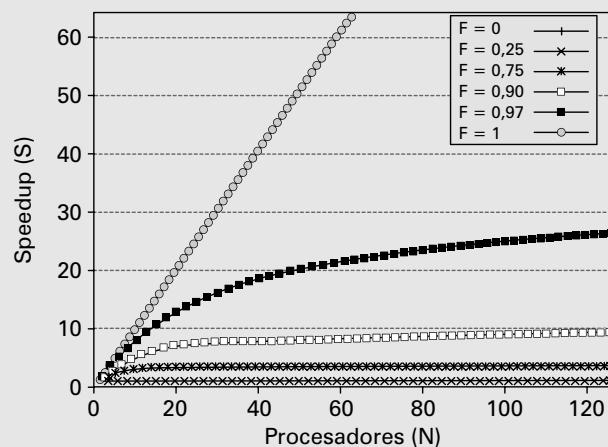
$$S_{\max(N \rightarrow \infty)} = \infty$$

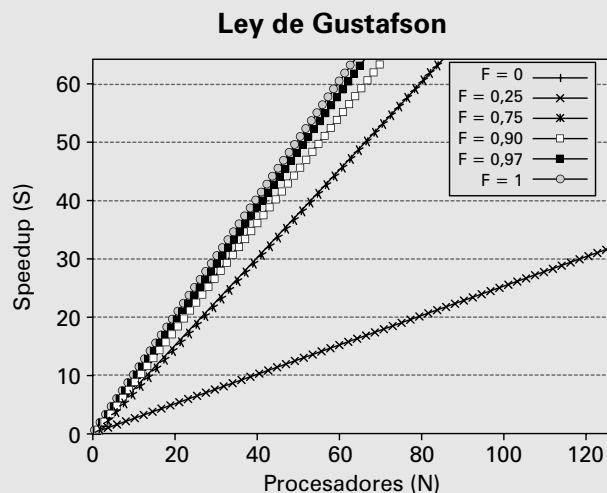
### Ejemplo 6.6

#### Comparación de la ley de Amdhal y la ley de Gustafson.

Para comprender mejor la diferencia entre las leyes de Amdhal y Gustafson, estudiemos la evolución del speedup que se consigue paralelizando una aplicación, en función de su fracción serie y del número de procesadores disponibles para su ejecución paralela:

#### Ley de Amdhal





Lo primero que se observa es que ambas leyes predicen la misma evolución del speedup para el caso óptimo en el que toda la aplicación es paralelizable ( $F=1$ ).

Además, se puede observar que en el resto de casos la ley de Amdhal es bastante más pesimista que la de Gustafson, ya que el speedup se ve fuertemente penalizado por la fracción serie de la aplicación, por pequeña que esta sea y por muchos procesadores que se puedan utilizar para ejecutar la parte paralela de dicha aplicación.

Esta aparente contradicción no lo es, ya que las dos leyes se basan en premisas diferentes: la ley de Amdhal es válida cuando las aplicaciones tienen un volumen de trabajo fijo mientras que la ley de Gustafson tiene en cuenta que ciertas aplicaciones pueden aumentar el volumen de trabajo cuando aumenta el número de procesadores. Y afortunadamente esto es bastante frecuente ya que, como se mencionaba antes, en muchas aplicaciones es típico aumentar la resolución espacio-temporal del problema cuando aumenta el tamaño del sistema.

De cualquier manera siempre sabemos que la ley de Amdhal nos da la estimación más pesimista del speedup que se puede obtener, y si es posible aumentar el tamaño del problema, el speedup que se consiga finalmente estará seguro por encima de este valor.

## 6.4 Técnicas de medida y benchmarks

Sea cual sea la métrica de rendimiento escogida para cuantificar las prestaciones de un sistema, es necesario establecer una metodología para realizar las medidas de rendimiento y escoger qué programa o conjunto de programas se deben ejecutar en el sistema para llevarlas a cabo. La evaluación de rendimiento del sistema se realiza en muchas ocasiones mediante simulaciones, pero estas técnicas escapan del alcance de este libro.

En la mayor parte de los casos los usuarios o diseñadores desean evaluar el rendimiento del sistema ejecutando en él sus propias aplicaciones, de manera que puedan predecir exactamente cuáles serán las prestaciones obtenidas para la carga típica que el sistema va a ejecutar. Pero también en la mayor parte de los casos esto no es posible por problemas de portabilidad que obligarían a recomilar las aplicaciones, porque éstas no están preparadas para realizar las medidas de rendimiento, porque estas medidas serían demasiado costosas en tiempo y/o en recursos, etc.

Por estos motivos se suele recurrir a un tipo especial de programas denominados benchmarks, específicamente diseñados para realizar la evaluación de rendimiento. Esto implica que son fáciles de instalar y

ejecutar, y que además cumplen con unos requisitos mínimos de portabilidad que permiten su ejecución sobre diferentes plataformas.

### 6.4.1. Técnicas de medida

Para obtener los valores de las métricas de rendimiento escogidas existen diferentes técnicas de medida que se pueden clasificar en cuatro grandes grupos que se explican a continuación. En cualquier caso, escoger la técnica de medida más adecuada para cada situación no es tarea sencilla, pero siempre hay que tener en cuenta el tipo de métrica de rendimiento que se desea medir (ya que no todas las técnicas serán adecuadas para ciertas métricas), el coste en tiempo y recursos que implica hacer las medidas, las perturbaciones que las propias medidas pueden introducir en el sistema y la precisión de los resultados obtenidos.

#### 6.4.1.1. UTILIZACIÓN DIRECTA DE LA INFORMACIÓN MANTENIDA POR EL SISTEMA OPERATIVO

Prácticamente todos los sistemas operativos actuales mantienen información sobre las diferentes tareas que se están ejecutando en el sistema y sobre la utilización que están haciendo de sus recursos.

Aunque la finalidad de mantener esta información no es proporcionar una herramienta de medida de rendimiento, el sistema de información del sistema operativo realiza funciones similares a las de cualquier monitor de rendimiento. La información típica que suele mantener el sistema operativo incluye la utilización del procesador y de la memoria, el número de tareas en el sistema, el tiempo de inicio y de finalización de cada tarea, el tiempo que cada tarea ha pasado realmente en el procesador, etc.

La principal ventaja de utilizar este tipo de información es que no hace falta desarrollar código específico para realizar las medidas de rendimiento, pero la principal desventaja es la sobrecarga que el acceso a esta información mantenida por el sistema operativo puede suponer dependiendo de la frecuencia con la que se realice.

Además, el formato en el que esta información se almacena no suele ser muy amigable e impone bastantes restricciones en su utilización. Por último, hay que tener en cuenta que el sistema operativo no suele mantener información acerca de la longitud de las colas de ejecución de los distintos recursos (distintos del procesador), del número de eventos relacionados con las funciones de E/S o del número de cambios de contexto, sólo por poner algunos ejemplos.

#### 6.4.1.2. MONITORES SOFTWARE

En este caso para realizar las medidas de rendimiento se ejecuta una aplicación específicamente programada para recoger y tratar la información necesaria. Normalmente esta información se recoge de la mantenida por el sistema operativo, por otras aplicaciones o de otro tipo de variables del sistema.

Este tipo de monitores pueden utilizar dos técnicas de medida:

- **Detección de eventos.** El monitor controla ciertas entradas del sistema operativo para detectar eventos tales como interrupciones, llamadas al sistema, fallos de página o finalización de turnos de planificación del procesador. De nuevo existen dos opciones:
  - **Cuenta de eventos.** Sólo interesa conocer el número de eventos de un determinado tipo que se han producido cuando finaliza la monitorización del sistema.
  - **Traza.** En este caso no sólo se almacena un contador con el número de eventos que se han producido sino también alguna parte del estado del sistema que permita identificar a cada evento que se ha producido de manera única. El problema de este tipo de monitores es que consumen muchos más recursos del sistema para hacer la evaluación de rendimiento, más tiempo y más almacenamiento.

- **Muestreo.** En este caso el monitor se activa cada cierto tiempo (periodo de monitorización) para recoger información sobre el estado del sistema.

La principal ventaja de los monitores software es que se pueden adecuar a las necesidades del usuario y recogen la información necesaria para realizar las medidas de rendimiento escogidas, ni más ni menos. Además se puede incorporar al monitor el tratamiento de la información recogida, de manera que si la métrica de rendimiento se obtiene de manera indirecta y no puede medirse directamente, el monitor proporciona el valor final de la métrica. Pero estos monitores suelen ser muy dependientes de la arquitectura y del sistema operativo, incluso en pequeños matices, por lo que no suelen ser reutilizables.

Comparando los monitores que funcionan por detección de eventos y por muestreo, se encuentran algunas diferencias significativas. En general, la detección de eventos suele darse a un nivel más bajo que el muestreo por lo que para utilizar monitores por detección de eventos suele ser imprescindible parchear el kernel del sistema operativo, ya que si no, no sería posible detectar los eventos (interrupciones, fallos de página, cambios de contexto, etc). Por el contrario, los monitores que funcionan por muestreo suelen ir por encima del sistema operativo, sin necesidad de modificar ninguna de sus rutinas ya que sólo recogen información acerca de ciertos aspectos del estado del sistema.

En cuanto a la sobrecarga que este tipo de monitores suponen para el sistema, el número de eventos de un determinado tipo que se va a producir es completamente impredecible cuando se comienza la monitorización, así que no es posible saber a priori cual va a ser la sobrecarga introducida por una detección de eventos. Por el contrario, una vez que se fija el periodo de monitorización de un monitor por muestreo, se conoce con exactitud la sobrecarga que el monitor va a suponer para el sistema, porque se sabe cuántas veces va a ejecutarse en un periodo de tiempo concreto.

Cuando el evento que se monitoriza se produce con una gran frecuencia, la sobrecarga que el monitor por detección de eventos supone para el sistema es inaceptable (especialmente si se basa en la realización de trazas) y se obtiene mejor rendimiento con un muestreo. Sin embargo, cuando el evento se produce con una frecuencia muy baja, el monitor por detección de eventos no supondrá una gran carga para el sistema y además, se ejecutará sólo cuando sea necesario. Además, en este último caso la elección del periodo de monitorización para el monitor por muestreo será crítica para la realización de las medidas, ya que si no se escoge bien puede que ciertos eventos pasen inadvertidos para el monitor.

#### 6.4.1.3. MONITORES HARDWARE

Las medidas de rendimiento han adquirido tanta importancia en el área de la arquitectura de computadoras que los fabricantes han terminado por incluir en el propio hardware monitores que permitan contar con gran precisión una multitud de eventos.

Los eventos que permiten medir este tipo de monitores son los de más bajo nivel, relacionados estrechamente con la arquitectura del sistema. Las ventajas que presentan estas herramientas de medida son tres:

- No se consumen recursos (tiempo de procesador y almacenamiento) para realizar la monitorización del sistema.
- Se pueden monitorizar eventos de muy bajo nivel que son completamente transparentes para el software e incluso para el sistema operativo.
- Son mucho más precisos que los monitores software en la medición de algunas magnitudes.

La principal desventaja de estos monitores es que las magnitudes que se pueden medir son físicas, no lógicas. Por lo tanto es mucho más difícil distinguir entre las diferentes tareas que están utilizando los recursos (no hay identificadores de proceso a este nivel). Además, la instalación, configuración y utilización de las librerías que permiten acceder a estos monitores suele ser complicada y exige un gran conocimiento de la arquitectura que se está monitorizando.

#### 6.4.1.4. ANÁLISIS DE PROGRAMAS (PROFILING)

Esta técnica suele utilizarse cuando la evaluación de rendimiento es a un nivel más alto y va orientada a la optimización de un código o de un sistema. En estos casos casi siempre se emplean medidas de tiempos y de utilización de recursos. Por ejemplo, un desarrollador puede analizar qué fracción del tiempo de ejecución total se consume en los diferentes bloques de una aplicación para saber en cuáles se tiene que centrar para mejorar el rendimiento global. O un administrador puede analizar en qué parte del sistema pasan más tiempo las aplicaciones que se ejecutan en él para detectar y eliminar los posibles cuellos de botella.

La manera general de realizar este tipo de análisis es el muestreo mediante interrupciones software. Se escogen aleatoriamente ciertas partes del código que se analiza o ciertas aplicaciones que se ejecutan en el sistema que se evalúa (normalmente, fijando un periodo de muestreo, de manera que cada cierto tiempo se produce una interrupción y la RTI es la encargada de recoger la información), y se infiere a partir del comportamiento de esta muestra (utilización de recursos y/o tiempos de ejecución) cuál es el comportamiento general del programa o sistema.

En el caso de analizar el comportamiento de un único programa, se puede utilizar la cuenta de bloques básicos si no se desea realizar una estimación estadística sino una evaluación exacta del programa. En este caso se divide el código en bloques básicos, que no son más que conjuntos de instrucciones en los que no hay ningún salto, es decir, que se asegura que se van a ejecutar todas ellas una vez que comience la ejecución de ese bloque.

Se insertan en cada bloque básico una serie de instrucciones y directivas que permiten realizar su análisis y contar cuántas veces se ejecutan en total, por lo que al finalizar la ejecución de la aplicación, se puede recoger toda esta información y realizar un profiling completo.

Obviamente, esta técnica no está sujeta a errores aleatorios como el muestreo, el problema es que hay que modificar el código y que la ejecución de las instrucciones que se añaden para realizar la evaluación de rendimiento puede añadir una gran sobrecarga a la aplicación, incluso, provocando un cambio en la utilización de la jerarquía de memoria (se almacena una gran cantidad de información relativa al profiling porque hay una gran cantidad de bloques básicos) que pueda empeorar mucho su tiempo de ejecución. Por lo que sólo se utiliza cuando se necesita una gran precisión en la evaluación de rendimiento.

#### 6.4.2. Tipos de benchmark

Existen multitud de dominios de aplicación en los que hoy en día puede ser necesario realizar una evaluación de rendimiento, y por ello, existen también distintos tipos de benchmarks. Por ejemplo, un diseñador de procesadores que está realizando simulaciones para escoger entre dos alternativas de planificación dinámica necesitará un código rápido y sencillo que permita realizar medidas de rendimiento asociadas sólo al procesador, pero una multinacional que debe escoger entre diferentes servidores para ejecutar sus aplicaciones de gestión corporativa, necesitará benchmarks mucho más variados y complejos para poder tomar su decisión.

Aunque existen diferentes clasificaciones de benchmarks, a continuación se detalla una de las más aceptadas:

- **Benchmark sintético.** Se trata de un programa desarrollado específicamente para la evaluación de rendimiento, por lo tanto, completamente artificial y que no realiza ningún trabajo útil. Para que pueda extraerse información útil a partir de las medidas tomadas durante su ejecución, se intenta que el conjunto de instrucciones que contiene este benchmark sea un conjunto representativo del tipo de aplicación que el sistema va a ejecutar en la realidad (proporción de operaciones de enteros y de coma flotante, porcentaje de instrucciones que acceden a memoria o que son saltos condicionales, etc). Sin embargo, aunque estas proporciones se estudien minuciosamente en las aplicaciones reales y se reproduzcan en el benchmark sintético, es posible que los efectos introducidos por el orden de las instrucciones en las aplicaciones reales (dependencias y riesgos) y por el patrón de

accesos a memoria, hagan que los resultados del benchmark sean diferentes de los que luego se obtengan en la realidad. Por otro lado, suelen ser benchmarks muy sencillos de desarrollar y utilizar. Un ejemplo son los conocidos benchmarks Whetstone y Dhrystone.

- **Microbenchmark.** Se trata de un benchmark sintético mucho más corto y sencillo que se centra en evaluar el rendimiento de un componente muy concreto del sistema, por ejemplo, el controlador de memoria o una determinada unidad de coma flotante. Para desarrollarlos hay que tener un gran conocimiento de arquitectura de computadoras, y suelen utilizarse para detectar cuellos de botella y para caracterizar el rendimiento máximo de las diferentes unidades de un sistema, por ejemplo, para realizar una simulación.
- **Kernel.** Se trata de la parte más costosa de ejecutar o más representativa de una aplicación real. Un kernel suele estar compuesto sólo por unas pocas líneas de código, por lo que al no utilizar la aplicación completa para realizar medidas de rendimiento, se eliminan los problemas de dificultad de ejecución, portabilidad o alto coste de ejecución. La idea detrás de este tipo de benchmark es que escogiendo de manera inteligente la parte de la aplicación real que se convierte en el kernel para medida de rendimiento, se puede predecir el rendimiento que el sistema tendría para la aplicación completa realizando medidas sólo sobre el kernel. El problema es que en muchos casos no es posible encontrar en una pequeña parte de una aplicación los riesgos de datos y control, los accesos a memoria, las operaciones de E/S, etc. que hacen que la aplicación completa obtenga unas prestaciones determinadas. Un par de ejemplos conocidos de kernels son los Livermore loops y los NAS kernels.
- **Benchmark de juguete.** Es una aplicación real que produce un resultado real, pero que es sencilla y que utiliza un conjunto de datos de entrada pequeño. Ejemplos típicos son el algoritmo de ordenación quicksort, las torres de Hanoi, o la transformada de Fourier rápida. Aunque hace unos años fueron muy populares, los resultados obtenidos con uno sólo de estos algoritmos no son adecuados para realizar una evaluación de rendimiento completa.
- **Conjunto de aplicaciones benchmark.** Por todos los inconvenientes señalados en los anteriores tipos de benchmark, se ha llegado a la conclusión de que en la mayor parte de los casos la mejor opción es utilizar un conjunto de aplicaciones representativas de la carga típica del sistema, previamente estandarizadas por algún organismo o empresa. Existen conjuntos para la evaluación de sistemas cuando su carga típica son aplicaciones científicas y de ingeniería (por ejemplo el conjunto Perfect Club) o aplicaciones de gráficos y multimedia (por ejemplo SPEC viewperf) por poner sólo un par de ejemplos. Para que la ejecución de esos benchmarks sea sencilla y no demasiado costosa, se suelen utilizar conjuntos de datos de entrada más sencillos que en la realidad. Esto puede hacer que no se caracterice del todo bien el rendimiento de la jerarquía de memoria o del sistema de E/S, pero hasta el momento estos conjuntos de aplicaciones han demostrado ser una buena opción en la mayor parte de los casos.

### CASO PRÁCTICO 6.1. SPEC.

Standard Performance Evaluation Corporation (SPEC) es una organización que se encarga de proponer y mantener conjuntos de aplicaciones benchmark para diferentes campos de aplicación, y de estandarizar las métricas y metodologías de medida asociadas a estos conjuntos.

Nos centraremos en este caso práctico en SPEC CPU, que es el conjunto de benchmarks más utilizado para caracterizar el rendimiento de CPUs en sistemas de propósito general. Por ejemplo, el conjunto SPEC CPU 2006 está compuesto por 29 aplicaciones que evalúan el rendimiento del sistema tanto en operaciones con enteros (12 de las aplicaciones) como en operaciones en coma flotante (las otras 17). Dentro de

este conjunto de benchmarks hay aplicaciones de compilación y programación en diferentes lenguajes, compresión de diferentes tipos de datos, algoritmos genéticos, inteligencia artificial y juegos, simulación, cómputo científico, etc.

| Benchmarks para enteros | Benchmarks para coma flotante |
|-------------------------|-------------------------------|
| 400.perlbench           | 410.bwaves                    |
| 401.bzip2               | 416.gamess                    |
| 403.gcc                 | 433.milc                      |
| 429.mcf                 | 434.zeusmp                    |
| 445.gobmk               | 435.gromacs                   |
| 456.hmmer               | 436.cactusADM                 |
| 458.sjeng               | 437.leslie3d                  |
| 462.libquantum          | 444.namd                      |
| 464.h264ref             | 447.dealll                    |
| 471.omnetpp             | 450.soplex                    |
| 473.astar               | 453.povray                    |
| 483.xalancbmk           | 454.calculix                  |
|                         | 459.GemsFDTD                  |
|                         | 465.tonto                     |
|                         | 470.lbm                       |
|                         | 481.wrf                       |
|                         | 482.sphinx3                   |
|                         | 999.specrand                  |

Se trata de un conjunto de aplicaciones que representa de la manera más completa posible la carga de trabajo habitual de las computadoras de propósito general actuales. Para todos los benchmarks existe una documentación que describe exhaustivamente el tipo de trabajo que realizan, las entradas que necesitan, las salidas que producen, el lenguaje de programación en el que están escritos y los aspectos relacionados con su portabilidad (ya que se evalúan con estos benchmarks una gran variedad de plataformas).

SPEC no sólo proporciona este conjunto de aplicaciones sino también una métrica de rendimiento asociada y una metodología para su medida. Lo primero que se hace es medir el tiempo de ejecución del sistema evaluado para cada una de las aplicaciones del conjunto de benchmarks.

A continuación se normalizan los tiempos obtenidos, dividiéndolos por los tiempos obtenidos para esas mismas aplicaciones en un sistema de referencia proporcionado por SPEC. Por último, se hace la media geométrica de todos estos tiempos normalizados, y esa es la métrica que permite cuantificar el rendimiento del sistema.

¿Por qué se utiliza la media geométrica? En principio es razonable utilizar algún tipo de media, ya que no es muy práctico manejar 29 tiempos de ejecución diferentes para realizar una evaluación de rendimiento y obtener conclusiones generales y sencillas. Y se utiliza la geométrica porque tiene una propiedad que no tienen otras medias como pueda ser la aritmética; si se modifica el sistema de referencia en algún momento (el que se utiliza para la normalización de los tiempos), el valor absoluto de la métrica de rendimiento para un sistema concreto se modifica también, pero su relación con la métrica obtenida para otros sistemas (el sistema 1 es 2 veces más rápido que el sistema 2, etc.) se mantiene exactamente igual.

Y esto es importante para una métrica como la de SPEC, que publica en su página web ([www.spec.org](http://www.spec.org)) evaluaciones de rendimiento de multitud de sistemas y plataformas por lo que se utiliza muy a menudo para hacer comparaciones de rendimiento y para encontrar configuraciones óptimas.

## 6.5 Métricas de rendimiento complejas

Las métricas sencillas definidas al principio de este capítulo suelen ser suficiente para evaluar el rendimiento de arquitecturas sencillas compuestas por un único procesador, una jerarquía de memoria y un sistema de E/S.

Pero en el caso de sistemas multiprocesador o multicomputador, sean estos tradicionales o vayan implementados en un único chip, casi siempre son necesarias métricas adicionales para obtener una evaluación de rendimiento completa.

Estas métricas suelen estar relacionadas con diferentes aspectos, como puedan ser la eficiencia con la que se utilizan todos los recursos disponibles, el consumo de potencia, el rendimiento del sistema de comunicaciones, la fiabilidad, la estabilidad, la disponibilidad, la tolerancia a fallos, la flexibilidad, la sensibilidad o la robustez.

Además para realizar medidas de rendimiento en estos sistemas existen benchmarks específicos como SPEC HPC, los NAS Parallel Benchmarks (NPB) o PARKBENCH, por poner sólo algunos ejemplos.

Si además estos sistemas paralelos están implementados on-chip, suelen utilizarse métricas de rendimiento de más bajo nivel como puedan ser el área, el consumo de potencia, el número de pines, etc; y benchmarks mucho más específicos.

### 6.5.1. Speedup en sistemas paralelos y eficiencia

Estas métricas son de las más utilizadas en el caso de arquitecturas multiprocesador y multicomputador. En estos sistemas con varios procesadores, el speedup suele utilizarse para cuantificar cuánto más rápido puede ejecutarse una aplicación si se paralleliza sobre N procesadores que si se ejecuta en uno sólo.

En este contexto se han definido distintos tipos de speedup:

- Speedup relativo:

$$S_{rel} = \frac{t \text{ de ejecución del código } P \text{ en 1 procesador}}{t \text{ de ejecución del código } P \text{ en } N \text{ procesadores}}$$

- Speedup real:

$$S_{real} = \frac{t \text{ de ejecución del mejor código secuencial en 1 procesador}}{t \text{ de ejecución del código } P \text{ en } N \text{ procesadores}}$$

- Speedup absoluto:

$$S_{abs} = \frac{t \text{ de ejecución del mejor código secuencial en el mejor procesador}}{t \text{ de ejecución del código } P \text{ en } N \text{ procesadores}}$$

- Speedup normalizado:

$$S_{norm} = \frac{S}{\text{coste sistema paralelo} / \text{coste sistema 1 procesador}}$$

Como se puede observar, la diferencia entre las tres primeras definiciones se encuentra en el tiempo de ejecución utilizado para caracterizar el comportamiento de la aplicación en el caso de ejecutarla sobre un único procesador.

El speedup absoluto es el más fiable si se desea cuantificar la ganancia real que se obtiene al ejecutar una aplicación sobre N procesadores, ya que compara el tiempo de ejecución en esta situación con el que se obtendría en el mejor de los casos para un único procesador; se escoge la mejor implementación secuencial posible de la aplicación y además se ejecuta sobre el procesador más rápido. Sin embargo se utiliza muy poco, ya que en pocas ocasiones se tiene acceso al mejor procesador para realizar las medidas,

y además este mejor procesador no es siempre el mismo, al contrario, varía bastante a menudo con las mejoras que se van introduciendo en los diseños.

El speedup real se utiliza algo más ya que no exige esta ejecución sobre el mejor procesador, pero sí que exige programar dos versiones de la aplicación, la paralela y la secuencial en su mejor versión, es decir, con todas las optimizaciones posibles. Normalmente se ejecuta la aplicación en el sistema paralelo y a continuación en el mismo sistema pero utilizando uno sólo de los procesadores. El problema es que, en ocasiones, al ejecutar la versión secuencial sobre un único procesador, no hay suficiente memoria para hacerlo.

Siempre que se maneje este tipo de speedup hay que tener en cuenta que los valores obtenidos son mejores cuanto peor es el procesador sobre el que se ejecuta la versión secuencial de la aplicación. En el caso del speedup absoluto, es muy común que la paralelización de la aplicación, si ésta no tiene una complejidad suficiente, no resulte en una mejora. Es decir, que se obtenga un speedup por debajo de 1 debido a los retardos que introduce la compartición de la memoria o la comunicación entre procesadores, y que no existen en la versión secuencial, que además se ejecuta sobre el mejor procesador posible.

Sin embargo, al utilizar el speedup real, la comparación de los tiempos de ejecución suele favorecer a la versión paralela, porque no se ejecuta la versión secuencial sobre la mejor plataforma posible.

Y si se utiliza el speedup relativo, esto ocurre en mayor medida, porque el código utilizado para hacer las medidas de la versión secuencial ni siquiera tiene que estar optimizado. Se suele utilizar como código secuencial el que se emplea como punto de partida para la paralelización de la aplicación. Y de nuevo, cuando menos eficiente sea esta versión del código, mayor valor tendrá el speedup.

En resumen, la utilización del speedup relativo o del speedup real es mucho más sencilla que la del absoluto, pero hay que ser cuidadosos puesto que puede llevar a conclusiones erróneas (sobrevalorando la mejora que se puede obtener gracias a la paralelización). Además, cabe destacar que en cualquiera de las tres primeras definiciones del speedup, aunque lo normal es que el límite máximo al paralelizar una aplicación sea el número de procesadores sobre el que se ejecuta la aplicación paralela,  $S_{max}=N$ , algunos autores han demostrado que este límite puede superarse. Por ejemplo, por aspectos relacionados con la jerarquía de memoria que limiten en gran medida el tiempo de ejecución de la versión secuencial de la aplicación cuando se ejecuta sobre un único procesador.

En cuanto a la última definición del speedup, el normalizado, intenta tener en cuenta el coste del sistema con un procesador y el del sistema paralelo para normalizar el speedup obtenido con la paralelización de la aplicación, que puede ser el absoluto, el real o el relativo (dependiendo del que se utilice, el coste del sistema con un procesador será uno u otro).

Aunque esta definición resulta muy atractiva en muchos casos de evaluación de rendimiento, la principal dificultad que plantea está en definir a su vez el coste de los dos sistemas que se comparan. ¿Qué aspectos se tienen en cuenta? Por un lado, hay elementos hardware y software. Y por otro, hay costes de adquisición, de puesta en marcha, de mantenimiento, etc, que no siempre son fáciles de cuantificar.

En esta misma línea de normalización del speedup existe una métrica mucho más fiable y sencilla de medir denominada eficiencia. La idea fundamental detrás de esta definición es que no es lo mismo obtener un determinado speedup al paralelizar una aplicación si se hace sobre 3 procesadores que sobre 30. Si fuera así, la paralelización sobre el sistema con menor número de procesadores estaría utilizando de manera mucho más eficiente los recursos y por lo tanto, ofrecería unas mejores prestaciones.

Por tanto, se define la eficiencia como:

$$\varepsilon = \frac{S}{N}$$

Donde S es el speedup absoluto, real o relativo, y N es el número de procesadores. Obviamente, como los speedups real y relativo se ven favorecidos por sistemas de un procesador con rendimiento pobre y por implementaciones secuenciales poco optimizadas, lo mismo ocurre con la eficiencia, por lo que hay que ser prudente en la interpretación de sus valores.

### 6.5.2. Escalabilidad

Como ya se ha explicado en secciones anteriores, el speedup que se puede conseguir al paralelizar una aplicación no depende sólo del número de procesadores de la arquitectura paralela sino también de ciertas características de la aplicación. Intuitivamente, un sistema se considera escalable si al aumentar el número de procesadores y el tamaño del problema, es capaz de aumentar también su rendimiento (o por lo menos, de mantenerlo).

Se definen distintos tipos de escalabilidad dependiendo de qué aspecto de rendimiento es el que tiene que escalar al aumentar el tamaño del problema y del sistema. Aunque existen multitud de definiciones de métricas de escalabilidad, todas ellas siguen la misma metodología para medir el valor de esta métrica:

- Se escoge un aspecto de rendimiento que caracterice el comportamiento del sistema: speedup, eficiencia, latencia, calidad de servicio, potencia, etc.
- Se analiza el valor de este aspecto de rendimiento en diferentes configuraciones del sistema, incrementando progresivamente el número de procesadores, y para diferentes tamaños del problema.
- El sistema se considera escalable si puede incrementar, o por lo menos mantener constante el aspecto de rendimiento escogido para caracterizar el comportamiento del sistema, al aumentar el número de procesadores (escogiendo un tamaño adecuado del problema).
- El grado de escalabilidad suele estar determinado por el cociente entre el incremento en el tamaño del problema y el incremento en el número de procesadores del sistema necesario para conseguir que el rendimiento se mantenga constante.

#### Ejemplo 6.7

##### Definición de la Escalabilidad P.

En los trabajos de Prasad Jogalekar y Murray Woodside se define una métrica muy completa para cuantificar la escalabilidad de arquitecturas de memoria distribuida que vamos a intentar comprender a continuación.

En este caso el aspecto de rendimiento escogido para escalar con el incremento del tamaño del sistema y del tamaño del problema es la potencia del sistema ( $P$ ), definida como:

$$P = \frac{\lambda}{T}$$

Donde  $\lambda$  es productividad del sistema (número de tareas completadas por unidad de tiempo) y  $T$  es el tiempo de ejecución medio de una tarea.

La escalabilidad  $P$  entre una configuración 1 y una configuración 2 del sistema ( $\psi_{1 \rightarrow 2}$ ) se define como:

$$\psi_{1 \rightarrow 2} = \frac{\frac{P_2}{C_2}}{\frac{P_1}{C_1}} = \frac{\lambda_2 \cdot T_1 \cdot C_1}{\lambda_1 \cdot T_2 \cdot C_2}$$

Donde  $C$  es una métrica para el coste del sistema definida por el usuario en función de sus necesidades (dependerá de si el sistema es un cluster, un grid, un cloud, una arquitectura on-chip, etc).

La escalabilidad perfecta sería  $\psi_{1 \rightarrow 2} = 1$ , ya que implica que la configuración 2 puede conseguir una productividad  $k$  veces mayor que la configuración 1, manteniendo el mismo tiempo de ejecución medio para una tarea y aumentando el coste en un factor  $k$ . Por encima de 1 el sistema presenta escalabilidad positiva, y por debajo, no es escalable.

Los autores proponen establecer una estrategia de escalado para realizar un análisis de escalabilidad completo del sistema. Esta estrategia debe fijar:

- Qué configuraciones del sistema se incluyen en el análisis. Normalmente se incrementa progresivamente el número de procesadores desde 1 hasta el máximo posible.
- En cada nueva configuración puede modificarse sólo el número de procesadores respecto de la anterior (estrategia de escalado fija) o si por el contrario, pueden modificarse los aspectos denominados posibilitadores de escalabilidad (estrategia de escalado variable) para conseguir que la potencia del sistema no decrezca. Los posibilitadores de escalabilidad están típicamente relacionados con el tamaño del problema, el diseño de la aplicación, la cantidad de memoria del sistema, etc.

### Ejemplo 6.8

#### Generalización de la escalabilidad P.

La definición de la escalabilidad del ejemplo anterior presenta un problema típico: si el tiempo medio de ejecución tiene un valor muy pequeño, el valor de la escalabilidad tiende a infinito, lo que hace que sea muy difícil de interpretar. Por eso los mismos autores han propuesto una generalización de esta métrica, redefiniendo la potencia como:

$$P = \lambda \cdot f(T, T_{esp})$$

Donde  $\lambda$  es de nuevo la productividad del sistema (número de tareas completadas por unidad de tiempo),  $T$  es el tiempo de ejecución medio de una tarea y  $T_{esp}$  es el tiempo de ejecución medio esperado, es decir, un valor al que se desearía llegar para este sistema y esta aplicación.

La función  $f$  se puede definir de diferentes maneras, los autores proponen la siguiente en su trabajo:

$$f(T, T_{esp}) = \frac{1}{1 + \frac{T}{T_{esp}}}$$

Por lo que la escalabilidad  $P$  entre una configuración 1 y una configuración 2 del sistema ( $\psi_{1 \rightarrow 2}$ ) se define como:

$$\psi_{1 \rightarrow 2} = \frac{\frac{P_2}{C_2}}{\frac{P_1}{C_1}} = \frac{\lambda_2 \cdot C_1 \cdot (T_1 + T_{esp1})}{\lambda_1 \cdot C_2 \cdot (T_2 + T_{esp2})}$$

Donde  $C$  es de nuevo una métrica para el coste del sistema definida por el usuario en función de sus necesidades.

## BIBLIOGRAFÍA Y LECTURAS RECOMENDADAS

- BUYYA, R. (1999): *High Performance Cluster Computing* (volúmenes 1 y 2), Prentice Hall.
- CULLER, D.; SINGH, J. P. & GUPTA, A. (1998): *Parallel Computer Architecture: A HardwareSoftware Approach*, Morgan Kaufmann.
- GUNTHER, N. (2000): *The Practical Performance Analyst*, iUniverse.
- HENNESSY, J. L. & PATTERSON, D. A. (2007): *Computer Architecture: A Quantitative Approach* (4.<sup>a</sup> ed.), Morgan Kaufmann.
- HWANG, K. (1992): *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw Hill.
- JAIN, H. K. (1991): *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley.
- JOGALEKAR, P. & WOODSIDE, M. (2000): "Evaluating the scalability of distributed systems", *IEEE Transactions on Parallel and Distributed Systems*, 11(6), 589-603.
- LILJA, D. J. (2005): *Measuring Computer Performance: A Practitioner's Guide*, Cambridge University Press.
- SAHNI, S. & THANVANTRI, V. (1996): "Performance metrics: keeping the focus on runtime"; *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(1), 43-56.
- SHIVA, S. G. (2005): *Advanced Computer Architectures*, CRC Press.
- SPEC (Standard Performance Evaluation Corporation). <http://www.spec.org/>.
- Top500 Supercomputer Sites. <http://www.top500.org/>.

## PROBLEMAS

- 6.1.** Se desea evaluar el rendimiento de tres configuraciones diferentes de una computadora: C1, C2 y C3. Para ello se utiliza un conocido benchmark compuesto por tres aplicaciones: PA, PB y PC. Los tiempos de ejecución que se miden para estas tres aplicaciones son, en segundos:

|    | C1  | C2  | C3  |
|----|-----|-----|-----|
| PA | 122 | 89  | 104 |
| PB | 50  | 140 | 99  |
| PC | 113 | 105 | 110 |

- a) Comparar el rendimiento de las tres configuraciones estudiadas con frases del tipo “La configuración X es N veces más rápida que la configuración Y”. Hacer esta comparación con el tiempo de ejecución total y con la media aritmética de los tiempos. Establecer conclusiones acerca de estas comparaciones y decidir cuál es la mejor configuración.
  - b) Tenemos a nuestra disposición los tiempos que este benchmark tarda en ejecutarse en una máquina de referencia de la organización propietaria del benchmark, que son 90, 45 y 102 segundos para las tres aplicaciones. Si queremos utilizar esta referencia para comparar los resultados de nuestra máquina con los de otros usuarios, ¿cuál es la manera correcta de hacerlo? Si lo hacemos así, ¿cuál es la mejor configuración?
  - c) Un mes después de obtener los resultados de nuestra evaluación de rendimiento, vemos que la máquina de referencia ha cambiado, siendo ahora los tiempos de referencia 80, 34 y 93 segundos. ¿Cambiará en algo la decisión que tomamos acerca de la mejor configuración si utilizamos estos nuevos tiempos?
  - d) Si nos damos cuenta de que el 90% del tiempo estamos utilizando aplicaciones del tipo de PA, el 8% del tiempo del tipo de PB y el 2% restante del tipo de PC, ¿cuál sería la mejor configuración?
- 6.2.** Se diseña un procesador nanoMIPS en el que las instrucciones enteras se ejecutan en 1 ciclo de reloj y las de coma flotante necesitan en media, 8 ciclos. Los programas que se van a ejecutar en este procesador invierten un 30% del tiempo en ejecutar operaciones en coma flotante. Desde el punto de vista coste-prestaciones ¿sería rentable rediseñar el procesador para que las operaciones de coma flotante fueran 5 veces más rápidas a costa de aumentar el coste total del procesador en un 50%?
- 6.3.** Se está valorando la posibilidad de incluir una extensión multimedia en un repertorio de instrucciones. Esta extensión consigue que las operaciones relacionadas con multimedia se ejecuten 16 veces más rápido.
- a) Dibujar un gráfico donde se muestre la evolución del speedup obtenido con la mejora con el porcentaje de tiempo que puede emplearse la extensión multimedia.
  - b) ¿Qué porcentaje de tiempo se debería utilizar la extensión del repertorio para conseguir una mejora global de 2?
  - c) ¿Y para conseguir la mitad del speedup máximo?
- 6.4.** Se valora introducir en el diseño de un procesador tres mejoras con ganancias G1=3, G2=10 y G3=25, que no pueden utilizarse simultáneamente. Si las mejoras 1 y 2 se pueden utilizar cada

una el 25% del tiempo, ¿qué fracción del tiempo se debe utilizar la mejora 3 para que el speedup total sea de 8?

- 6.5.** Una empresa adquiere un servidor valorado en 2000 euros en el que se ejecuta una aplicación A1 el 40% del tiempo y una aplicación A2 el resto del tiempo. La primera aplicación consume un 10% de su tiempo en operaciones de E/S, un 20% en operaciones en coma flotante y el resto en operaciones enteras. La segunda aplicación realiza operaciones de E/S el 60% del tiempo, y el resto lo dedica a operaciones con enteros. Se decide mejorar el sistema de E/S en un 180% y mejorar el coprocesador para operaciones en coma flotante de manera que duplique su velocidad. ¿Cuál sería la máxima inversión que compensaría realizar para hacer estas mejoras desde el punto de vista coste-prestaciones?
- 6.6.** La universidad acaba de adquirir una arquitectura de memoria compartida para ejecutar una aplicación de simulación el 70% del tiempo y otra de cálculo científico el resto del tiempo. La aplicación de simulación dedica el 35% del tiempo a realizar operaciones en coma flotante y el 10% del tiempo a realizar accesos a disco. La de cálculo científico realiza accesos a disco el 25% del tiempo que se está ejecutando. Todas las instrucciones tienen un CPI=1 y para ejecutar una operación en coma flotante el procesador necesita ejecutar en media 150 instrucciones enteras, ya que no incorpora unidades en coma flotante. Se propone adquirir un coprocesador para las operaciones de coma flotante que sólo necesite una instrucción para realizar una operación en coma flotante y que tenga para estas instrucciones un CPI=10. La otra opción sería adquirir un nuevo disco duro con el doble de velocidad. ¿Cuál de las dos opciones se debería escoger?
- 6.7.** En una arquitectura de memoria distribuida de 64 procesadores se ejecuta una aplicación con un 60% de fracción serie. Calcular:
- El speedup máximo que puede conseguirse al paralelizar esta aplicación respecto de su versión secuencial.
  - La eficiencia máxima que puede conseguirse en esta paralelización.
- 6.8.** Se ejecuta una aplicación de cálculo científico en una arquitectura de memoria compartida en la que el 17% del tiempo se invierte en realizar operaciones de E/S que no son paralelizables.
- ¿Qué speedup máximo puede obtenerse si se paraleliza esta aplicación sobre 32 nodos?
  - La aplicación inicial resuelve una serie de ecuaciones diferenciales sobre una malla espacial de 100 cortes por dimensión. Si se amplía el sistema a 64 nodos, el número de cortes puede incrementarse hasta 1000 por dimensión. ¿Cuál es en este caso la ganancia máxima que se puede conseguir con la paralelización?

## AUTOEVALUACIÓN

1. ¿Qué características son deseables en una métrica de rendimiento?
2. ¿Qué significa que una métrica sea lineal?
3. ¿Es la frecuencia de reloj una buena métrica de rendimiento? ¿Por qué?
4. ¿Qué enuncia la ley de Amdhal y cuáles son las implicaciones de esta ley?
5. ¿Qué diferencias hay entre la ley de Amdhal y la de Gustafson?
6. ¿Qué ventajas e inconvenientes tiene utilizar directamente la información de rendimiento proporcionada por el sistema operativo para realizar una evaluación de rendimiento?
7. ¿Qué es un monitor software y qué alternativas existen para su programación?
8. ¿Qué diferencias hay entre un kernel y un benchmark de juguete? ¿Cuáles son sus ventajas e inconvenientes?
9. ¿Qué diferencias hay entre el speedup relativo, el real y el absoluto?
10. ¿Cómo se define la eficiencia de una arquitectura paralela?

---

# Índice analítico

---

- 3Dnow!, 15
- Acceso a memoria, 79, 80
- Acceso Directo a Memoria (DMA), 126
- Adelantamiento o cortocircuito, 43, 44, 45
- AGP, 117, 119
- Algoritmo
  - de encaminamiento, 252
  - de ordenación quicksort, 311
  - de equilibrio de carga, 286
  - de Tomasulo, 146, 148, 172
- Aliasing, 151
- Alineación
  - de accesos a memoria, 8
  - de los registros del procesador, 8
- Almacenamiento
  - de operandos, 3, 4
  - interno tipo acumulador, 3
  - interno tipo GPR, 3
  - interno tipo pila, 3
  - local, 212
- ALUOp, 28
- ALUSrcA multiciclo, 28
- ALUSrcB multiciclo, 28
- Análisis de programas (profiling), 310
- Ancho
  - de banda, 81, 111, 207, 212, 243, 300
  - de datos, 111, 212
- Aprovechamiento de los recursos, 284
- Arbitraje del bus, 114
- Arquitecturas
  - débilmente acopladas, 239
  - de memoria compartida, 237, 254
- de memoria compartida-distribuida, 238, 274
- de memoria distribuida, 239, 283
- fuertemente acopladas, 237
- Atomicidad, 270
- Back-End, 139
- Benchmark
  - de juguete, 311
  - sintético, 310
  - tipo Kernel, 311
- Benchmarks, 307
  - Dhrystone, 311
  - Livermore loops, 311
  - NAS Kernels, 311
  - Whetstone, 311
- Big Endian, 7
- Bits de veneno (poison bits), 181
- Branch folding, 160
- BTB (Branch Target Buffer), 154
- Bucle de espera, 120
- Buffer
  - de escritura, 92, 194
  - de prebúsqueda, 200
  - de predicción de saltos, 151
  - de predicción destino de salto, 151, 154
  - de Reordenamiento (ROB), 174
- Burst Extended Data Out (BEDO DRAM), 102
- Bus
  - compartido, 237
  - de memoria, 117
  - del sistema, 117
- Buses, 110, 246

- asíncronos, 113
- de ciclo partido, 114
- de E/S, 112
- de expansión, 117
- semisíncronos, 114
- síncronos, 113
- 
- Caché
  - de trazas, 205
  - de víctimas, 194, 195
  - no bloqueante, 81, 196
  - predictiva, 197, 198
  - pseudoasociativa, 197, 199
  - segmentada, 204
- Cachés multinivel, 83, 192
- Canales de memoria principal independientes, 210
- CAS (Column Address Strobe), 101
- CDB (Common Data Bus), 146, 174
- Cerrojo, 273
- Chipset
  - norte, 117
  - sur, 118
- CISC (Complex Instruction Set Computer), 2, 299
- Cloud, 283, 287
- Cluster, 283
  - Beowulf, 284
  - Dedicado, 283
  - No Dedicado, 283
- CMT (Coarse MultiThreading), 181, 183
- Codificación
  - de longitud fija, 12
  - de longitud variable, 12
  - híbrida, 12
- Código de Hamming, 219
- Códigos de identificación, 115
- Coherencia, 78, 237, 257, 275
- Colisiones, 246
- Commit, 173
- Comparación de rendimiento, 300
- Compare&Swap, 272
- Comunicación
  - explícita, 240
  - implícita, 237
- Conexión punto a punto, 215, 245
- Conjunto de aplicaciones benchmark, 311
- Comutación, 249
  - de circuitos, 249
  - de paquetes, 250
  - vermiforme, 251
  - virtual cut-through, 250
- Consignas, 145
- Consistencia, 237, 255, 269, 275, 298
- Consumo de potencia, 300, 313
- Contador
  - de programa, 20
  - de programa de excepción o EPC, 34
- Control de escrituras, 147
- Control
  - de flujo, 253
  - de flujo asíncrono, 253
  - de flujo basado en créditos, 253
  - de flujo de comutación, 253
  - de flujo físico, 253
  - de flujo STOP&GO, 253
- Controlador
  - de DMA (DMAC), 126
  - de interrupciones o PIC, 123
  - de memoria, 101, 117
- CPI, 299
- Crecimiento progresivo, 284
- Cuenta de eventos, 308
- 
- DDR, 103
- DDR2 y DDR3, 103
- Decode (D), 15
- Decodificación, 139
  - multinivel, 23
- Descubrimiento de servicios, 287
- Desenrollado de bucles, 168
- Detección de eventos, 308
- Dirección
  - física, 79, 86, 87
  - virtual, 79
- Direccionamiento
  - indirecto con registro, 12
  - relativo al PC, 12
- Direcciones de memoria, 7
- Directorio, 275
  - limitado, 282
- Disponibilidad, 112, 218, 284, 313
- DMA
  - Físico, 127
  - modo ráfaga, 127
  - modo robo de ciclo, 127
  - Virtual, 127
- DRAM
  - convencional, 100, 101
  - síncrona, 103
- DSP, 242
- Dual Channel, 210
- 
- E/S
  - aislada, 111
  - mapeada en memoria, 111
- Ecuación de prestaciones, 18
- Eficiencia, 313, 314
- Ejecución de instrucciones fuera de orden, 139
- Emisión

- de instrucciones, 139
- múltiple de instrucciones, 167
- Emplazamiento**
  - asociativo, 85
  - asociativo por conjuntos, 85
  - directo, 85
- Encaminamiento, 252
- Enmascaramiento selectivo, 123
- Equilibrio de carga, 286
- Escalabilidad, 284, 315
- Escena 3D, 224
- Escritura Directa, 88
- Especulación, 173
- Espera de respuesta, 120
- Estabilidad, 313
- Estaciones de reserva, 145, 147
- Estado**
  - de las instrucciones, 142
  - de las unidades funcionales, 142
  - del banco de registros, 142
- Estrategia de escalado, 316
- Etiquetas o tags, 82
- Evaluación
  - de prestaciones de un procesador, 18
  - de rendimiento, 298
- Eventos globales, 257, 277
- Excepciones
  - imprecisas, 63
  - precisas, 64
- Exclusión mutua, 273
- Execution (X), 15
- Extended Data Out DRAM o EDO DRAM, 102
- Extensiones multimedia, 14
- Facilidad de medida, 298
- Fallo de página, 105
- Fallos
  - de capacidad, 79
  - iniciales, 79
  - por conflicto, 79
  - prioritarios, 193
- Fetch (F), 15
- Fetch&Op, 272
- Fiabilidad, 112, 298, 313
- Flexibilidad, 284, 313
- Flit (Flow control unit), 245
- FPM DRAM (Fast Page Mode DRAM), 101
- Fracción serie, 305
- Fragmentación, 105
  - externa, 105
  - interna, 105
- Frecuencia
  - de funcionamiento, 111
  - de operación del bus, 212
- de reloj, 299
- Front-End, 139
- Funct, 20
- Ganancia o Speedup, 19, 302
- Geometría, 224
- Gestión del sistema de E/S, 110, 120
- Globus, 287
- GPU o Graphic Processing Unit, 223
- Grid, 286, 287
- Hiperpaginación (trashing), 105
- Hipersegmentación, 299
- Hipertransporte, 215, 216
- I, 18
- ICH, 118
- IDE/ATA, 118
- Identificación**
  - hardware por vectores, 122
  - software por encuesta (polling), 122
- Inclusión, 77
- Información mantenida por el sistema operativo, 308
- Instrucciones**
  - de check, 181
  - de test, 181
- Interfaz, 110, 244
- Interrupciones, 120
  - multinivel, 123
- Itanium (arquitecturas EPIC), 167
- Jerarquía**
  - de buses, 117
  - de memoria, 76
- Latencia, 81, 112, 243, 300
- Lectura de operandos (LO), 139
- Ley de**
  - Amdhal, 303
  - Gustafson, 306
- Linealidad, 298
- Líneas de identificación, 115
- Little Endian, 7
- Load Linked&Store Conditional, 273
- Localidad**
  - espacial, 78, 199
  - temporal, 78, 194
- Lock, 212, 273
- Marcos, 82
- MCH, 118
- Memoria**
  - caché, 76, 82, 192
  - caché con asignación en escritura, 95

- caché direccionaladas con las direcciones virtuales, 200
- caché físicamente accedida, pero virtualmente indexada, 202
- caché sin asignación en escritura, 95
- centralizada, 237
- principal, 76, 100, 206
- principal con palabra ensanchada, 207
- principal entrelazada, 207
- principal segmentada, 211
- simétrica, 237
- virtual, 76, 104
- virtual paginada, 105
- virtual paginada/segmentada, 105
- Memory Access (M)**, 15
- Mensaje**, 243
- MESI**, 263
- Método de la Pizarra**, 141, 148
- Métricas de rendimiento**, 298
- MFLOPS** (millones de instrucciones en coma flotante por segundo), 299
- Microbenchmark**, 311
- Microinstrucción**, 30, 31
- Microprograma**, 30
- Middle-Endian**, 7
- Middleware**, 239, 287, 289
- MIMD** (Multiple Instruction Multiple Data), 236
- MIPS**  
(millones de instrucciones por segundo), 299
- MIPS64**, 16
- MISD** (Multiple Instruction Single data), 236
- MMX**, 15
- Modelo**
  - de consistencia, 256, 269
  - de consistencia de ordenación débil, 271
  - de consistencia secuencial, 269
  - relajado de consistencia, 270
- Modo**
  - de direccionamiento de registro, 10
  - de direccionamiento directo o absoluto, 10
  - de direccionamiento indirecto con desplazamiento, 11
  - de direccionamiento inmediato, 9
- Modos de direccionamiento**, 7
- Monitores**
  - hardware, 309
  - software, 308
- MPI** (Message Passing Interface), 285
- MPoC**, MultiProcessor on Chip, 240
- MSI**, 259
- MTTDL**, 300
- MTTF**, 300
- Muestreo**, 309
- Multicomputador**, 240, 313
- Multifuncional**, 58, 59
- Multiprocesador**, 237, 313
- Multithreading**, 181
- N.<sup>o</sup> de transferencias/ciclo**, 111
- NanoMips**, 15, 20
  - con especulación, 175
  - monociclo, 21, 22, 23, 24
  - multiciclo, 26, 28, 29, 30, 35
  - multifuncional, 59, 62
  - segmentado, 37, 38
  - superescalar estático, 171
- Network On Chip**, NoC, 242
- No lineal**, 59
- Nodo**
  - local, 276
  - propietario, 276
  - remoto, 277
- NOW** (Network Of Workstations), 283
- NPU** o Network Processing Unit, 223
- NUMA** (Non Uniform Memory Access), 238
- ODR** (Octal Data Rate), 104
- Opcode**, 20
- Operación de E/S**, 110
- Organizaciones virtuales (VO)**, 286
- Página**, 104
- Palabra crítica primero**, 97
- Paralelismo**
  - a nivel de instrucción (ILP), 36, 139, 181
  - a nivel de proceso, 36
  - de datos, 236
  - funcional: SISD (Single Instruction Single Data), 236
- Paridad**, 219
- PCI Express**, 119, 215
- PCSrc multiciclo**, 28
- PCWrite multiciclo**, 28
- PCWriteCond multiciclo**, 28
- Peer2Peer**, 283
- Penalización**
  - por fallo, 192
  - por fallo en MC, 81
- Petición de interrupción (IRQ)**, 121
- Phit (PHysical unIT)**, 245
- Pila de dirección de retorno**, 160
- Planificación**
  - de instrucciones de acceso a memoria, 139
  - dinámica de instrucciones, 138
  - dinámica de instrucciones centralizada, 139, 141
  - dinámica de instrucciones distribuida, 139, 145
  - dinámica distribuida, 173
- Política**
  - de emplazamiento, 85
  - de escritura, 88

- Post-Escritura, 88
- Prebúsqueda, 199
- Predecodificación de instrucciones, 139
- Predicción
  - de salto no tomado, 49
  - de salto tomado, 50
  - dinámica de saltos, 150
- Predictor
  - de confianza de saltos, 161
  - de saltos adaptativo, 166
  - de saltos correlado, 162
- Prioridad a los fallos, 193
- Procesador
  - monociclo, 19, 20
  - multiciclo, 19, 25
  - vectorial, 169, 242
- Procesamiento
  - de Texturas y Fragmentos, 225
  - de Vértices, 225
- Productividad del sistema, 300
- Protocolos
  - de arbitraje, 114
  - de coherencia basados en directorio, 275
  - de coherencia de actualización, 258
  - de coherencia de invalidación, 258
  - de sincronización, 113
- Puntos de control, 22
- PWHit, 258
- PWMiss, 258
- RAID, 217
- Rápida respuesta a los avances tecnológicos, 284
- RAS (Raw Address Strobe), 101
- Rasterización, 224
- RAW o Read After Write, 41
- RDRAM, 104
- Rearranque rápido, 97
- Recuperación
  - off line, 220
  - on line, 220
- Red
  - de área Extensa (WAN), 242
  - de área local (LAN), 242
  - de área sistema (SAN), 242
- Redes
  - comutadas o indirectas, 247
  - crossbar, 247
  - de barras cruzadas, 247
  - de Clos, 248
  - de Mariposa, 248
  - de medio compartido, 246
  - dinámicas, 246
  - directas, 245
  - estáticas, 245
- jerárquicas, 247
- Monoetapa, 247
- multibus, 247
- multietapa bloqueante, 248
- Multietapa, 248
- multietapa no bloqueante, 248
- Omega, 248
- Redundancia, 218
- Redundant Array of Inexpensive Disks, 217
- Reemplazamiento
  - aleatorio, 88
  - FIFO, 88
  - LRU, 88
- Registro
  - de futuro, 65
  - de historia, 65
  - de historia global de saltos (GBH o Global Branch History), 162
  - de máscara, 123
- Registros
  - de propósito general (GPR o General Purpose Registers), 3
  - de segmentación, 37
  - intermedios, 26
- Relación coste/prestaciones, 284
- Relleno de ranura, 52
- Renderizado, 224
- Reordenación del código, 47
- Repertorio de instrucciones, 2
- Repetibilidad, 298
- Resolución, 300
- Riesgos
  - de control, 41, 48
  - de datos, 41, 43
  - estructurales, 40, 43
- RISC (Reduced Instruction Set Computer), 2
- Robustez, 313
- Ruta de datos, 20, 25, 37, 48
- Rutina
  - de Tratamiento de Excepción o RTE, 34
  - de Tratamiento de Interrupción (RTI), 121
- Salto retardado, 52
- Scoreboard, 141
- Segmentación, 36
- Segmento, 104
- Sensibilidad, 313
- Señalización diferencial, 213
- SIMD (Single Instruction Multiple Data), 169, 236
- Sincronización, 237, 256, 275
- Sistema de E/S, 108
- SMP (Symmetric MultiProcessing), 181, 237
- SMT, 183
- SoC, System on Chip, 240

- Speedup, 300
  - absoluto, 313
  - normalizado, 313
  - real, 313
  - relativo, 313
- SSE, 15
- Standard Performance Evaluation Corporation (SPEC), 311
- Stripping, 218
- Subinstrucciones, 167
- Superescalar estático, 169
- Swap, 272
- Tabla de historia de saltos, 151
- Tamaño
  - de la caché, 83
  - de la diagonal, 300
  - de marco, 83
- Tasa de fallos, 81, 192
- Taxonomía de Flynn, 236
- Terminación, 173
- Test&Set, 272
- Testigo (token), 247
- Texturas, 224
- Three Channel, 210
- Tiempo
  - de acierto de MC, 81
  - de CPU, 300
  - de ejecución, 18, 300
  - de respuesta, 18, 300
  - medio de acceso a memoria, 81, 192, 300
  - medio hasta fallo de un disco (Mean Time To Failure o MTTF), 221
  - medio hasta pérdida de datos (Mean Time To Data Loss o MTTDL), 221
- Tipos
  - de excepciones, 34
  - de fallos en la memoria caché, 79
- TLB, 200, 203, 204
  - (Translation Lookahead Buffer), 106
- Tolerancia a fallos, 112, 218, 313
- Torres de Hanoi, 311
- Traducción
  - de direcciones asociativa, 106
  - de direcciones directa, 106
  - de direcciones mixta, 106
- Transformada de Fourier rápida, 311
- Tratamiento de excepciones, 33, 63, 181
- Traza, 308
- UMA (Uniform Memory Access), 237
- Unidad
  - de control, 23, 29, 38
  - de control cableada, 29
  - de control como máquina de estados, 29
  - de control global, 23, 28
  - de control microprogramada, 29
- Unidades funcionales segmentadas, 60
- USB, 119
- Vector
  - de compartidos, 276
  - de compartidos por grupos, 282
- Ventana de instrucciones, 139, 141
- Very Long Instruction Word (VLIW, Palabra de Instrucción Muy Larga), 167
- WAR o Write After Read, 41, 139
- WAW o Write After Write, 41, 139
- Web Services, 287
- Write-merging, 194
- Writeback (W), 15
- XDR DRAM (eXtreme Data Rate DRAM), 104
- Z-Buffer, 225





Este libro de texto se centra en el diseño y evaluación de arquitecturas de computadoras que incorporen las técnicas de aumento de prestaciones actuales en el diseño del procesador o procesadores, la jerarquía de memoria y el sistema de E/S.

Se trata de un manual que no ha sido concebido como libro de consulta o de referencia, sino como una herramienta muy potente para el estudio autónomo y/o dirigido de los alumnos de las titulaciones relacionadas con la informática, las tecnologías de la información y las comunicaciones.

Adaptado a los nuevos planes de estudios universitarios, es accesible, ameno y está escrito con un enfoque completamente pedagógico (basado en la utilización de figuras ilustrativas, ejemplos resueltos, casos prácticos, pruebas de autoevaluación, resúmenes de conceptos importantes) que surge de la experiencia de años impartiendo materias en el área de Arquitectura de Computadores de los dos autores del libro. Además, este libro lleva asociado un sitio web con recursos adicionales para estudiantes y profesores.

Desde esta perspectiva didáctica, el libro se estructura en seis capítulos cuyo objetivo es el aprendizaje basado en competencias. Estos capítulos se centran en el estudio del diseño del procesador, la jerarquía de memoria y el sistema de E/S, el aumento de prestaciones del procesador, la mejora de la jerarquía de memoria y del sistema de E/S, el diseño de sistemas multiprocesador y multicomputador y la evaluación del rendimiento de las arquitecturas.

#### Otros libros de interés

Organización y arquitectura de computadoras, 7.<sup>a</sup> ed

William Stallings

PEARSON PRENTICE HALL

ISBN 978-84-896-6082-3



Organización de computadoras:

un enfoque estructurado, 4.<sup>a</sup> ed

Andrew S. Tanenbaum

PEARSON PRENTICE HALL

ISBN 978-97-017-0399-1



**Prentice Hall**  
es un sello editorial de



[www.pearsoneducacion.com](http://www.pearsoneducacion.com)

