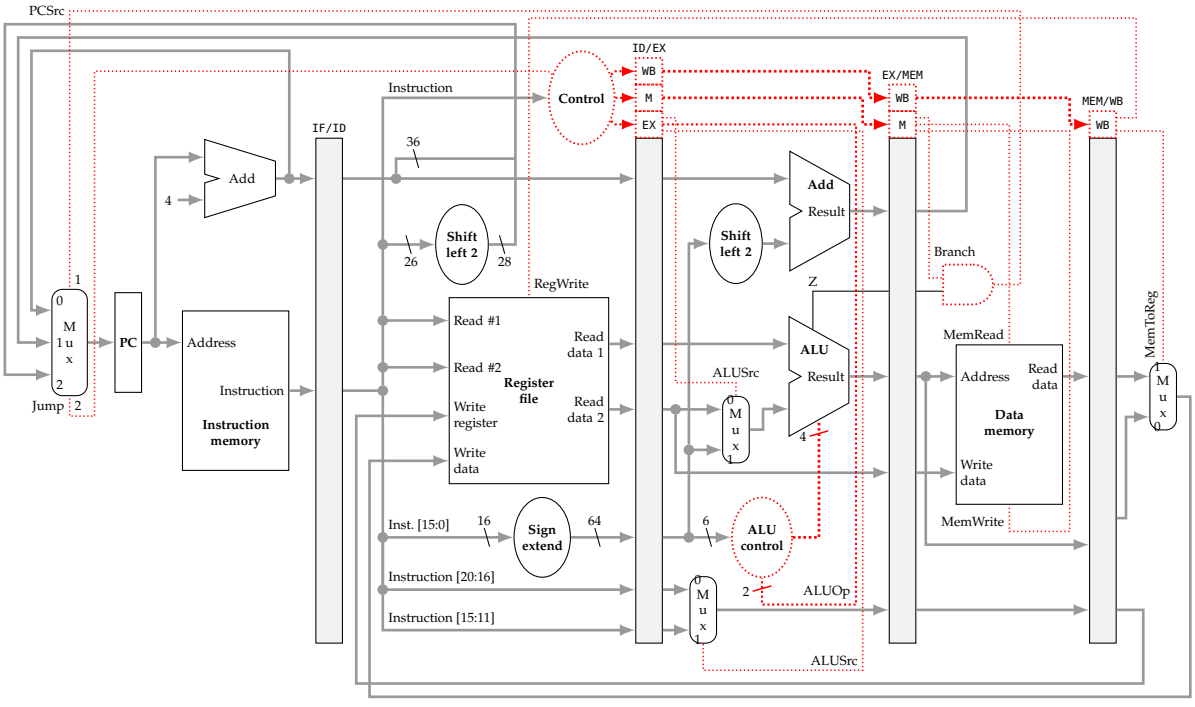
Juan Carlos Granda Candás
José María López López
Manuel García Vázquez
Julio Molleda Meré
Rubén Usamentiaga Fernández
Joaquín Entrialgo Castaño
Francisco Javier de la Calle Herrero

Arquitectura de Computadores



Textos universitarios
ediuno

ARQUITECTURA DE COMPUTADORES

ARQUITECTURA DE COMPUTADORES

Juan Carlos Granda Candás
José María López López
Manuel García Vázquez
Julio Molleda Meré
Rubén Usamentiaga Fernández
Joaquín Entrialgo Castaño
Francisco Javier de la Calle Herrero

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Servicio de Publicaciones de la Universidad de Oviedo
ISNI: 0000 0004 8513 7929
Edificio de Servicios - Campus de Humanidades
33011 Oviedo - Asturias
985 10 95 03 / 985 10 59 56
servipub@uniovi.es
www.publicaciones.uniovi.es

ISBN: 978-84-17445-49-2

Resumen de contenidos

Índice general

Prólogo

Aunque los principios básicos son los mismos, poco tienen que ver los computadores actuales con la idea inicial concebida por John von Neumann en 1945 basada en los trabajos de John Presper Eckert y John William Mauchly en el ENIAC.

Los computadores han evolucionado a máquinas con un formidable rendimiento que los hace imprescindibles en muchos campos. De hecho, en los últimos tiempos hemos visto cómo el aspecto tradicional en el que se presentaban los computadores ha cambiado para hacerse cada vez más ubicuos. Es posible encontrar computadores en infinidad de formas, como por ejemplo equipos portátiles, dispositivos móviles, sistemas empotrados, etc.

En este libro los autores pretendemos ilustrar cómo los principios de la arquitectura von Neumann se aplican en los computadores actuales, y cuáles son las mejoras que estos incorporan para mejorar el rendimiento, así como la influencia que ejercen los sistemas operativos multitarea modernos y el necesario soporte a la virtualización sobre el hardware subyacente.

Gijón, 19 de julio de 2022.

Los autores.

Capítulo 1

Introducción

En este capítulo se presenta una introducción a la arquitectura del computador suponiendo unos conocimientos básicos sobre los fundamentos de los computadores. A partir de este punto, se aborda el rendimiento del computador como un factor crucial en el diseño de las arquitecturas de computadores. En ocasiones el rendimiento está directamente relacionado con el coste, pero no siempre un computador de mayor coste ofrece un rendimiento superior a otro de menor coste. Es por tanto necesario poder evaluar el rendimiento que ofrece un computador de forma objetiva para así poder comparar.

El capítulo aborda el problema de evaluar el rendimiento de un computador, al tiempo que se sientan las bases necesarias para comprender las mejoras del rendimiento que se plantearán a lo largo del presente libro en los diferentes componentes del computador.

1.1. El computador

Antes de introducir aspectos avanzados del computador tales como su arquitectura y cómo medir el rendimiento, es conveniente revisar algunos conceptos básicos. Dentro del primer apartado dedicado a la estructura básica del computador se muestra su estructura y los grandes bloques que lo constituyen. A continuación, se describen los diferentes niveles de abstracción que facilitan la construcción y programación de una máquina tan compleja como es el computador. Finalmente, se presentan los principios básicos que guían el desarrollo de los computadores.

1.1.1. Estructura básica

Un computador se define como una máquina que es capaz de recibir información de entrada, procesarla bajo el control de un programa y generar información a través de medios de salida, todo ello sin intervención de un operador humano.

Aunque habitualmente la imagen de un computador es una máquina con diferentes periféricos conectados como un teclado, un ratón o un monitor, la realidad es

que hoy en día existen computadores de muchos tipos, que van desde el típico ordenador personal o portátil a tabletas, teléfonos móviles e incluso sistemas empotrados como los que se encuentran en vehículos y electrodomésticos.

La construcción de un computador capaz de ejecutar programas plantea una serie de necesidades:

- Almacenar el programa a ejecutar. El programa a ejecutar está formado por instrucciones que deben almacenarse en algún tipo de contenedor dentro del computador.
- Almacenar los datos del programa. Los datos de entrada y salida del programa deben almacenarse también en un contenedor dentro del computador.
- Mecanismo de lectura y ejecución del programa. Debe existir un mecanismo que lea las instrucciones del programa almacenado y las ejecute sobre los datos de entrada para generar los datos de salida.
- Mecanismo de entrada/salida. Tanto el programa a ejecutar como los datos con los que trabaja deben ser introducidos al computador de alguna forma, ya sea por un usuario o por otro sistema externo. Igualmente, debe ser posible mostrar los datos de salida al usuario o enviarlos a otro sistema.

Tal como ocurre en otros campos de la ingeniería, el diseño de un computador debe seguir unos requisitos y cumplir una serie de restricciones de rendimiento, coste y consumo de energía. Por tanto, el diseño de un computador está guiado habitualmente por soluciones de compromiso entre estos factores. Aunque existen varios posibles diseños para los computadores, todos ellos tienen en común tres componentes: la memoria, la CPU y los periféricos, tal como se plantea en la denominada arquitectura von Neumann (figura 1.1). La memoria constituye el contenedor donde se almacenan en forma de secuencias de bits tanto los programas como los datos que utilizan, los periféricos permiten introducir y extraer los datos, y la CPU lee los programas y los ejecuta. En esta última destacan dos elementos:

- Unidad de control: interpreta las instrucciones del programa y genera las señales de control necesarias para la ejecución de las mismas.
- Unidad aritmético-lógica (ALU): se utiliza para operar sobre datos binarios.

Existen también una serie de buses que interconectan dichos elementos. Los buses son canales de comunicación compartidos:

- Bus de direcciones. A través de este bus la CPU indica a la memoria o a los periféricos la localización de la información a leer o escribir.
- Bus de datos. Sobre este bus se transmite la información entre los diferentes elementos del computador.
- Bus de control. Contiene líneas que gestionan la transmisión de información dentro del computador.

Existe una variante de la arquitectura von Neumann, denominada arquitectura Harvard, que dispone de una memoria específica para almacenar datos y otra para almacenar el código de los programas.

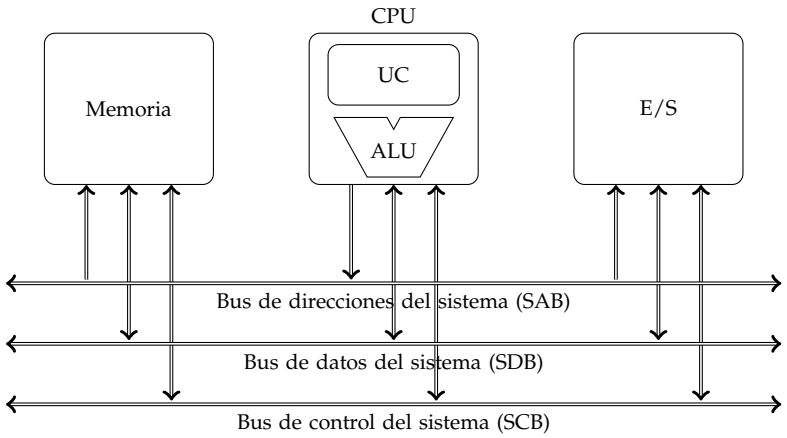


Figura 1.1: Arquitectura von Neumann

1.1.2. Máquina multi-nivel

Un desafío que deben afrontar los computadores de hoy en día es la «distancia conceptual» que existe entre el lenguaje que entiende el computador —el lenguaje de la máquina, codificado en una secuencia de bits— y el lenguaje que utilizan los programadores para implementar sus algoritmos. Esta distancia es cada vez mayor debido a dos factores:

- Al programador le interesa utilizar un lenguaje próximo al lenguaje natural para describir las soluciones a los problemas a resolver, con lo que se simplifica el desarrollo de los programas.
- Interesa que el lenguaje del computador sea lo más sencillo posible para poder utilizar el menor número de recursos hardware y al mismo tiempo que sea altamente eficiente.

Para lidiar con este problema es común definir un nuevo conjunto de instrucciones más próximo al lenguaje natural, L1, a partir del lenguaje que utiliza la máquina, L0. De esta forma, el lenguaje L1 se define a partir de las instrucciones disponibles en el lenguaje L0. Este nuevo lenguaje da lugar a una máquina virtual, M1, construida a partir de la máquina física, M0, que puede ser utilizada por los programadores para desarrollar programas en el lenguaje L1.

Para ejecutar programas escritos en el lenguaje L1 sobre la máquina M0 (que utiliza el lenguaje L0) son posibles dos técnicas no excluyentes:

- Traducción: Antes de ejecutar el programa L1, este debe traducirse completamente a instrucciones en el lenguaje L0.
- Interpretación: El programa L1 sirve como entrada a un programa escrito en L0 (intérprete) que se encarga de ejecutar las instrucciones L1 sobre la máquina M0 sin necesidad de una traducción previa.

Si bien la definición de un nuevo lenguaje permite simplificar el desarrollo de programas por parte de los programadores, la distancia entre los lenguajes L1 y L0 no debe ser excesiva para que sea posible realizar la traducción y/o interpretación, con lo que el lenguaje L1 todavía sigue estando lejos del lenguaje natural. Para solucionar este problema, se pueden definir más niveles para aproximarse progresivamente al lenguaje natural. Hoy en día, los computadores suelen definir un conjunto de niveles comunes:

- Nivel de lógica digital (nivel 0): Formado por puertas lógicas, constituye el hardware del computador.
- Nivel de microarquitectura (nivel 1): Este nivel está formado por el camino de datos y la unidad de control. En algunas máquinas las operaciones sobre el camino de datos se controlan mediante una unidad de control microprogramada, mientras que en otras se utilizan unidades de control cableadas. En el primer caso, el microprograma es el intérprete de las instrucciones del nivel 2. La tendencia hoy en día es a eliminar este microprograma o, al menos, seguir una aproximación híbrida donde la parte cableada de la unidad de control se encarga de las instrucciones del nivel 2 más sencillas (habitualmente las más comunes) y la parte microprogramada se encarga de las más complejas (las menos comunes).
- Nivel de arquitectura del juego de instrucciones (nivel 2): Especificación, determinada por el fabricante del procesador, del lenguaje que entiende la CPU, los registros disponibles, los modos de direccionamiento, etc.
- Nivel de sistema operativo (nivel 3): Es un nivel híbrido donde muchas instrucciones son del nivel 2, y otras se interpretan (llamadas a servicios del sistema operativo). Esto quiere decir que habrá instrucciones que directamente llegan al nivel 2 y otras que se sustituyen por rutinas del sistema operativo que ofrecen servicios a los programas de usuario. El sistema operativo proporciona una capa de abstracción de los niveles inferiores, de forma que los programas escritos para un sistema operativo puedan ejecutarse abstrayéndose del hardware subyacente.
- Nivel de lenguaje ensamblador (nivel 4): Es el nivel más bajo en el que pueden escribirse los programas de usuario. Es el primer lenguaje simbólico y habitualmente es traducido (compilado) al nivel 3. En este nivel 4 pueden definirse abstracciones y nuevos tipos de datos que no existen en el nivel 3.
- Nivel 5 y superiores: Son niveles que definen lenguajes de alto nivel, por ejemplo C o C++. Estos lenguajes permiten la compatibilidad de los programas entre varios sistemas operativos. Es posible escribir en un lenguaje de alto nivel programas que se ejecuten sobre varios sistemas operativos. No obstante, si el lenguaje no es interpretado puede ser necesario recompilar los programas de un sistema operativo a otro.

La figura 1.2 resume los niveles que habitualmente se definen en el computador de hoy en día y cuál es el método más habitual para ejecutar los programas definidos en una máquina virtual sobre la máquina inmediatamente inferior.

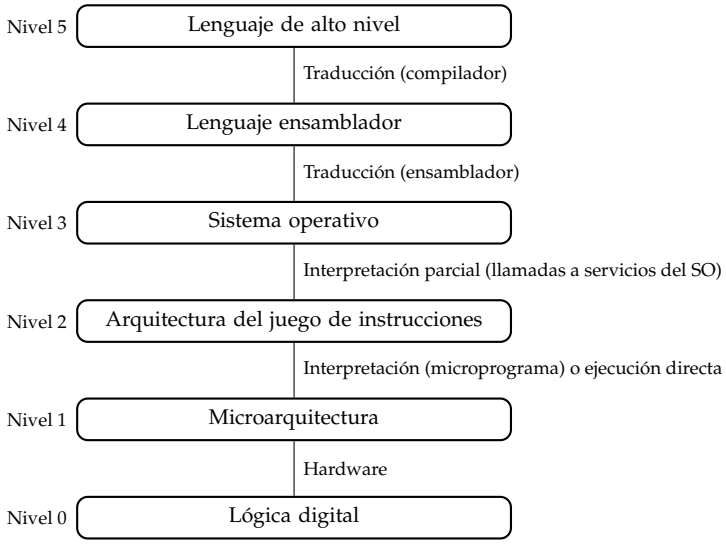


Figura 1.2: Niveles virtuales del computador actual

1.1.3. Principios básicos de diseño

En el diseño de los computadores actuales subyacen varias ideas que se aplican a todos los ámbitos de la arquitectura de los computadores. Estas ideas se han venido siguiendo casi desde los primeros computadores y han ayudado al tremendo desarrollo que han experimentado a lo largo de los años.

Los principios básicos que rigen el diseño de las arquitecturas de los computadores son:

- Diseñar para el cambio rápido. Las tecnologías de fabricación de computadores han evolucionado y siguen evolucionando, con capacidades de integración cada vez mayores, lo que permite disponer de más puertas lógicas para implementar los componentes del computador. Cuando se diseña una nueva arquitectura debe hacerse teniendo en cuenta que debe ser capaz de sacar provecho de las capacidades cada vez mayores que proporcionará la tecnología para evitar quedarse obsoleta.

- Uso de abstracciones. El diseño de un computador es una tarea extremadamente compleja, muy difícil de abordar en su conjunto. Por esta razón, es útil utilizar abstracciones que simplifiquen el diseño y oculten detalles de bajo nivel que no sean relevantes para la resolución de un problema concreto. El diseño actual de las máquinas multi-nivel, tal como se ha visto, es un ejemplo del uso de abstracciones.

- Optimizar el caso más común. Tal como determina la ley de Amdahl, que se abordará en la sección 1.4.2, la mejora del rendimiento del computador debido a una mejora en el rendimiento de un elemento particular estará limitada por

la fracción del tiempo en que se usa este elemento. Por esta razón, deben plantearse diseños que optimicen los casos más comunes (por ejemplo la ejecución de instrucciones comunes) frente a los casos menos comunes.

- Mejora del rendimiento con paralelismo. Siempre que sea posible debe favorecerse el uso del paralelismo para incrementar el rendimiento. Las tecnologías de fabricación tienen unas limitaciones físicas que impiden que un dispositivo pueda trabajar por encima de una determinada velocidad. Si el computador se diseña permitiendo que varios de esos dispositivos puedan trabajar en paralelo se consigue mejorar el rendimiento.
- Segmentación de tareas (*pipelining*). Se trata de una técnica de paralelismo empleada en la inmensa mayoría de las CPU actuales. Por su importancia se trata en detalle en el capítulo 2.
- Mejora del rendimiento mediante predicción. En ocasiones se puede mejorar el rendimiento comenzando una tarea suponiendo que será necesaria en lugar de esperar a saberlo con certeza. Si la predicción es correcta se habrá ganado un tiempo, mientras que si fue incorrecta habrá que deshacer el trabajo hecho. La clave está en acertar en las predicciones la mayor parte de las ocasiones.
- Jerarquía de memoria. Es un concepto clave en el diseño del sistema de memoria para combinar diferentes tecnologías y extraer lo mejor de cada una de ellas que será tratado en el capítulo 3.
- Fiabilidad a través de redundancia. Si bien durante este texto no se abordarán cuestiones de fiabilidad de los computadores, una práctica habitual es alcanzarla replicando componentes.

1.2. Arquitectura del juego de instrucciones

El diseño de un computador supone definir cuál es su arquitectura, que en gran medida se refiere a la arquitectura de la CPU que lo gobierna. Esta arquitectura comprende aspectos tales como qué elementos componen el computador y cuál es su comportamiento e interacciones, cómo se interconectan estos elementos o qué mecanismos de entrada/salida utiliza para comunicarse.

Hoy en día, la arquitectura de una CPU, y por extensión la del computador, suele referirse a la arquitectura del juego de instrucciones (o ISA de su acrónimo en inglés). La definición de la **arquitectura del juego de instrucciones** de una CPU es la decisión más crítica de diseño, ya que determina todos aquellos aspectos visibles al programador[1]. Incluye, entre otras cosas, el tipo de datos que maneja la CPU (enteros, números reales, cadenas, etc.), los registros que están a disposición del programador, el juego de instrucciones disponible, o cuál es el formato de las direcciones y los modos de direccionamiento.

Las arquitecturas del juego de instrucciones se han clasificado históricamente en dos grandes grupos: las arquitecturas RISC (*Reduced Instruction Set Computer*) y CISC

[1]Programador a bajo nivel, desarrollador de compiladores o programador de sistemas.

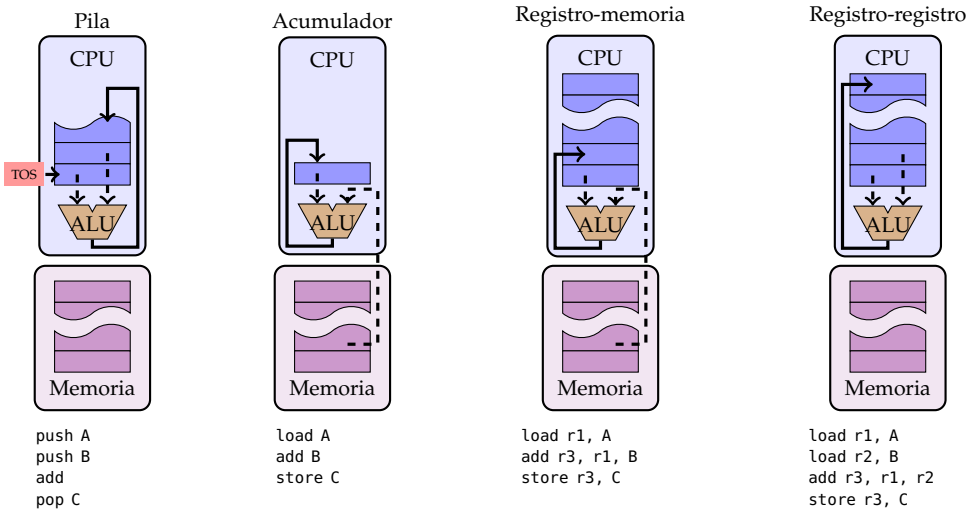


Figura 1.3: Tipos de modelo de máquina

(*Complex Instruction Set Computer*). Las arquitecturas RISC definen instrucciones muy simples que pueden ejecutarse rápidamente con muy poco hardware. Algunos ejemplos de este tipo de arquitecturas son: ARM (muy extendida en dispositivos móviles), MIPS, IA-64 y PowerPC. En cambio, las arquitecturas CISC definen instrucciones complejas que pueden realizar operaciones de alto nivel, facilitando la labor del programador. Ejemplos de arquitecturas CISC incluyen x86 y Motorola 68 000.

La definición del juego de instrucciones de una CPU puede responder a cuatro modelos de máquina distintos, ilustrados en la figura 1.3.

- Pila. En una máquina de pila la CPU dispone de un conjunto de registros organizados en forma de pila y de un registro especial denominado cima de la pila (`TOS`, *top of stack*). Los operandos de las instrucciones son implícitos y se obtienen de la pila de registros, empezando por el apuntado por el registro `TOS`. El resultado de la operación se almacena también en la cima de la pila de registros. Por ejemplo, una operación de asignación de la suma de dos operandos podría tener la siguiente forma:

  ```
  push A
  push B
  add
  pop  C
  ```

  Inicialmente se cargan en la pila de registros los valores de las variables `A` y `B`, tras lo cual se ejecuta la instrucción de suma. Esta instrucción extrae de la pila los dos últimos valores, los suma y almacena el resultado en la misma pila. La instrucción `pop` extrae el valor apuntado por el registro `TOS` y lo almacena en la variable `C`. La ventaja de una CPU basada en un modelo de máquina de pila es un menor número de bits necesario para codificar las instrucciones, ya que no se necesitan bits del código de instrucción para indicar los operandos a utilizar.

- Acumulador. En una máquina de tipo acumulador existe un registro especial, denominado precisamente registro acumulador, que se utiliza implícitamente como operando. El resultado de una operación se almacena en el registro acumulador. De esta forma, una operación de asignación de una suma podría tener la siguiente forma:

```
load  A
add   B
store C
```

  La primera instrucción carga en el registro acumulador el valor de la variable `A`. A continuación, se puede realizar la operación de suma, indicando la variable cuyo valor va a sumarse con el valor del registro acumulador. El resultado, almacenado en el registro acumulador, se guarda en la variable `C` con la última instrucción.

- Registro-memoria. En una máquina de tipo registro-memoria un operando puede estar almacenado en uno de los registros que incorpora la CPU y el otro directamente en memoria. El resultado de la operación se almacena en otro de los registros de la CPU. Bajo estas condiciones, una operación de asignación de una suma podría tener la siguiente forma:

```
load  r1, A
add   r3, r1, B
store r3, C
```

  La primera instrucción carga en el registro `r1` el valor de la variable `A`. A continuación, se realiza la suma entre el valor almacenado en este registro y la variable `B`, almacenada en memoria. El resultado se almacena en el registro `r3`. Finalmente, el resultado de la suma puede almacenarse en memoria en la variable `C` desde el registro `r3`.

- Registro-registro o más comúnmente conocida como carga/almacenamiento. En una máquina de tipo carga/almacenamiento todos los operandos con los que trabaja una instrucción deben estar en los registros de la CPU. La misma operación de asignación de una suma se realizaría de la siguiente manera en una máquina de este tipo:

```
load  r1, A
load  r2, B
add   r3, r1, r2
store r3, C
```

  Las primeras instrucciones cargan en los registros `r1` y `r2` los operandos. La instrucción de suma debe indicar explícitamente cuáles son los registros que contienen los operandos y en cuál se almacenará el resultado. Tras realizar la operación, el resultado puede almacenarse en memoria en la variable `C` desde el registro `r3`. La particularidad de este modelo de máquina es que únicamente acceden a memoria las instrucciones de carga (`load`) y almacenamiento (`store`).

Hoy en día ARM es la arquitectura más empleada, pues domina el mundo de los dispositivos móviles. Esta arquitectura se inspiró en la arquitectura MIPS64, que se

empleará a lo largo del libro a modo de ejemplo. Esta arquitectura sigue un modelo de máquina de carga/almacenamiento.

1.3. Microarquitectura

La microarquitectura se refiere a la organización interna de la CPU, no visible al programador. Para implementar una determinada arquitectura del juego de instrucciones es posible definir varias microarquitecturas. Por ejemplo, Nehalem (Core i5) y AMD K10 (Athlon II) son microarquitecturas de Intel y AMD respectivamente que implementan la misma arquitectura del juego de instrucciones (x86).

La microarquitectura define aspectos tales como el número de unidades funcionales dentro de la CPU, por ejemplo el número de unidades aritmético-lógicas, el número y tipo de cachés o el ancho de los buses de memoria. También determina el tipo de organización, dando lugar a distintos tipos de CPU: segmentadas, superescalares o multinúcleo, entre otras.

El objetivo primordial de una microarquitectura es aprovechar al máximo la tecnología de fabricación disponible para proporcionar el máximo rendimiento al implementar una arquitectura del juego de instrucciones dada.

En el capítulo 2 se proponen diferentes microarquitecturas para la implementación de una versión reducida de la arquitectura MIPS64.

1.4. Rendimiento

En los apartados anteriores se trató la arquitectura del juego de instrucciones del computador, que está ligada a su funcionalidad, es decir, a las instrucciones que puede ejecutar. Sin embargo, el rendimiento está ligado a la cantidad de instrucciones que el computador puede ejecutar en un determinado tiempo, que depende en gran medida de su microarquitectura.

En este apartado se introduce el concepto de rendimiento y cómo se mide en un computador.

1.4.1. Concepto de rendimiento

Un computador es un sistema muy complejo en el que se interrelacionan muchos elementos. Resulta complicado delimitar el concepto de rendimiento de un computador, ya que depende del punto de vista del usuario del computador y del objetivo perseguido con el funcionamiento del mismo. Así, por ejemplo, desde el punto de vista del usuario interesa el tiempo empleado en ejecutar una tarea. En cambio, como administrador del sistema interesa la cantidad de tareas que ejecuta por unidad de tiempo. Por esta razón, es habitual utilizar diferentes formas de medir el rendimiento a través de lo que se conocen como métricas.

Una **métrica** es una magnitud que cuantifica un aspecto medible de un sistema. Los resultados obtenidos por dos computadores para una métrica permitirán compararlos. Existen distintos tipos de métricas dependiendo de la propiedad del sistema a medir.

Comúnmente el rendimiento del computador está asociado a la capacidad de realizar trabajo. Para cuantificar el rendimiento del computador se utilizan principalmente dos métricas: tiempo de respuesta y productividad. El **tiempo de respuesta** se refiere al tiempo que invierte el computador en realizar una tarea, mientras que la **productividad** indica el número de tareas completadas por unidad de tiempo.

$$\text{Productividad} = \frac{\text{Tareas completadas}}{\text{Tiempo de referencia}}$$

Para que tanto la productividad como el tiempo de respuesta tengan sentido es necesario definir claramente la tarea sobre la que se miden. Por ejemplo, la tarea puede referirse a la ejecución de un programa completo, la ejecución de una única instrucción, servir una página web, responder a una consulta a una base de datos, etc. De esta forma, el tiempo de respuesta se refiere al tiempo desde que se inicia la tarea hasta que termina (el programa, la instrucción, la página web o la consulta), mientras que la productividad se refiere al número de tareas (programas, instrucciones, páginas o consultas) que se completan por unidad de tiempo.

La métrica de productividad es la que suele emplearse de forma más general para hacer referencia al rendimiento del computador. En el caso particular de que un sistema solamente sea capaz de realizar una tarea en cada instante, la productividad y el tiempo de respuesta están relacionados de forma inversa. Por ejemplo, si el tiempo de respuesta medio de una tarea es de 1 ms, podrían llevarse a cabo $(1/0.001\,\text{s}) = 1000$ tareas idénticas por segundo. Sin embargo, esta relación no se mantiene cuando existe la posibilidad de realizar más de una tarea simultáneamente.

Esta diferencia de comportamiento puede observarse cuando se realizan mejoras para aumentar el rendimiento del computador. Así, por ejemplo, si se parte de un procesador que ejecuta instrucciones de forma secuencial y cuya frecuencia de trabajo es de 1 GHz, el tiempo empleado en ejecutar una instrucción será de t segundos. Se plantean dos posibles mejoras:

- Sustituir el procesador por otro con idéntica microarquitectura, pero que trabaje al doble de frecuencia, es decir, 2 GHz.
- Sustituir el procesador por otro que incorpore dos núcleos (dos CPU) con la misma microarquitectura que la inicial e idéntica frecuencia de trabajo.

El resultado obtenido con estas mejoras será el siguiente:

- En el primer caso, el procesador, al tener una frecuencia doble, será capaz de ejecutar las instrucciones en la mitad de tiempo, luego el tiempo empleado en ejecutar una instrucción será $\frac{t}{2}$ segundos. El tiempo de respuesta se reduce a la mitad y en el mismo tiempo se ejecutarán el doble de instrucciones, es decir, la productividad se duplica.
- En el segundo caso, al existir dos núcleos capaces de ejecutar instrucciones simultáneamente, la productividad del nuevo sistema se duplica también. En cambio cada uno de los núcleos no ha sido modificado respecto al sistema inicial, luego el tiempo empleado en ejecutar la instrucción se mantiene, es decir, seguirá siendo t segundos.

A partir de este ejemplo se puede ver cómo afectan los factores sobre los que se actúa para aumentar el rendimiento. Por un lado, se puede disminuir el tiempo de respuesta, que implicará también un aumento de la productividad. Para conseguir este objetivo, es necesario incorporar mejoras tecnológicas u organizativas. Por otro lado, se puede incrementar el nivel de paralelismo de los distintos componentes. Esto consigue aumentar la productividad, aunque en general no consigue disminuir el tiempo de respuesta, pudiendo llegar incluso a aumentarlo.

Si el computador es capaz de realizar varias tareas de forma simultánea, lo que es muy habitual, se utiliza como métrica de rendimiento, en general, la productividad.

El objetivo final de las métricas es poder comparar el rendimiento de los computadores. A modo de ejemplo, supóngase un computador A que es capaz de completar 30 tareas en un segundo, mientras que otro computador B es capaz de completar 20 tareas en un segundo. ¿Cuántas veces el computador A es más rápido que el computador B, es decir, cuál es la aceleración del rendimiento (*speedup*) del computador A respecto al computador B?

$$\text{Aceleración} = \frac{\text{Rendimiento}_\text{A}}{\text{Rendimiento}_\text{B}} = \frac{\text{Productividad}_\text{A}}{\text{Productividad}_\text{B}} = \frac{30}{20} = 1.5$$

La aceleración en el rendimiento es la ratio que existe entre el rendimiento de uno y otro computador. Se dice que el computador A tiene una aceleración de 1.5. Una aceleración igual a 1 indica que el rendimiento de ambos computadores es el mismo. Si la aceleración es mayor que 1, indica que el computador A tiene mayor rendimiento que el computador B y viceversa si la aceleración es menor que 1.

El término tiempo de respuesta suele utilizarse cuando se evalúa el rendimiento de la CPU ejecutando programas. No obstante, cuando se analiza el rendimiento de sistemas de memoria o interconexión se utilizan más a menudo los términos **latencia** (por tiempo de respuesta), referido al tiempo que tarda el sistema en proporcionar los datos solicitados, y **ancho de banda** (por productividad), referido a la cantidad de información que puede proporcionar por unidad de tiempo.

Aunque hasta ahora se ha considerado que el tiempo de respuesta y la productividad son medidas constantes, en la mayor parte de las ocasiones deben aproximarse a través de variables aleatorias, pues varían entre unas mediciones y otras dependiendo del momento de la medición y de la carga del computador en ese instante. De esta forma, por ejemplo, en lugar de tomar un tiempo de respuesta fijo, se aproxima a través de una variable aleatoria, que habitualmente se considera normalmente distribuida $\sim \mathcal{N}(\mu, \sigma^2)$, categorizada por un valor medio y una desviación típica.

Por esta razón, cobra especial importancia la estadística cuando se mide el rendimiento de los computadores. Por poner un ejemplo, si se mide 10 veces (n) el tiempo de respuesta en la ejecución de un programa en un computador y se obtienen los siguientes resultados:

Medida	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
T. resp (s)	3.2	2.9	3.1	3.0	2.8	3.1	3.2	3.0	3.3	2.7

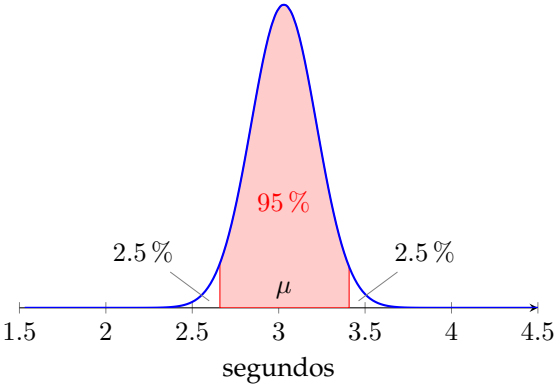


Figura 1.4: Función de densidad de probabilidad del tiempo de respuesta

se puede calcular la media ($\overline{x}$) y la desviación típica (s) de las mediciones tomadas, es decir, la media y la desviación típica muestrales:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{3.2 + 2.9 + 3.1 + 3.0 + 2.8 + 3.1 + 3.2 + 3.0 + 3.3 + 2.7}{10} = 3.03\,\text{s}$$

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}} = 0.189\,\text{s}$$

Para estimar la media poblacional (μ) usamos como estimador la media muestral ($\overline{x}$), mientras que la desviación típica muestral (s) se usa como estimador de la desviación típica poblacional (σ). Asumiendo que el tiempo de respuesta sigue una distribución normal, el área comprendida en el intervalo $[\mu - 2\sigma, \mu + 2\sigma]$ abarca aproximadamente el 95 % de todos los tiempos de respuesta para el programa, o lo que es lo mismo, el tiempo de respuesta del programa se encuentra con una probabilidad del 95 % en ese intervalo[2], en este ejemplo $[2.62, 3.38]$. Esto se aprecia en la función de densidad de probabilidad del tiempo de respuesta en la figura 1.4.

1.4.2. Ley de Amdahl

Como se ha visto en la sección 1.1, el computador está formado por varios componentes que trabajan conjuntamente para ejecutar programas: CPU, memoria, sistema de interconexión y sistema de entrada/salida. Las mejoras en el rendimiento

[2]En realidad, dado que se tiene un número de medidas bajo (<30) y se desconoce la desviación típica poblacional, habría que utilizar una distribución t de *Student* para calcular el intervalo de confianza. Por simplicidad, se utiliza la distribución normal, con lo que el intervalo $[\mu - 2\sigma, \mu + 2\sigma]$ es una aproximación válida para una confianza del 95 %.

del computador se consiguen introduciendo mejoras en los distintos componentes que lo forman. No obstante, una mejora en el rendimiento de un componente en un factor p no incrementa el rendimiento de todo el computador en ese mismo factor.

Para el caso de un computador que ejecuta una única tarea en cada instante la ley de Amdahl determina que:

> *La aceleración o ganancia obtenida (A) en el rendimiento de un sistema completo debido a la mejora de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.*

El tiempo de respuesta del computador utilizando el componente mejorado se calcula a través de la suma de dos factores: el tiempo de respuesta debido a los componentes que no han sido mejorados, que permanece constante, y el tiempo empleado utilizando el componente mejorado, que se ve reducido en función de la mejora aplicada:

$$\text{T. respuesta}_{\text{mejorado}} = \text{T. respuesta}_{\text{original}} \times \left((1 - \text{Fracción}_{\text{mejorada}}) + \frac{\text{Fracción}_{\text{mejorada}}}{\text{Acelerac.}_{\text{mejorada}}} \right)$$

La fórmula de Amdahl puede deducirse a partir de la aceleración del rendimiento obtenida utilizando el componente mejorado respecto al rendimiento con el componente original:

$$A = \frac{\text{Rendimiento}_{\text{mejorado}}}{\text{Rendimiento}_{\text{original}}} = \frac{\text{T. respuesta}_{\text{original}}}{\text{T. respuesta}_{\text{mejorado}}}$$

Sustituyendo, tenemos que:

$$A = \frac{1}{(1 - \text{Fracción}_{\text{mejorada}}) + \frac{\text{Fracción}_{\text{mejorada}}}{\text{Acelerac.}_{\text{mejorada}}}}$$

Por ejemplo, supongamos que tenemos un computador que tarda en ejecutar un programa 100 segundos. De este tiempo, 20 segundos son debidos a la ejecución por parte de la CPU, 35 a los accesos a memoria, 40 a operaciones de entrada/salida sobre el disco duro, mientras que el resto se debe a la espera por los diferentes mecanismos de interconexión dentro del computador. ¿Cuál sería la aceleración en la ejecución del programa que se obtendría si se sustituye la CPU por otra el triple de rápida? ¿Y si se sustituye el disco duro por uno el doble de rápido? ¿Cuáles serían los nuevos tiempos de ejecución?

Inicialmente, se calcula la fracción de tiempo correspondiente a los componentes a mejorar:

$$\text{Fracción}_{\text{CPU}} = \frac{20}{100} = 0.2$$

$$\text{Fracción}_{\text{disco}} = \frac{40}{100} = 0.4$$

Para el caso de sustituir la CPU:

$$A = \frac{1}{(1-0.2)+0.2/3} \approx 1.15$$

$$\text{T. respuesta}_{\text{mejorado}} = \frac{\text{T. respuesta}_{\text{original}}}{A} \approx \frac{100}{1.15} = 86.96 \text{ segundos}$$

En el caso de que sustituya el disco:

$$A = \frac{1}{(1-0.4)+0.4/2} = 1.25$$

$$\text{T. respuesta}_{\text{mejorado}} = \frac{\text{T. respuesta}_{\text{original}}}{A} = \frac{100}{1.25} = 80 \text{ segundos}$$

Como se puede observar, la mejora de rendimiento sustituyendo el disco por uno el doble de rápido es mayor que la obtenida sustituyendo la CPU por una el triple de rápida. La razón es que el disco se emplea en el ejemplo durante una fracción de tiempo mucho mayor.

1.4.3. Rendimiento de la CPU

El rendimiento del computador depende de todos y cada uno de los elementos que lo componen. No obstante, el componente más complejo y por lo tanto el más susceptible de ser optimizado es la CPU. Por esta razón, cuando se estudia el rendimiento del computador se presta especial atención al rendimiento de la CPU.

Para comparar el rendimiento de dos CPU es necesario definir métricas que proporcionen una indicación de su rendimiento. Como se ha visto anteriormente, el tiempo de respuesta o la productividad son dos formas distintas de medir el rendimiento. Sin embargo, estas métricas hacen que la comparación del rendimiento de dos CPU sea en la práctica complicada.

El tiempo de respuesta de un programa indica el tiempo que transcurre desde que comienza el programa hasta que finaliza y suele ser variable, ya que incluye no solo el tiempo que el programa está realmente ejecutándose, al que se denomina **tiempo de CPU**[3], sino además otros factores, mostrados en la figura 1.5:

- Tiempo de CPU. Este tiempo depende fundamentalmente de la CPU, pero intervienen otros elementos como la memoria. Al fin y al cabo durante la ejecución de las instrucciones por parte de la CPU es necesario leer y escribir en la memoria. El tiempo de CPU se puede a su vez dividir en dos partes, el tiempo de CPU de usuario y el tiempo de CPU del sistema. Por ejemplo, durante la ejecución de un programa este llama a servicios del sistema operativo, los cuales se ejecutan en un modo de ejecución en la CPU denominado modo privilegiado y forman parte del tiempo de CPU de sistema.

[3]En algunos textos se utiliza *tiempo de ejecución* para referirse al tiempo de respuesta y en otros para referirse al tiempo de CPU. Para evitar confusiones, en este texto se utiliza tiempo de respuesta y tiempo de CPU para referirse al tiempo que tarda en finalizar el programa y al tiempo efectivo de ejecución por parte de la CPU, respectivamente.

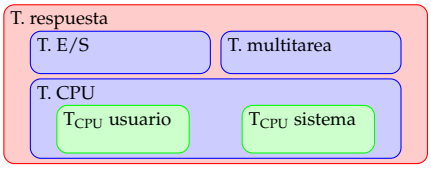


Figura 1.5: Factores que afectan al tiempo de respuesta de un programa

- El tiempo de espera por operaciones de entrada/salida. Por ejemplo, puede ser necesario esperar por una operación de lectura de disco antes de que el programa pueda continuar la ejecución.
- El tiempo multitarea. Durante breves periodos de tiempo el sistema operativo necesita ejecutarse para llevar a cabo operaciones que no están asociadas al programa en cuestión, como por ejemplo administrar los recursos de la máquina. Además, sobre un sistema operativo multitarea se ejecutan varias tareas o programas simultáneamente, no solo la tarea sobre la que se mide el tiempo de respuesta.

De los factores anteriores, el tiempo de CPU es el que proporciona una idea más cercana al rendimiento de la CPU. Por esta razón si se emplea el tiempo de respuesta de un programa para estimar el rendimiento de la CPU debe hacerse de forma muy cuidadosa para reducir al mínimo posible los otros factores que influyen en el tiempo de respuesta.

Otra alternativa para la medición del rendimiento de la CPU es medir la productividad de la CPU en instrucciones por segundo durante la ejecución del programa (habitualmente en millones de instrucciones por segundo o MIPS). El problema es que los MIPS no son comparables en dos CPU que no implementan el mismo juego de instrucciones.

Aunque la medición del rendimiento de la CPU es una labor complicada, resulta conveniente analizar desde el punto de vista teórico cuáles son los factores más determinantes. El siguiente apartado lleva a cabo este análisis.

Análisis teórico del tiempo de CPU

En un programa con un solo hilo de ejecución el tiempo de CPU depende de varios factores. El primer factor importante a considerar es el periodo de reloj (T), que es el inverso de la frecuencia (f). De esta forma puede hablarse indistintamente de periodo o frecuencia de reloj.

$$f = T^{-1} = \frac{1}{T}$$

Entonces, puede definirse el tiempo de CPU de un programa en relación a la frecuencia o el periodo de la señal de reloj como:

$$\mathrm{T_{CPU}} = \frac{\text{Ciclos de CPU del programa}}{f} = \text{Ciclos de CPU del programa} \times T$$

El número de Ciclos Por Instrucción (CPI) indican el número de ciclos que son necesarios para completar cada instrucción del programa y se pueden calcular como:

$$\text{CPI} = \frac{\text{Ciclos de CPU del programa}}{\text{Instrucciones del programa}}$$

A partir del CPI, se puede calcular el tiempo de CPU como:

$$\text{T}_{\text{CPU}} = \frac{\text{Instrucciones del programa} \times \text{CPI}}{f} = \text{Instrucciones del programa} \times \text{CPI} \times T$$

A esta fórmula se la conoce como la **ley de hierro** (*Iron Law*) del rendimiento de una CPU. Como se puede observar en la ecuación anterior, el tiempo de CPU final depende de tres factores relacionados:

- El número de instrucciones del programa. Este número depende de la arquitectura del juego de instrucciones empleada y del compilador. Por ejemplo, los programas en una arquitectura RISC suelen tener muchas más instrucciones que sus equivalentes en una arquitectura CISC. Por otra parte, el compilador es el responsable de traducir los programas escritos en un lenguaje de alto nivel a instrucciones de la CPU, por lo que juega un papel fundamental en el número de instrucciones del programa.

- El número de ciclos por instrucción (CPI). Depende de la organización interna de la CPU y de la complejidad de las instrucciones a ejecutar. Las CPU CISC implementan instrucciones que realizan operaciones complejas, lo que redunda en programas con un menor número de instrucciones. La parte negativa es que estas instrucciones requieren muchos ciclos de reloj para ser ejecutadas, es decir, tienen CPI alto. Por el contrario, las CPU RISC implementan instrucciones muy sencillas que requieren pocos ciclos para ejecutarse, es decir, tienen CPI bajo, pero que suponen programas con un mayor número de instrucciones.

  El principio básico enunciado por la ley de Amdahl de que la ganancia de rendimiento es proporcional a la fracción de uso del componente mejorado puede ser utilizado para reducir el CPI. En la fórmula anterior para el cálculo del tiempo de CPU el valor del CPI empleado es el CPI medio evaluado teniendo en cuenta todas las instrucciones máquina del programa. Esto se traduce en que los diseñadores de juegos de instrucciones buscan reducir el CPI de las instrucciones más comúnmente utilizadas en los programas, frente a instrucciones menos utilizadas, pues su peso en el rendimiento final será mayor.

- El periodo de reloj. Indica lo rápido que trabaja la CPU. Para reducir el ciclo de reloj, y aumentar la frecuencia de trabajo, es habitualmente necesario mejorar la tecnología de fabricación de la CPU para conseguir circuitos más rápidos. No obstante, también se puede disminuir el periodo de reloj incluyendo mejoras organizativas como el *pipeline*, tal como se verá en el tema dedicado a la CPU.

A partir de la ley de hierro se puede deducir la siguiente expresión para calcular los MIPS en función del CPI:

$$\text{MIPS} = \frac{1}{\text{CPI} \times T \times 10^6}$$

Y, por lo tanto, el tiempo de CPU de un programa se puede expresar como:

$$\text{T}_{\text{CPU}} = \frac{\text{Instrucciones del programa}}{10^6 \times \text{MIPS}}$$

1.4.4. Benchmarks

Un *benchmark* es un programa pensado para evaluar el rendimiento de un computador o de una de sus partes. Son habituales los benchmarks para evaluar el rendimiento de la CPU y del sistema de memoria.

Como se ha visto, el rendimiento de un computador no puede calcularse directamente la mayor parte de las veces debido a su complejidad, sino que debe estimarse a partir de mediciones. El resultado de estas mediciones varía en función de las condiciones a las que está sometido el computador. Un *benchmark* somete al computador, o a la parte del mismo a evaluar, a una determinada **carga de trabajo** para medir el rendimiento bajo la misma. Por esta razón, los resultados para distintos *benchmarks* pueden variar para el mismo computador.

Un *benchmark* debería someter al computador a una carga representativa del trabajo habitual para el que el computador está pensado, de tal forma que los resultados sean significativos. Por ejemplo, si se desea evaluar el rendimiento de un computador orientado a tareas de ofimática utilizando un *benchmark*, la carga que representa el *benchmark* debería ser parecida a la que supondrían los programas ofimáticos que se utilizarían en el día a día sobre el computador. Por otro lado, en la evaluación de una estación de trabajo dedicada al desarrollo de programas en lenguaje C debería utilizarse un *benchmark* con una carga representativa de compiladores, entornos integrados de desarrollo, enlazadores, etc. Por tanto, un *benchmark* está pensado para evaluar el rendimiento de un computador bajo unas determinadas circunstancias. Un computador puede obtener un bajo rendimiento medido con un *benchmark* con una determinada carga, mientras que puede obtener un rendimiento alto si se evalúa con otra carga distinta.

Hay dos tipos de *benchmarks* dependiendo del tipo de carga que utilizan:

- Carga real. Son *benchmarks* que utilizan programas reales usados en el trabajo habitual del computador sobre el que se va a aplicar el *benchmark*. La principal ventaja es que evalúan el rendimiento bajo unas condiciones reales de utilización del computador. Su gran inconveniente es que son difíciles de reproducir, con lo que las medidas obtenidas con el *benchmark* pueden tener gran variabilidad.

	Productividad C1	Productividad C2
Benchmark B1	5	10
Benchmark B2	5	2

Tabla 1.1: Ejemplo de medición de rendimiento con dos *benchmarks*

	$A_{C1/C2}$	$A_{C2/C1}$
Benchmark B1	0.5	2
Benchmark B2	2.5	0.4
Media aritmética B1 y B2	1.5	1.2
Media geométrica B1 y B2	1.12	0.89

Tabla 1.2: Aceleraciones utilizando *benchmarks* individuales o estadísticos combinando ambos

- Carga sintética. Estos *benchmarks* ejecutan pequeños programas que intentan reproducir las operaciones más habituales que se desarrollan sobre los programas reales. Sin embargo, no son programas reales. Su ventaja es que son fácilmente reproducibles. Por contra, los resultados obtenidos no siempre se corresponden con el rendimiento observado en la ejecución de programas reales.

Existen *benchmarks* para estimar el rendimiento de muchas partes del computador. Los hay orientados a medir el rendimiento de la CPU, del sistema de memoria, de la interfaz gráfica, del disco duro, etc.

A partir del resultado proporcionado por un *benchmark* se puede comparar el rendimiento de dos computadores y calcular la aceleración de uno respecto a otro para la carga de trabajo asociada al *benchmark*. No obstante, en ocasiones es necesario ejecutar diferentes *benchmarks* para evaluar el rendimiento del computador frente a diferentes cargas de trabajo, obteniéndose diferentes aceleraciones. Puede ocurrir que un computador tenga un mayor rendimiento que otro para una carga de trabajo, pero menor para otra carga de trabajo diferente. La tabla 1.1 muestra el resultado de la ejecución de dos *benchmarks* B1 y B2, sobre 2 computadores C1 y C2.

El rendimiento del computador C1 para el *benchmark* B1 es menor que el del computador C2, tal como se deduce de sus productividades. Sin embargo, para el *benchmark* B2 ocurre justo lo contrario, el computador C1 tiene un rendimiento mayor que C2. Estos resultados se reflejan en las aceleraciones $A_{C1/C2}$, de C1 respecto de C2, y la inversa ($A_{C2/C1}$), dependiendo del computador que se tome como referencia, mostradas en la tabla 1.2.

Cuando es necesario resumir el rendimiento de un computador respecto a otro empleando un único número se puede emplear la media de las aceleraciones. Una opción sería emplear la media aritmética, en cuyo caso la aceleración agregada del computador C1 respecto de C2 sería $A_{C1/C2} = (0.5 + 2.5)/2 = 1.5$, de donde se deduciría que el computador C1 es 1.5 veces más rápido que C2. Por su parte, si se llevase a cabo la media aritmética, pero en este caso del computador C2 respecto de C1, se obtendría $A_{C2/C1} = (2+0.4)/2 = 1.2$, de donde se deduciría que el computador

C2 es 1.2 veces más rápido que C1. Como se puede observar, se obtienen resultados contradictorios dependiendo del computador que se tome como referencia. Por esta razón no se emplea la media aritmética de aceleraciones.

Para obtener una aceleración agregada a partir de las aceleraciones obtenidas para n *benchmarks* suele emplearse la media geométrica, de acuerdo a la expresión general:

$$\mathrm{A} = \sqrt[n]{\mathrm{A}_1 \cdot \mathrm{A}_2 \cdot ... \cdot \mathrm{A}_n}$$

En el ejemplo de la tabla 1.2 se puede observar cómo los resultados de aceleración son coherentes cuando se usa la media geométrica. La aceleración del computador C1 respecto de C2 es 1.12, por lo que el computador C1 tiene un rendimiento 1.12 veces superior a C2. Tomando como referencia el computador C2, el computador C2 es 0.89 veces más rápido que C1, o lo que es lo mismo, el computador C1 es $(1/0.89) = 1.12$ veces más rápido que C2. Por lo tanto, las aceleraciones obtenidas son coherentes sea cual sea el equipo tomado como referencia.

Capítulo 2

La CPU

Este capítulo se centra en el estudio de las características fundamentales en las que se basa el diseño de las CPU actuales. Para ello, se utiliza la arquitectura MIPS64 a modo de ejemplo. El capítulo comienza con una breve descripción de la arquitectura del juego de instrucciones MIPS64 para, posteriormente, plantear posibles implementaciones. A partir de aquí, se abordan los principios de diseño de las CPU actuales:

- Mejoras del rendimiento. Estas mejoras giran en torno al concepto de *pipeline* (segmentación de instrucciones) y se detallan en los apartados 2.3 a 2.6.
- Soporte a los sistemas operativos multitarea. Se describe en el apartado 2.7.
- Soporte a la virtualización. Se trata de forma introductoria en el apartado 2.8.

Adicionalmente, a lo largo de este capítulo se harán breves reseñas a la gestión de energía de la CPU, pues también supone una de las características críticas en los computadores actuales.

2.1. Arquitectura MIPS64

Para ilustrar los conceptos abordados en este tema se utilizará una arquitectura real: MIPS64. El principal motivo es su sencillez. Se trata de una arquitectura RISC siguiendo el modelo de máquina de carga/almacenamiento. Esto implica que la CPU únicamente opera sobre registros internos, por lo que cualquier dato debe ser traído desde memoria a uno de estos registros antes de poder realizar operaciones sobre él. Además, solo acceden a memoria las instrucciones de carga y almacenamiento, lo que repercute en una implementación más sencilla como se verá más adelante.

MIPS64 constituye una extensión de la arquitectura MIPS32 con la que es totalmente compatible, por lo que puede ejecutar indistintamente programas desarrollados para MIPS32 y MIPS64. Procesadores implementando la arquitectura MIPS64 se pueden encontrar en consolas de videojuegos portátiles y dispositivos de red.

Esta arquitectura presenta grandes similitudes con la arquitectura ARMv8, otra arquitectura RISC, muy común en teléfonos móviles inteligentes y gran cantidad de dispositivos empotrados.

Hoy en día, RISC se ha convertido en el paradigma de diseño de la mayor parte de las CPU. Todas ellas, incluidas las CPU MIPS64, se basan en los mismos principios de diseño:

- Instrucciones muy sencillas y de tamaño fijo, de tal forma que puedan decodificarse muy rápidamente.
- Pocos modos de direccionamiento y simples, lo que contribuye a un diseño hardware más sencillo.
- Gran número de registros de propósito general. De esta forma, muchas de las variables de los programas se almacenan en registros, reduciendo el número de accesos a memoria necesarios y mejorando el rendimiento.

MIPS64 es una arquitectura de 64 bits con un juego de instrucciones muy simple, que fue introducida en 1991 y se considera la primera arquitectura de 64 bits. Dispone de un conjunto de 32 registros para enteros de 64 bits denominados `R0`, ..., `R31`; si bien `R0` siempre vale 0 y `R31` está reservado para almacenar la dirección de retorno en ciertas instrucciones de salto. Existen también 32 registros de punto flotante de 64 bits denominados `FP0`, ..., `FP31` y un registro de estado para operaciones sobre números enteros y otro para operaciones sobre punto flotante.

La memoria es direccionable a nivel de bytes utilizando direcciones de 64 bits. Esto significa que una CPU MIPS64 es capaz de direccionar 2^{64} bytes, o, lo que es lo mismo, 16 EiB (16 exbibytes). Al tratarse de una máquina de carga/almacenamiento, las operaciones de lectura (carga) y de escritura (almacenamiento) de memoria se realizan siempre sobre los registros de propósito general (`Rx` o `FPx`).

MIPS64 sigue un modelo Harvard en su configuración de memoria. Esto quiere decir que las CPU MIPS64 disponen de memorias independientes para instrucciones y para datos, de forma que es posible acceder simultáneamente a ambas. Este modelo se ilustra en la figura 2.1, donde se pueden apreciar las dos memorias (de instrucciones y datos). Esto implica que estas CPU tienen replicadas las líneas de acceso a memoria.

2.1.1. Tipos de datos

La arquitectura MIPS64 es capaz de trabajar de forma nativa con datos de tipo entero y de tipo real. Es importante resaltar el aspecto *de forma nativa*, pues si bien la CPU solo entiende estos tipos de datos, los diferentes niveles de abstracción que se despliegan en un computador (ensamblador, sistema operativo, lenguaje de programación, etc.) pueden definir otros tipos de datos que se implementan a partir de los anteriores. Por ejemplo, un tipo de datos *cadena de caracteres* podría definirse a partir de un tipo de datos entero como una secuencia de enteros.

MIPS64 puede trabajar con enteros en complemento a 2 con los siguientes tamaños, ilustrados en la figura 2.2:

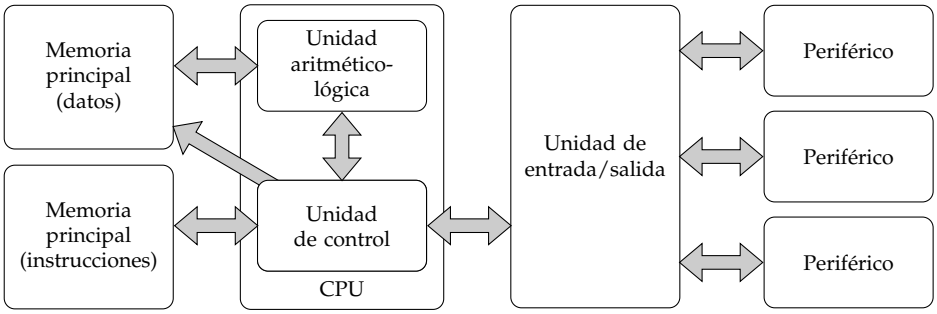


Figura 2.1: Arquitectura Harvard

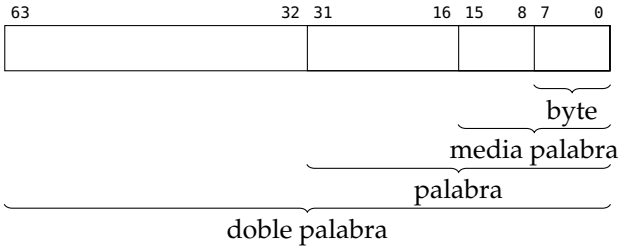


Figura 2.2: Tipos de datos soportados por la arquitectura MIPS64

- **byte**: es un entero de 8 bits.
- **media palabra** (*half-word*): es un entero de 16 bits.
- **palabra** (*word*): es un entero de 32 bits.
- **doble palabra** (*doble-word*): es un entero de 64 bits.

Cuando se realizan operaciones de carga o almacenamiento de registros para enteros (Rx), esto es, lecturas y escrituras de memoria de valores enteros, se puede acceder a 1 byte, una media palabra (2 bytes), una palabra (4 bytes) o una doble palabra (8 bytes) en memoria. Cuando se produce la carga de un valor de menos de 64 bits (byte, media palabra o palabra) es posible realizarla con signo o sin signo. Si se realiza sin signo inicialmente todo el registro se pone a 0 y después se copia a la parte baja del mismo el dato leído de memoria. En cambio, si se realiza con signo se copia el dato de memoria en la parte baja del registro y se extiende el bit de signo, es decir, se replica el bit más significativo del dato en los bits más significativos del registro no modificados por el dato leído. Esto se ilustra con un ejemplo en la figura 2.3, donde se carga la media palabra 90BDh en un registro. Cuando se hace sin signo, el registro de 64 bits se cargará con el valor 0000 0000 0000 90BDh. En cambio, si la carga se realiza con signo, el valor del registro será FFFF FFFF FFFF 90BDh. En todo caso, los accesos a más de un byte de memoria deben estar alineados, esto es, la dirección de memoria debe ser múltiplo de 2 para el acceso a medias palabras, de 4 para el acceso a palabras, y de 8 para el acceso a dobles palabras. La figura 2.4 muestra dos accesos a palabras, uno alineado y otro no. Como se puede apreciar, el

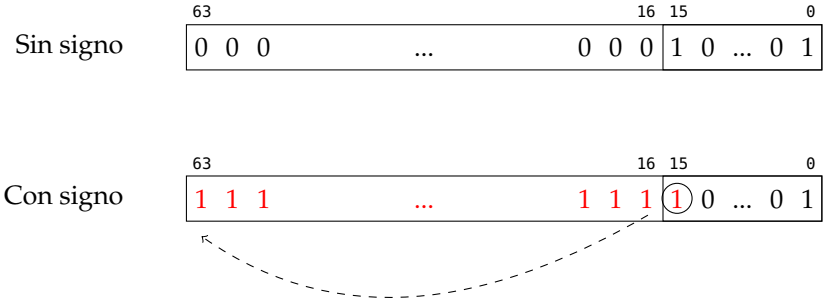


Figura 2.3: Carga de la media palabra 90BDh en un registro sin signo (*zero-extended*) y con signo (*sign-extended*)

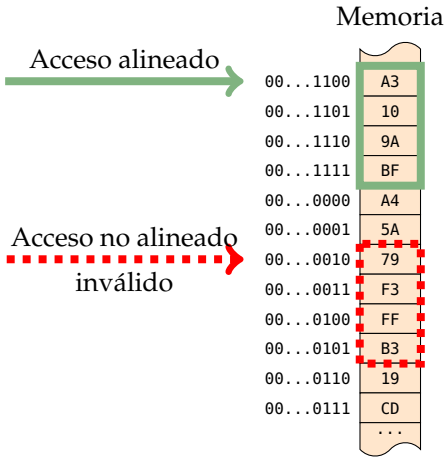


Figura 2.4: Accesos alineados y no alineados a palabras en memoria

segundo de los accesos se realiza a una dirección de memoria que no es múltiplo de 4 (el tamaño en bytes de una palabra).

Otra cuestión importante a tener en cuenta cuando se almacena en memoria un dato compuesto de más de un byte es cómo se ordenan estos bytes. Esta ordenación es configurable por hardware en una CPU MIPS64, pudiendo ser de tipo *big-endian* o *little-endian* (se dice que es una arquitectura *bi-endian*). Para el caso del acceso alineado de la figura 2.4 la palabra se interpretaría como el entero BF9A 10A3h en una configuración *little-endian*, y el entero A310 9ABFh en una configuración *big-endian*.

Para el caso de números reales, la arquitectura MIPS64 soporta dos tamaños: precisión simple (palabra de 32 bits) y precisión doble (doble palabra de 64 bits), ambos utilizando una codificación IEEE-754. Al igual que ocurre con los datos enteros, la carga de los registros de punto flotante debe realizarse utilizando direcciones alineadas.

2.1.2. Juego de instrucciones

Como cualquier arquitectura, MIPS64 define un juego de instrucciones que incluye instrucciones aritméticas, lógicas, de transferencia de datos y saltos, entre otras. Al tratarse de una arquitectura RISC, el juego de instrucciones de MIPS64 es muy simple comparado con otros juegos de instrucciones, como por ejemplo el que soportan las CPU x86, montadas en computadores personales. Esta simplicidad permite el uso de códigos de instrucción de tamaño constante, lo que en última instancia redunda en un hardware más sencillo y facilita la carga de los códigos de instrucción y su decodificación. Además, como se ha visto en el capítulo 1, las instrucciones sencillas consiguen fácilmente un CPI bajo, es decir, se pueden ejecutar en muy pocos ciclos de reloj.

La arquitectura MIPS64 define, entre otros, los siguientes tipos de instrucciones:

- Instrucciones de carga y almacenamiento. Se usan para leer y escribir en la memoria. Siempre implican la utilización de un registro de propósito general entero o de punto flotante. Solamente este tipo de instrucciones accede a memoria.
- Instrucciones aritmético-lógicas. Son instrucciones para la realización de operaciones aritméticas, tanto sobre enteros como sobre números reales, y operaciones lógicas.
- Instrucciones de control de flujo. Estas instrucciones permiten modificar el flujo secuencial de los programas. Existen saltos de tipo incondicional (*jumps*), que siempre se toman, y saltos condicionales (*branches*), que se toman en base a una determinada condición.
- Otras instrucciones. Se incluyen desplazamientos de bits, conversión entre enteros y reales, entre otras.

Aunque se emplea la arquitectura MIPS64 para ilustrar muchos de los conceptos teóricos de la CPU durante el capítulo, no se considera el juego de instrucciones completo sino una versión reducida del mismo que se puede encontrar en el apéndice A.

Modos de direccionamiento

Las instrucciones que ejecuta una CPU requieren el uso de operandos y reservan una cantidad determinada de bits en el código de la instrucción para hacerles referencia. Los modos de direccionamiento de una CPU determinan cómo se obtienen estos operandos a partir de estos bits. Existen tres posibles ubicaciones de los operandos con sus ventajas e inconvenientes.

Por un lado, los operandos pueden encontrarse en memoria. La principal ventaja es la cantidad de posiciones de memoria disponibles. Como contrapartida, están el tiempo de acceso al operando más elevado y el mayor número de bits a utilizar en el código de una instrucción para referenciar al operando, ya que en última instancia se necesita una dirección de memoria que lo identifique.

Otra opción es el uso de operandos ubicados en registros de la CPU. Esta alternativa es mucho más rápida que la anterior, pues el tiempo de acceso a un registro de la CPU es considerablemente inferior al acceso a memoria. Además, se necesitarían unos pocos bits del código de instrucción para identificar el registro (para 32 registros enteros se necesitan 5 bits). El principal inconveniente es el escaso número de registros disponibles en comparación con la capacidad de la memoria.

Por último, en lugar de utilizar unos bits en el código de la instrucción para referenciar el operando (en memoria o en registro), este puede estar contenido directamente en el propio código de la instrucción. En ese caso el acceso al operando es inmediato, si bien solo es válido para constantes, pues se codifican junto con el resto de la instrucción. Otro problema es el elevado número de bits necesarios en el código de instrucción y que solo sirve para operandos fuente, es decir, no se puede almacenar el operando resultado en el código de la instrucción como valor inmediato.

MIPS64 admite 5 modos de direccionamiento:

- Inmediato. En este caso el valor del operando se incluye en el propio código de la instrucción. Por ejemplo, la siguiente instrucción realiza la suma del entero de 64 bits almacenado en el registro **r8** con el número -3 y almacena el resultado en el registro **r4**. En este caso, la constante -3 vendrá codificada en complemento a 2 en el propio código de la instrucción.

```
daddi r4, r8, -3
```

- Registro. Con este modo se incluye el número de registro dentro del código de la instrucción. Así, para el caso anterior, además de incluir la constante -3, el código de la instrucción incluirá 5 bits que identifican el registro fuente (**r8**) y otros 5 bits que identifican el registro destino (**r4**). Usando este modo de direccionamiento el valor del operando se encuentra en el registro referenciado.

- Base más desplazamiento (indexado). En este modo el operando se encuentra en una posición de memoria indicada por la suma del valor de un registro con una constante codificada en el código de la instrucción. A continuación se muestran varios ejemplos de carga de una doble palabra:

```
ld r12, 200(r0)  ; direccionamiento directo a la dirección 200 (r0 siempre vale 0)
ld r12, 0(r3)    ; direccionamiento indirecto a la dirección contenida en r3
ld r12, 350(r3)  ; direccionamiento indexado a la dirección contenida en r3 + 350
```

  Cuando se utiliza **r0** como registro base se habla de *direccionamiento directo*, pues como este registro siempre vale 0, la dirección final donde se ubica el operando en memoria viene dada directamente por la constante. Si en cambio se utiliza 0 como constante, se habla de *direccionamiento indirecto*, pues es el registro base el que determina la posición en memoria del operando. En cualquier caso, debe tenerse en cuenta que la suma del registro base y el desplazamiento debe dar como resultado un acceso alineado a memoria. En el ejemplo, la suma de 350 y **r3** debería ser múltiplo de 8.

- Relativo al PC. Se utiliza en saltos condicionales para calcular la dirección de salto como la suma del contador de programa (PC) y un valor calculado a partir de una constante codificada en el propio código de instrucción. El siguiente ejemplo muestra una instrucción de salto condicional que salta cuando los valores de ambos registros son iguales.

```
beq r2, r9, -5
```

  La constante indica el número de instrucciones a saltar, ya que es desplazada 2 bits a la izquierda (es equivalente a multiplicar por 4, que es el tamaño en bytes del código de instrucción de todas las instrucciones) antes de sumarla al registro PC. En este caso $-5 \times 4 = -20$. Así, el número de posiciones a sumar al PC es siempre múltiplo de 4, lo que garantiza un acceso alineado a la memoria de código.

  Con este modo de direccionamiento los saltos condicionales pueden saltar una distancia de $\pm$128 KiB.

- Pseudodirecto. Se utiliza en saltos incondicionales para calcular la dirección de salto como la concatenación de 26 bits del código de la instrucción desplazados 2 posiciones a la izquierda (28 bits) y los 36 bits $(64 - 28)$ más significativos del contador de programa. A continuación se muestra un ejemplo de salto incondicional:

```
j 1000
```

  En este ejemplo, suponiendo que el registro PC contiene D000 0000 0000 0000h al ejecutar la instrucción de salto, la dirección destino se calcula tomando los 36 bits más significativos de este valor (D 0000 0000h) y concatenándolos con el resultado de desplazar la constante 2 bits hacia la izquierda (1000$\times$4 = 4000 = FA0h), lo que resulta en un valor final para el PC de D000 0000 0000 0FA0h (que será siempre múltiplo de 4).

  Con este modo de direccionamiento la memoria de código se organiza en zonas de 256 MiB. Modificando el valor de la constante adecuadamente, se podría saltar a cualquier instrucción almacenada dentro de la misma zona actualmente apuntada por el PC. En el ejemplo anterior se accede a la dirección FA0h dentro del rango de 256 MiB apuntado por el PC.

Codificación de las instrucciones

Como se mencionó anteriormente, todas las instrucciones en MIPS64 se codifican con el mismo número de bits. Concretamente, todos los códigos de instrucción tienen un tamaño de 32 bits, que se codifican siguiendo alguno de los tres tipos mostrados en la figura 2.5. Las instrucciones aritméticas y lógicas son todas de tipo R. Algunas instrucciones de tipo I son las instrucciones de carga y almacenamiento, y las de salto condicional. Por último, las instrucciones de salto incondicional son de tipo J.

Los 32 bits que forman el código de instrucción se dividen en campos:

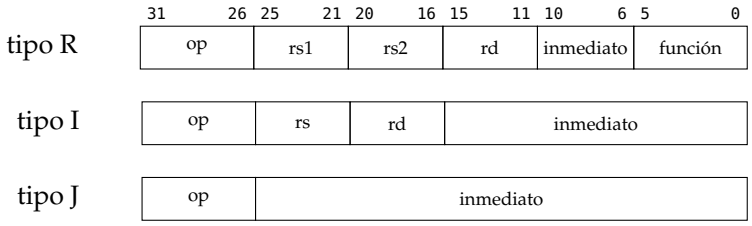


Figura 2.5: Codificación de instrucciones en MIPS64

- `op`. Existe para todos los tipos de instrucción y representa el código de operación (*operation code* o *opcode*), que identifica la operación realizada por la instrucción.
- `función`. Concreta la variante de la operación que representa el campo `op`. Por ejemplo, el código de operación es el mismo para las instrucciones **`dadd`** y **`dsub`** (`0`) y se diferencian únicamente en el código de función: `2Ch` y `2Eh`, respectivamente.
- `rs/rs1`. Es el número de registro que contiene el primer operando fuente.
- `rs2`. Es el número de registro que contiene el segundo operando fuente.
- `rd`. Es el número de registro donde se almacenará el operando destino.
- `inmediato`. Representa valores inmediatos codificados junto con el resto del código de instrucción. Este valor inmediato se utiliza, por ejemplo, para los desplazamientos de bits en el caso de las instrucciones de tipo R, para los saltos condicionales en el caso de las instrucciones de tipo I y para los saltos incondicionales en el caso de las instrucciones de tipo J.

2.2. Microarquitectura monociclo

Una vez se ha definido la arquitectura del juego de instrucciones de una CPU, en este caso la arquitectura MIPS64, llega el momento de implementarla. Tal como se comentó en el capítulo 1, son posibles varias implementaciones. La implementación más sencilla es mediante una microarquitectura monociclo, donde cada instrucción necesita un único ciclo de reloj para ejecutarse.

Para ilustrar las ideas que subyacen tras esta microarquitectura se implementará un subconjunto de la arquitectura del juego de instrucciones MIPS64. Concretamente, se implementarán las instrucciones de carga y almacenamiento (sobre dobles palabras), las instrucciones aritméticas básicas sobre enteros (excluidas las instrucciones de desplazamientos de bits, multiplicación y división), una instrucción de salto condicional (**`beq`**), y una instrucción de salto incondicional (**`j`** `imm_26`) a modo de ejemplo. Quedan excluidas las instrucciones sobre números reales. La tabla 2.1 enumera las instrucciones a implementar.

Toda microarquitectura consta de dos elementos básicos: el camino de datos y la unidad de control. El primero constituye la parte ejecutiva de la CPU. Su misión

Instrucción	Tipo	Descripción
ld Rd,imm_16(Ri)	I	Carga de doble palabra
sd Rs,imm_16(Ri)	I	Almacenamiento de doble palabra
beq Rs1,Rs2,imm_16	I	Salto condicional si los registros son iguales
daddi Rd,Rs,imm_16	I	Suma de registro y valor inmediato
j imm_26	J	Salto incondicional
dadd Rd,Rs1,Rs2	R	Suma de registros
dsub Rd,Rs1,Rs2	R	Resta de registros
and Rd,Rs1,Rs2	R	Operación lógica *and*
or Rd,Rs1,Rs2	R	Operación lógica *or*
slt Rd,Rs1,Rs2	R	*set on less than* (`if Rs1<Rs2 Rd=1; else Rd=0`)

Tabla 2.1: Instrucciones a implementar por la microarquitectura

es llevar a cabo las operaciones que dictan las instrucciones: sumar enteros, aplicar operaciones lógicas, realizar saltos, etc. Por su parte, la unidad de control es la encargada de generar las señales de control necesarias, en la secuencia correcta, para que cada instrucción pueda ejecutarse sobre el camino de datos.

La unidad de control recibe cierta información del camino de datos: código de operación de la instrucción a ejecutar y algunos *flags* de estado durante la ejecución; y genera diferentes señales de control que llegan a las distintas unidades funcionales que constituyen el camino de datos.

En los siguientes apartados se aborda la construcción del camino de datos de forma incremental; la unidad de control para este camino de datos se detalla en el apéndice B.

2.2.1. Unidades funcionales

Antes de desarrollar el camino de datos es conveniente identificar las unidades funcionales necesarias para implementarlo, para lo cual es necesario tener en cuenta las instrucciones que ejecuta.

Gracias a la simplicidad del juego de instrucciones de MIPS64, la mayor parte de las instrucciones se comportan de una forma regular, lo que facilita la identificación de las unidades funcionales. Así, presentan pasos comunes:

- Búsqueda del código de instrucción. Inicialmente, el código de la instrucción apuntado por el contador de programa (`PC`) debe ser leído de la memoria de instrucciones. De esto se deduce que debe existir un registro al efecto en el camino de datos. Como las direcciones con las que trabaja una CPU MIPS64 son de 64 bits, el registro `PC` tiene un tamaño de 64 bits y se implementa con 64 biestables D (uno por bit). Este registro se lee para buscar en memoria la siguiente instrucción a ejecutar y se escribe para apuntar a la siguiente instrucción, bien sea la siguiente en orden secuencial, o bien una distinta si se cambia el flujo de ejecución de instrucciones como resultado de un salto. Puede implementarse junto con los registros de propósito general dentro del fichero de registros.

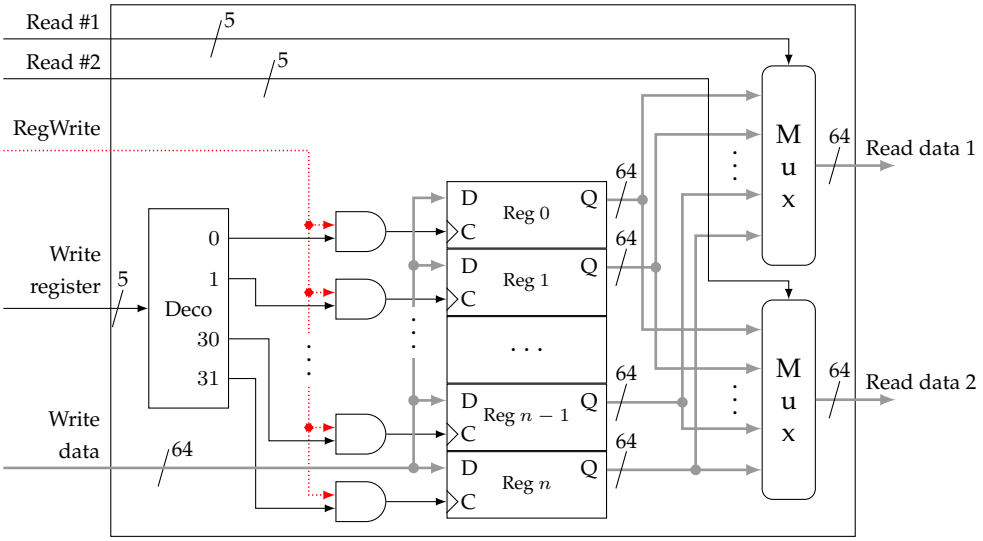


Figura 2.6: Posible implementación del fichero de registros

- Acceso a registros. La inmensa mayoría de instrucciones requiere leer uno o varios registros. Estos habitualmente se implementan como posiciones en un fichero de registros multipuerto. Se trata de una memoria ultra-rápida en la que puede accederse a varias posiciones al mismo tiempo. El fichero de registros constituye otra unidad funcional, de la que se muestra una posible implementación en la figura 2.6.

  Como se puede apreciar en la figura 2.6, un fichero de registros contiene un conjunto de registros. Para el caso de la arquitectura MIPS64 reducida, existen un total de 32 registros enteros de 64 bits, donde cada registro se implementa mediante 64 biestables D. Las entradas y salidas mostradas en la figura para cada registro llegarían a cada uno de sus 64 biestables. El contenido de los registros siempre está disponible en las líneas `Q`. Como en MIPS64 algunas instrucciones requieren la lectura de dos registros al mismo tiempo, por ejemplo **`dadd r2, r1, r6`** necesita leer **`r1`** y **`r6`**, se disponen dos multiplexores a la salida de los registros para seleccionar, a través de las líneas `Read #1` y `Read #2`, qué dos registros leer. Los valores de ambos registros serán volcados a las líneas `Read data 1` y `Read data 2` respectivamente. Cada uno de estos multiplexores recibe 5 líneas para seleccionar uno de entre los 32 registros. Para el caso de escritura, los registros pueden modificarse activando su línea `C` y poniendo en las líneas `Write data` los 64 bits a escribir. La línea `C` del registro que se quiere escribir se activa habilitando la línea `RegWrite` al mismo tiempo que se indica el número de registro en las líneas `Write register`.

- Uso de la ALU. Otro de los componentes básicos de un camino de datos es la unidad aritmético-lógica, donde se realizan las operaciones de aritmética de enteros y lógicas. La ALU en una CPU MIPS64 trabaja con operandos de 64 bits. Recibe dos operandos de entrada y genera un resultado de 64 bits.

Bits de control	Operación
0000	*and*
0001	*or*
0010	suma
0110	resta
0111	*set on less than*

Tabla 2.2: Operaciones de la ALU en función de las señales de control

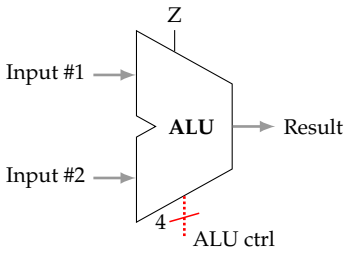


Figura 2.7: Unidad aritmético-lógica

Además, activará un *flag* Z (señal Zero) si el resultado de la última operación ha sido 0. La operación a realizar por parte de la ALU (suma, resta, *and*, *or* o *set on less than* –establecer si es menor–) se indica a través de 4 líneas de control que recibe como entrada[1]. Los posibles valores de estas señales y su significado se muestran en la tabla 2.2. La figura 2.7 presenta una representación esquemática de esta unidad funcional.

Por otra parte, algunas instrucciones harán uso de otros elementos auxiliares como extensores de signo, que replican el bit de signo de un entero para generar un valor de 64 bits, mientras que otras utilizarán desplazadores de bits, como por ejemplo las operaciones de salto.

2.2.2. Camino de datos monociclo

En este apartado se va a construir de forma incremental el camino de datos que soporte el subconjunto del juego de instrucciones MIPS64 listado en la tabla 2.1. Cada instrucción se ejecutará en un único ciclo de reloj.

Como se comentó anteriormente, la primera operación que deben realizar todas las instrucciones es la búsqueda del código de la instrucción en memoria. En la figura 2.8 se muestra la parte del camino de datos que se encarga de la búsqueda del código de la instrucción. Aparece el registro contador de programa (PC), cuyo valor se utiliza para acceder a la memoria de instrucciones y obtener el código de la instrucción. Este valor se corresponde con la dirección de memoria donde se encuentra el código de instrucción. Una vez leído este código, pasa a estar disponible en las

[1] En este texto no se abordará el diseño de una ALU.

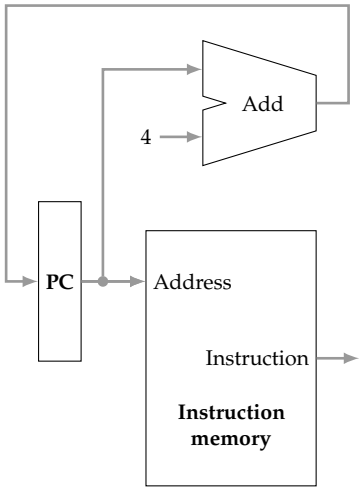


Figura 2.8: Búsqueda de la instrucción a ejecutar e incremento del PC

líneas Instruction. Finalmente, es necesario incrementar el contador de programa en 4 posiciones, ya que todos los códigos de instrucción tienen un tamaño de 32 bits (4 posiciones de memoria). Para ello se utiliza un sumador específico con uno de los operandos programado en hardware con el valor 4. Este proceso de búsqueda del código de instrucción e incremento del contador de programa se repite en cada ciclo de la señal de reloj.

A continuación, muchas instrucciones requieren leer uno o dos registros. Por ejemplo, las instrucciones de tipo R (ver figura 2.5) requieren leer dos registros, mientras que las de tipo I requieren leer un único registro. Para ello es necesario el acceso al fichero de registros.

Considerando únicamente el subconjunto del juego de instrucciones a soportar, el tipo R se corresponde con instrucciones que realizan operaciones aritméticas, como por ejemplo `dadd r1, r2, r3`, y lógicas, como `or r5, r9, r6`. Estas instrucciones leen dos registros, realizan un cálculo con ellos utilizando la ALU y por último almacenan el resultado en otro registro. La figura 2.9 muestra una posible implementación de esta parte del camino de datos. En las instrucciones de tipo R los números de los registros a leer se codifican en el código de instrucción utilizando 5 bits. El primer operando viene determinado por los bits de peso 25 a 21 y el segundo por los bits de peso 20 a 16. Los valores leídos de los registros deben enviarse a la ALU para proceder con la operación determinada por la instrucción (suma, resta, *and*, *or*, etc.). El resultado se almacenará en el registro indicado por los bits 15 a 11. Se muestran también las señales de control que determinan el comportamiento del fichero de registros y de la ALU. Estas son activadas por la unidad de control para ejecutar las instrucciones sobre el camino de datos. En este caso, se activa la señal RegWrite para almacenar el resultado de la ALU en el registro correspondiente. Una posible implementación del control se aborda en el apéndice B.

Una primera versión del camino de datos soportando las instrucciones de tipo R se obtiene combinando la lógica para la búsqueda del código de la instrucción e

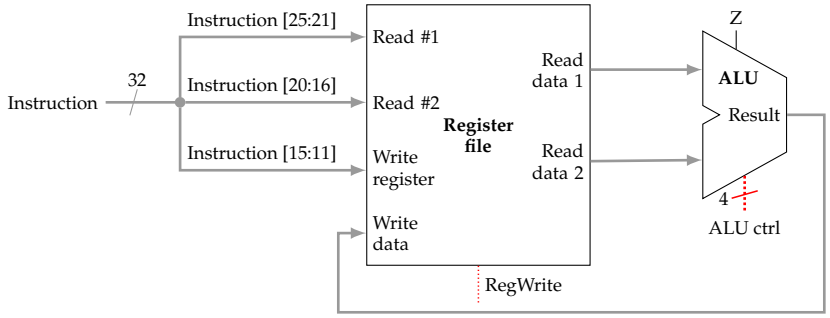


Figura 2.9: Ejecución de instrucciones de tipo R

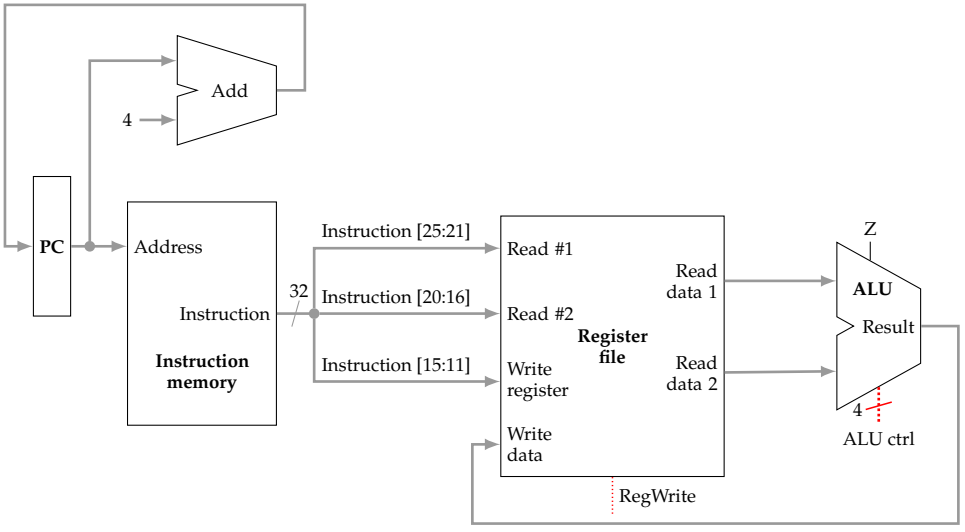


Figura 2.10: Camino de datos para la ejecución de instrucciones de tipo R (instrucciones lógicas y aritméticas sobre enteros)

incremento del PC (figura 2.8) y la lógica para la ejecución de las instrucciones de tipo R (figura 2.9). Este camino de datos se muestra en la figura 2.10.

El siguiente tipo de instrucciones a considerar son las de acceso a memoria, tanto carga como almacenamiento. Estas son del tipo I, si bien no comprenden todas las de este tipo, y son de la forma `ld r3, 200(r4)` o `sd r6, 12(r5)`. Siempre implican un registro (destino en el caso de una carga y fuente en el caso de un almacenamiento), un registro base y un valor inmediato de 16 bits. En estas instrucciones la dirección final se calcula como la suma del registro base y el valor inmediato con el signo extendido a 64 bits. En la figura 2.11 se muestra la lógica necesaria para soportar las instrucciones de acceso a memoria.

En el caso de una instrucción de carga, el registro base se obtiene de los bits 25 a 21 del código de la instrucción. Este registro debe sumarse con el valor inmediato representado con los bits 15 a 0 extendido a 64 bits utilizando la ALU. De ahí la necesidad de incluir una unidad de extensión de signo. Tras la suma, el resultado

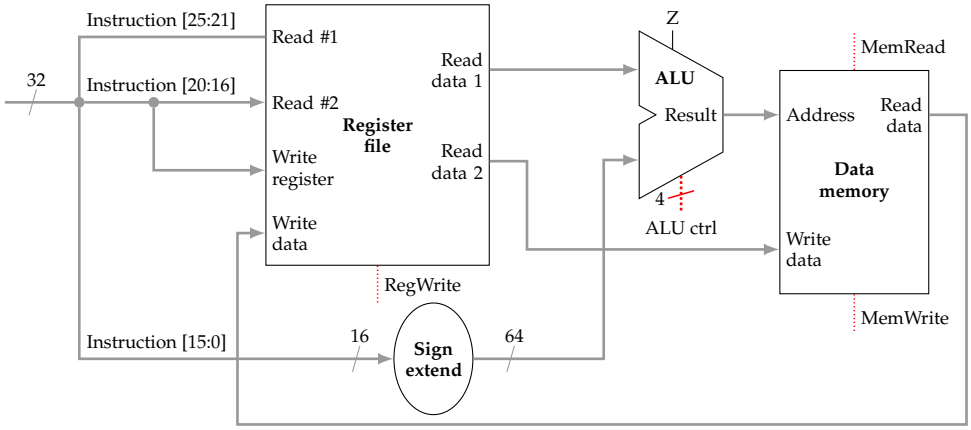


Figura 2.11: Ejecución de instrucciones de carga y almacenamiento

constituye la dirección de memoria a leer. Debe activarse además la señal `MemRead` para señalizar una lectura de memoria. Una vez leído, el dato se vuelca a las líneas `Write data` del fichero de registros. El registro a escribir vendrá determinado por los bits 20 a 16 del código de instrucción. Por último, se activa la señal `RegWrite` para almacenar el valor en el registro correspondiente.

El paso inicial en una instrucción de almacenamiento es idéntico al de una instrucción de carga. La dirección de memoria se obtiene de la misma forma, es decir, sumando registro base y valor inmediato. A partir de aquí comienzan las diferencias. En las operaciones de almacenamiento es necesario leer un segundo registro, indicado a través de los bits 20 a 16 del código de instrucción. El contenido de este registro se vuelca en las líneas `Write data` de memoria. Tras activar la señal `MemWrite` los datos quedan almacenados en memoria.

La lógica para ejecutar las instrucciones de carga y almacenamiento presentada en la figura 2.11 no es directamente combinable con el camino de datos básico mostrado en la figura 2.10. Por ejemplo, en las instrucciones aritmético-lógicas el registro destino viene dado por los bits 15 a 11 del código de instrucción y el valor a volcar en las líneas `Write data` del fichero de registros es el resultado generado por la ALU, mientras que en las operaciones de carga el registro destino viene dado por los bits 20 a 16 y el valor de las líneas `Write data` es el leído de memoria. Por esta razón, deben añadirse multiplexores con sus respectivas líneas de control para identificar estas situaciones y seleccionar qué valores tomar en cada caso. En la figura 2.12 se muestra el camino de datos resultante con tres multiplexores adicionales.

El multiplexor sobre las líneas `Write register` del fichero de registros sirve para seleccionar a través de la línea `RegDst` el campo del código de instrucción que indica el registro destino (bits 20 a 16 en operaciones de carga y bits 15 a 11 en operaciones aritmético-lógicas). Esta línea de control estará activada durante las instrucciones aritmético-lógicas y desactivada durante las instrucciones de lectura de memoria. Por otro lado, es necesario otro multiplexor a la entrada del segundo operando de la ALU, pues se puede operar tanto con un segundo registro como con un valor

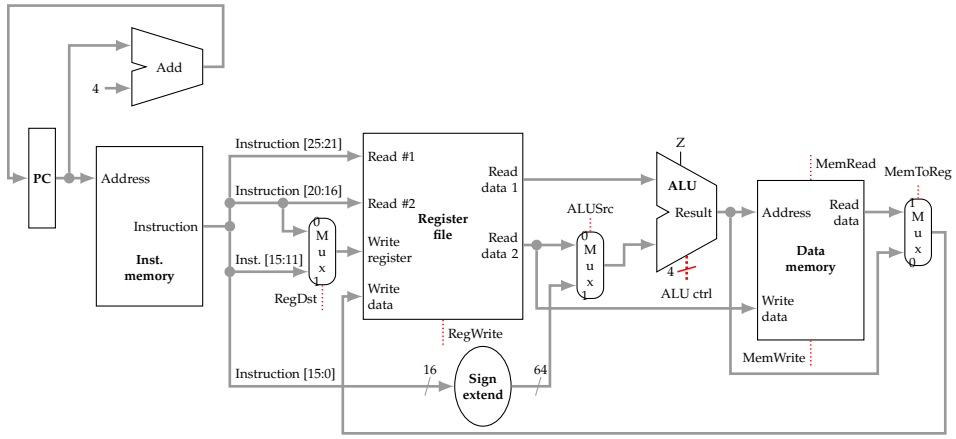


Figura 2.12: Camino de datos para la ejecución de instrucciones de tipo R y de carga/almacenamiento

inmediato. Este multiplexor se controla a través de la línea `ALUSrc`, que estará activada durante las instrucciones de carga y almacenamiento, y desactivada durante las aritmético-lógicas. Por último, existe un multiplexor a la entrada de las líneas `Write data` del fichero de registros, justo a la derecha de la memoria de datos. El objetivo de este multiplexor es seleccionar a través de la línea `MemToReg` los datos a volcar a las líneas `Write data`: bien el resultado de la ALU en las instrucciones aritmético-lógicas, o bien el valor leído en memoria durante una instrucción de carga.

Las siguientes instrucciones a añadir al camino de datos son las instrucciones de salto condicional. Por simplicidad, solo se considera la instrucción **`beq`** `Rs1,Rs2,imm_16`, que salta si el contenido de ambos registros es el mismo. La dirección de salto se calcula como la suma del contador de programa y el valor inmediato desplazado dos bits a la izquierda[2] y con el signo extendido. Un detalle a tener en cuenta es que el valor del registro `PC` ya se habrá incrementado en 4 para apuntar a la siguiente instrucción antes de realizar la suma. Por tanto, las instrucciones de salto condicional realizan dos tareas: comprobar la condición de salto, para lo cual utilizan la ALU, y calcular la dirección de salto, para lo cual necesitan un sumador específico, ya que la ALU ya está en uso. En la figura 2.13 se muestra la lógica necesaria para incluir la instrucción de salto condicional en el camino de datos.

Los dos registros leídos actúan de operandos de entrada en la ALU. Para comprobar la igualdad de ambos basta con aplicar una operación de resta. En ese caso, el *flag* `Z` se activará si el resultado es 0, es decir, si ambos operandos son iguales. Por otra parte, el valor inmediato que se utiliza para calcular la dirección de salto se extiende a 64 bits y se desplaza dos bits a la izquierda, que es equivalente a multiplicarlo por 4. Finalmente, el valor desplazado se suma al valor que tiene el contador de programa, que actualmente apunta a la siguiente instrucción a ejecutar (la instrucción siguiente a **`beq`**). El salto se tomará, y el registro `PC` será escrito con la dirección de destino si se cumple la condición de igualdad (*flag* `Z`) y se activa la señal de control `Branch`.

[2]Esto se hace para que la dirección a sumar al `PC` sea múltiplo de 4, es decir, el tamaño del código de instrucción.

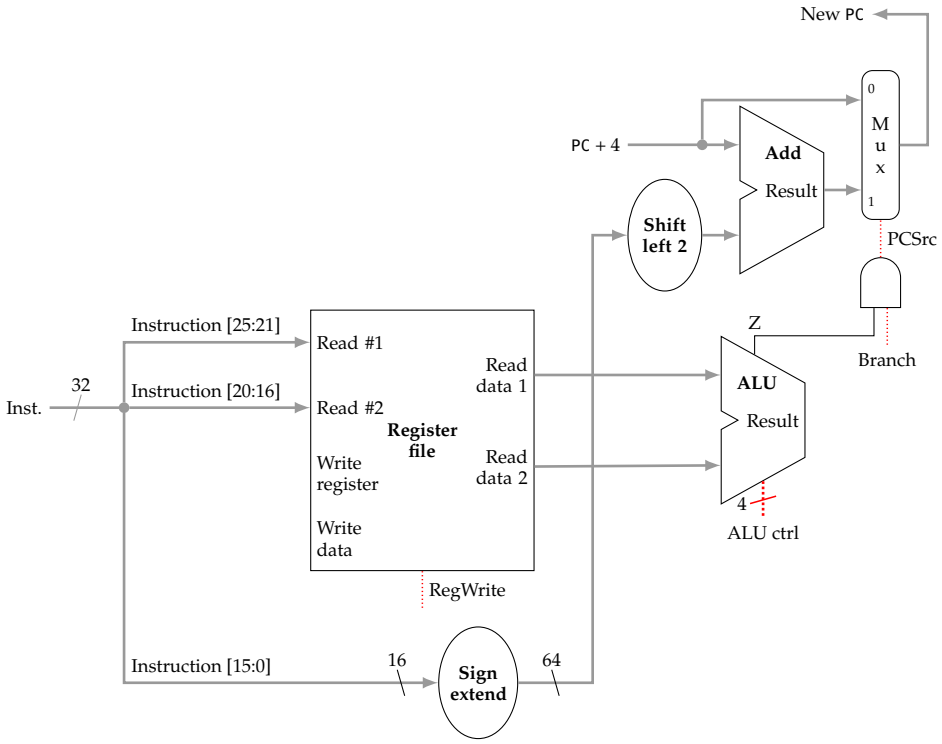


Figura 2.13: Ejecución de la instrucción de salto condicional **beq**

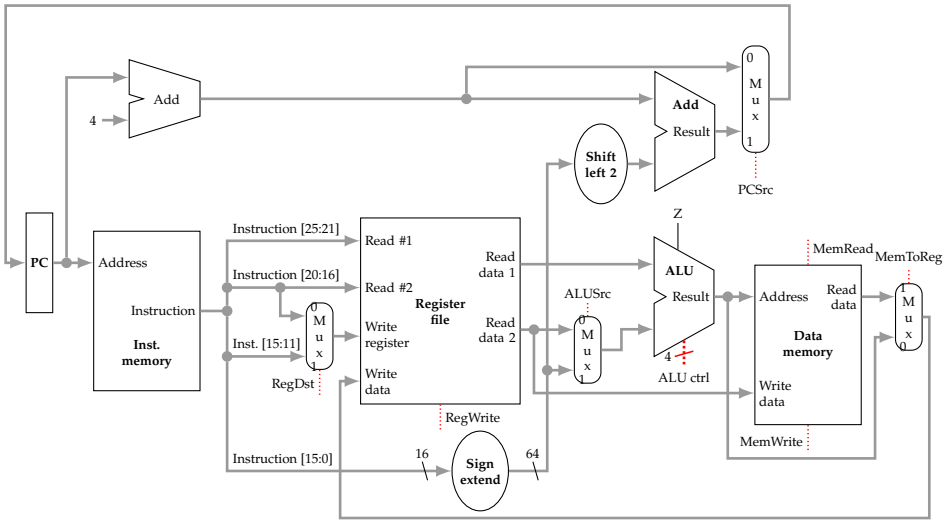


Figura 2.14: Camino de datos para la ejecución de instrucciones de tipo R, carga/almacenamiento y salto condicional **`beq`**

Se puede definir una señal de control genérica `PCSrc` que identifica si se cumple la condición de salto (para contemplar varios tipos de condiciones), que seleccionará el valor a escribir en el registro `PC` a través del multiplexor a la salida del sumador. Será la dirección de la instrucción siguiente si no hay salto y la dirección destino del salto si lo hay.

La integración de la lógica para la ejecución de la instrucción de salto en el camino de datos es relativamente sencilla. La figura 2.14 muestra el aspecto del camino de datos tras la integración. Como se puede apreciar, no es necesario ningún elemento auxiliar adicional.

Por último, toca el turno de añadir las instrucciones de salto incondicional. De nuevo, solo se considera una instrucción por simplicidad: `j imm_26`. Esta instrucción cambia el valor del `PC` incondicionalmente; es un salto que siempre se toma. La dirección de salto se calcula como el valor inmediato de 26 bits desplazado dos bits (28 bits en total) concatenados con los 36 bits más significativos del `PC` para formar un valor de 64 bits. Debe tenerse en cuenta también que el valor del `PC` ya ha sido incrementado antes de realizar el salto. La lógica para implementar el salto incondicional se detalla en la figura 2.15. Esta viene gobernada por una señal de control, denominada `Jump`, que se activa cuando se ejecuta una instrucción de salto incondicional. En otro caso, el valor a escribir en el contador de programa es el valor incrementado en 4.

Para combinar la lógica para ejecutar la instrucción de salto incondicional con la ya existente en el camino de datos no es necesario introducir elementos adicionales. No obstante, debe tenerse en cuenta que el nuevo valor para el contador del programa puede venir determinado por la lógica del salto incondicional o por el multiplexor a la salida del sumador específico para los saltos condicionales. La fi-

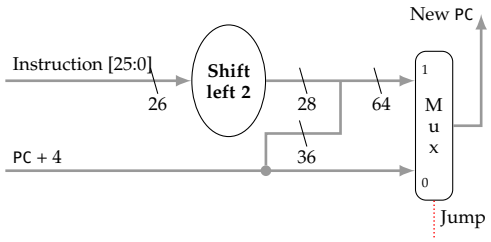


Figura 2.15: Ejecución de la instrucción de salto incondicional `j`

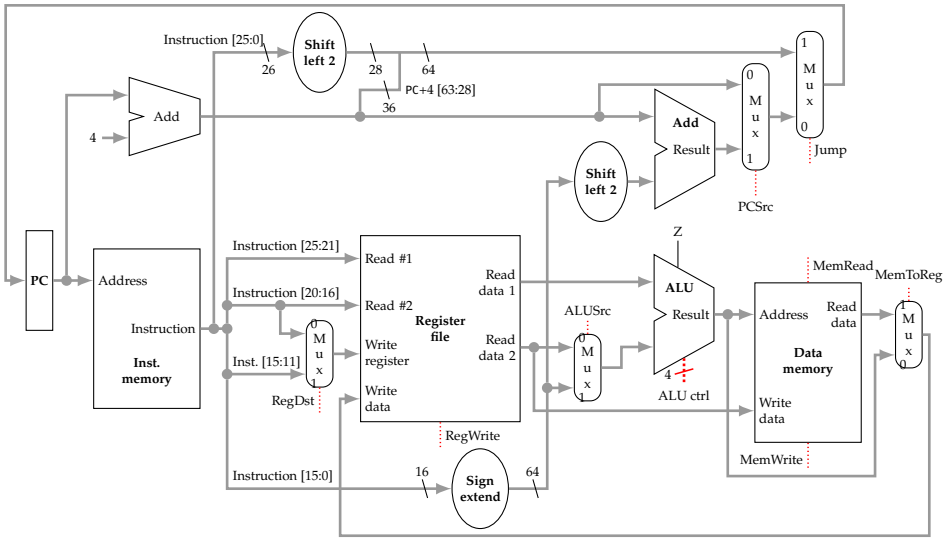


Figura 2.16: Camino de datos para la ejecución de instrucciones de tipo R, acceso a memoria, salto condicional `beq` y salto incondicional `j`

gura 2.16 muestra el camino de datos completo para ejecutar las instrucciones de la tabla 2.1.

En el apéndice B se esboza la construcción de la unidad de control necesaria para el camino de datos desarrollado.

2.2.3. Deficiencias

Si bien una implementación monociclo es posible en la práctica, las CPU actuales no siguen este enfoque. Aunque la CPU monociclo consigue un CPI de 1, el ciclo de reloj debe ser lo suficientemente largo para permitir ejecutar cualquier instrucción, bien sea una instrucción sencilla o una más compleja. Un ejemplo de instrucción simple sería una instrucción de suma, que accede a la memoria de instrucciones, el fichero de registros, la ALU y el fichero de registros de nuevo para guardar el resultado. Por el contrario, una instrucción compleja sería una instrucción de carga, que accede a las mismas unidades funcionales que la anterior, más el acceso a la memoria de datos.

Microarquitectura	CPI	Duración del ciclo (T)
Monociclo	1	Ciclo largo
Multiciclo	>1	Ciclo corto
Segmentada	1	Ciclo corto

Tabla 2.3: Microarquitecturas y ciclos de reloj

Todas las instrucciones necesitarían un ciclo de reloj, por lo que no sería posible optimizar la ejecución de las instrucciones más comunes. Habría que optimizar la ejecución de todas las instrucciones para poder reducir el ciclo de reloj y por lo tanto el tiempo de ejecución de los programas según dicta la ley de hierro.

2.3. Microarquitectura segmentada

La microarquitectura monociclo consigue ejecutar las instrucciones en un único ciclo de reloj, si bien como contrapartida este ciclo es largo para incluir todas las operaciones que se llevan a cabo en el camino de datos. Por otro lado, también es posible una microarquitectura multiciclo donde la ejecución de las instrucciones se divide en pasos, cada uno de los cuales requiere un ciclo de reloj, si bien más corto que para el caso monociclo.

Con las CPU resulta crítico conseguir que las instrucciones se ejecuten tan rápido como sea posible, es decir, que el tiempo de CPU sea el más bajo posible. Este venía fijado por la ley de hierro, vista en el capítulo 1, que se representaba con la siguiente expresión:

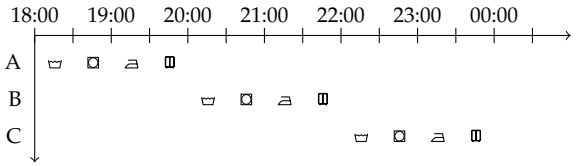
$$\mathrm{T_{CPU}} = \text{Instrucciones del programa} \times \text{CPI} \times T$$

La microarquitectura no afecta al primero de los factores (instrucciones por programa), ya que viene fijado por el juego de instrucciones, luego idealmente el objetivo pasa por ejecutar las instrucciones con el menor CPI posible sin aumentar el periodo de reloj. En esta sección se presentará una técnica, denominada segmentación o *pipelining*, que permite ejecutar instrucciones con un CPI ideal de 1, al mismo tiempo que se mantiene un periodo de reloj corto. Para ello, la ejecución de instrucciones se divide en etapas que trabajan en paralelo, de tal forma que la CPU está ejecutando simultáneamente varias instrucciones, si bien en diferentes etapas. En la tabla 2.3 se resumen las características de varios tipos de microarquitecturas.

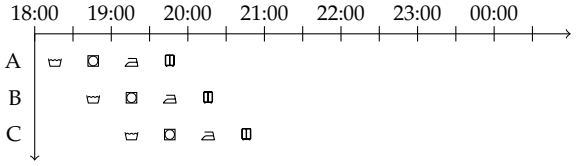
Una gran ventaja de la segmentación es que es totalmente transparente al software. Las instrucciones se ejecutan más rápidamente sin necesidad de modificar las aplicaciones o el sistema operativo. De ahí que hoy en día la práctica totalidad de CPU implementan esta técnica.

La técnica de segmentación es muy sencilla conceptualmente y se aplica en infinidad de ámbitos. Para ilustrar el funcionamiento en detalle de la segmentación se usará otro ejemplo: el proceso de hacer la colada. Este proceso podría ser como sigue:

- Meter la ropa en la lavadora.



(a) No segmentado



(b) Segmentado

Figura 2.17: Proceso de hacer la colada no segmentado y segmentado

- Sacar la ropa de la lavadora y meterla en la secadora.
- Sacar la ropa de la secadora y plancharla.
- Colocar la ropa en el armario.

En la versión no segmentada (secuencial) del proceso solo se empezaría con la segunda colada cuando la ropa de la primera estuviese colocada correctamente en el armario. Intuitivamente puede apreciarse que el proceso no es eficiente, pues recursos como la lavadora, la secadora o la plancha estarían sin utilizar la mayor parte del tiempo. Un ejemplo de este proceso en su versión no segmentada se muestra en la figura 2.17a.

En cambio, en la versión segmentada del proceso se metería la segunda colada en la lavadora justo tras sacar la primera colada y meterla en la secadora. Solo cuando terminasen la lavadora y la secadora se procedería a planchar la primera colada, al tiempo que la segunda pasaría a la secadora y una tercera se metería en la lavadora y, así, sucesivamente. Este proceso se ilustra en la figura 2.17b.

Un aspecto importante a tener en cuenta de la segmentación es la necesidad de que todas las etapas avancen en paralelo, independientemente del tiempo que necesite individualmente cada una de ellas. Por ejemplo, la lavadora podría tener un ciclo de 40 minutos, la secadora de 50 minutos, el planchado podría necesitar 45 minutos y colocar la ropa en el armario 20 minutos (40, 50, 45, 20). Esto significa que la lavadora permanecerá parada durante 10 minutos esperando a que termine la secadora, que es la etapa más lenta, para proceder con la siguiente colada. Está claro que la situación más favorable es aquella en la que todas las etapas están ocupadas todo el tiempo, es decir, las etapas están balanceadas y requieren el mismo tiempo para completarse. En otro caso, el avance del cauce está determinado por la etapa más lenta.

El uso de la segmentación produce una mejora en la productividad (número de coladas por unidad de tiempo) pero no en el tiempo de respuesta (tiempo en realizar una única colada). Suponiendo el caso ideal donde todas las etapas están balanceadas, por ejemplo que todas las etapas de la colada necesitasen 30 minutos, cada

colada todavía necesitará pasar por las cuatro etapas, lo que requeriría un total de 2 horas. No obstante, si el número de coladas a realizar es alto, llegará un momento en el que todas las etapas del proceso estarán trabajando en una colada y, así, cada 30 minutos se terminará de colocar una colada en el armario, es decir, se concluirá el trabajo de una colada. Esto supone una aceleración ideal del rendimiento de 4, que coincide con el número de etapas, lo que quiere decir que idealmente será posible realizar 4 veces más coladas en el mismo tiempo con la versión segmentada del proceso. En este caso ideal no se tienen en cuenta los tiempos en los que hay etapas sin colada (al principio, hasta que se coloca en el armario la primera colada, y al final, desde que la última colada pasa a la secadora y la lavadora pasa a estar ociosa).

Sin embargo, es tremendamente complicado alcanzar la situación ideal en el que todas las etapas requieren el mismo tiempo, es decir, están balanceadas. En la práctica, la mejora en la productividad es menor y el tiempo de respuesta aumenta por los tiempos de espera entre unas etapas y otras. Si volvemos al ejemplo inicial donde cada etapa requiere un tiempo distinto (40, 50, 45, 20), se pueden hacer unos cálculos inmediatos. El tiempo necesario para realizar una colada en modo no segmentado es la suma de los tiempos de cada etapa, es decir, 155 minutos (2h:35min). Considerando que en la versión segmentada todas las etapas deben avanzar en paralelo, el tiempo mínimo de avance de las etapas será 50 minutos. A partir de este tiempo podemos calcular la ganancia en la productividad real (ignorando transitorios con etapas ociosas), que siempre es menor o igual a la ideal.

$$A_{\text{Productividad}} = \frac{155}{50} = 3.1$$

Por su parte, el tiempo de respuesta será la suma de los tiempos de todas las etapas, es decir, 200 minutos, con lo que esta métrica empeora respecto a la versión no segmentada. Es importante resaltar que la segmentación mejora la productividad pero no el tiempo de respuesta en la ejecución de instrucciones, llegando a empeorar este último.

Esta forma de trabajar puede extrapolarse a la ejecución de instrucciones para construir CPU segmentadas. Lo que en el ejemplo anterior es aplicable a las coladas, también lo es a las instrucciones que ejecuta una CPU. En el caso de MIPS, la ejecución de instrucciones se divide comúnmente en 5 etapas:

1. `IF` (***I**nstruction **F**etch*): búsqueda del código de instrucción en la memoria. En esta etapa se lee desde la memoria de instrucciones el código de la instrucción apuntado por el registro contador de programa y se incrementa este último en 4 posiciones para apuntar a la siguiente instrucción.

2. `ID` (***I**nstruction **D**ecode*): decodificación de la instrucción y búsqueda de registros que actuarán como operandos. En esta etapa se procede a la decodificación de la instrucción, se accede al fichero de registros para leer los dos registros que actuarán como operandos (se leen siempre, aunque después se descarten porque la instrucción no los necesite), se extiende el signo del valor inmediato de 16 bits (aunque la instrucción no sea de tipo I) y se desplaza el valor inmediato de 26 bits (aunque no sea de tipo J). En esta etapa también se actualiza el contador de programa con la dirección destino de los saltos incondicionales.

3. `EX` (***EX**ecute*): ejecución en la ALU y cálculo de las direcciones destino de los saltos condicionales. La ejecución en la ALU puede estar asociada a una instrucción aritmético-lógica, a la evaluación de un salto condicional, o al cálculo de la dirección de memoria efectiva en el caso de las instrucciones de carga o almacenamiento.

4. `MEM` (***MEM**ory*): acceso a memoria en las operaciones de carga y almacenamiento. En esta etapa se accede a la memoria de datos y se actualiza el contador de programa con la dirección de salto condicional previamente calculada.

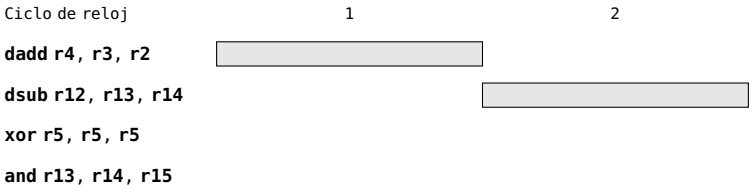
5. `WB` (***W**rite **B**ack*): escritura diferida en el fichero de registros. En esta etapa se procede a escribir en el fichero de registros tras una operación en la ALU o la carga de datos de memoria.

En una CPU segmentada el tiempo necesario para que todas las etapas terminen determina el periodo de reloj de la misma. El camino de datos segmentado, o cauce segmentado, avanzará con cada ciclo de reloj. Así, cuanta menor duración tengan las etapas, menor será el ciclo de reloj de la CPU y, como resultado, mayor será la frecuencia de trabajo. Una consecuencia directa es que puede aumentarse la frecuencia de trabajo de una CPU sin más que incrementar la profundidad del cauce de ejecución, es decir, aumentar el número de etapas. No obstante, como se comentaba anteriormente, es extremadamente complicado conseguir cauces de ejecución con gran número de etapas donde además estén balanceadas, más allá de la compleja gestión que implica un cauce profundo. Además, en cauces profundos aparecen problemas con los saltos condicionales, como se verá más adelante.

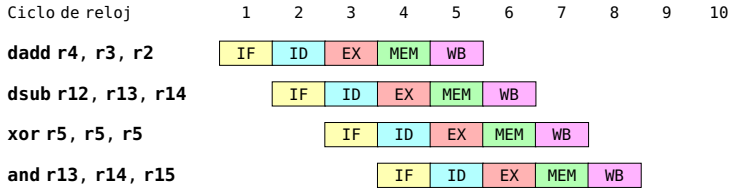
Como conclusión, cabe decir que si bien la segmentación no mejora el tiempo de ejecución de las instrucciones individuales (de hecho, habitualmente lo empeora), sí aumenta la productividad de la CPU al ejecutar las instrucciones. Es importante destacar que la productividad en la ejecución de instrucciones es la métrica más importante de las dos desde el punto de vista práctico, pues los programas se componen de gran cantidad de instrucciones que deben ser ejecutadas y en última instancia se busca que el conjunto de instrucciones que forman los programas se ejecuten lo más rápido posible, y no cada instrucción de forma individual.

En la figura 2.18a se muestra un ejemplo de cómo representar la ejecución no segmentada de instrucciones por parte de la CPU monociclo. Todas las instrucciones se ejecutan secuencialmente y necesitan un ciclo de reloj largo. En cambio, en la figura 2.18b se muestra cómo sería la ejecución de forma segmentada. En este último caso, el CPI observado en condiciones ideales será también 1, pero con un ciclo de reloj mucho más corto que en el caso anterior. Se asume que la CPU segmentada es una versión creada a partir de la versión monociclo dividiendo la ejecución en 5 etapas perfectamente balanceadas. De ahí que la duración del ciclo de la CPU segmentada sea exactamente una quinta parte de la duración del ciclo de la CPU monociclo. Se puede observar cómo en el tiempo requerido por la CPU segmentada para ejecutar las cuatro instrucciones, la CPU monociclo aún estaría ejecutando la segunda instrucción.

Teóricamente, si en lugar de dividir la ejecución de instrucciones en 5 etapas se pudiese dividir en 10, se multiplicaría por 10 la productividad en la ejecución de instrucciones, es decir, el número de instrucciones por segundo que es capaz de ejecutar



(a) Secuencial



(b) Segmentada

Figura 2.18: Ejecución de instrucciones de forma secuencial y segmentada

la CPU segmentada en comparación con la CPU no segmentada. No obstante, el balanceo de las etapas y la gestión de la segmentación dentro de la unidad de control de la CPU se vuelve más compleja cuanto mayor es el número de etapas. En la práctica, las CPU empleadas en los computadores actuales suelen usar entre 10 y 20 etapas segmentadas. Por ejemplo, el procesador Intel Core i7 basado en microarquitectura *Nehalem* dispone de un cauce segmentado de 14 etapas.

Además de la dificultad intrínseca del balanceo de las etapas que hace complicado llegar a alcanzar la aceleración máxima teórica de una CPU segmentada (equivalente al número de etapas), existen otros problemas, denominados **riesgos de la segmentación**, que disminuyen aún más la aceleración obtenida con las CPU segmentadas. Se estudiarán estos riesgos tras analizar una posible implementación segmentada para MIPS64.

2.3.1. Camino de datos segmentado

En la figura 2.19 se muestra el camino de datos monociclo desarrollado en la sección 2.2.2, identificando las 5 etapas en las que se dividirá para implementar la versión segmentada con las etapas descritas en el apartado anterior. El balanceo de las etapas en esta propuesta es alto. Las tres etapas más lentas son `IF`, `EX` y `MEM`, ya que o bien hacen uso de la memoria, o bien se realiza una operación en la ALU. Por contra, las etapas `ID` y `WB` son más rápidas. Ambas etapas acceden al fichero de registros. La etapa `ID` también decodifica la instrucción y calcula la dirección de salto incondicional. No obstante, conviene recordar que todas las etapas deben avanzar al mismo tiempo, por lo que la duración del ciclo de reloj de la CPU viene impuesta por la etapa más lenta.

Como se aprecia en la figura 2.19, el código de la instrucción a ejecutar así como los datos con los que se va operando avanzan desde el lado izquierdo del camino de

datos hacia el derecho con dos excepciones. Por un lado, en la etapa *Write back*, el resultado de una operación en la ALU o el dato leído de memoria debe escribirse en el fichero de registros. Por otro, el avance también es de derecha a izquierda cuando se actualiza el valor del registro `PC` como resultado de un salto condicional que se toma o uno incondicional. Como se verá más adelante, los datos moviéndose de izquierda a derecha en el camino de datos no tienen efectos indeseables sobre las instrucciones actualmente en ejecución, pero sí estos movimientos en sentido contrario.

Otra cuestión a tener en cuenta es que no todas las instrucciones hacen uso de los recursos de todas las etapas, pero sí deben pasar por cada una de estas 5 etapas. Por ejemplo, la instrucción `dadd r3, r2, r1` debe pasar por cada una de las 5 etapas, pero en su etapa `MEM` no accede a memoria, simplemente pasa por dicha etapa sin realizar ninguna acción.

Para permitir que cada una de las etapas del cauce segmentado trabaje sobre una instrucción diferente y que todas avancen al mismo tiempo es necesario introducir nuevos elementos en el camino de datos. Se trata de los denominados **registros de segmentación**, que actúan de *buffer* entre unas etapas y otras. La figura 2.20 muestra el camino de datos con los registros de segmentación. Al principio del ciclo los registros de segmentación y el `PC` se leen para obtener la información que cada etapa requiere para trabajar sobre la instrucción que se encuentra en ella. Justo al final de cada ciclo de reloj los registros de segmentación y el `PC` almacenan la información presente a su entrada. De esta forma, los datos generados por las etapas avanzan de un registro de segmentación a otro y hacia el `PC` en cada ciclo de reloj. Además, para simplificar el esquema se introduce una ligera modificación respecto al camino de datos mostrado en la figura 2.19. Se combinan los dos multiplexores que determinan el valor a escribir en el registro `PC` en uno solo con las tres entradas: `PC`+4, el destino del salto condicional y el destino del salto incondicional.

De vuelta al símil de hacer la colada, un registro de segmentación sería la cesta en que se depositaría la ropa una vez lavada en la lavadora hasta introducirla en la secadora.

Por supuesto, estos registros deben tener un tamaño suficiente para almancenar los datos que fluyen de una etapa a otra. En la implementación de la figura 2.20, el registro `IF/ID` debe tener un tamaño de 96 bits (64 bits del valor incrementado de `PC` y 32 del código de la instrucción), el registro `ID/EX` tiene un tamaño de 256 bits, el registro `EX/MEM` de 193 bits y el registro `MEM/WB` de 128 bits.

A partir de esta implementación inicial, son necesarias una serie de modificaciones para introducir el control segmentado. Esto se aborda en el apéndice B.

2.3.2. Riesgos de la segmentación

Se denominan riesgos de la segmentación a aquellas situaciones que pueden obligar a detener temporalmente la ejecución paralela de instrucciones. Una instrucción se detiene en una etapa cuando no se cumple alguna condición necesaria para completarla. En este caso, el registro de segmentación de entrada de la etapa no se modifica al final del ciclo de reloj, lo que significa que la etapa se repite sobre la misma instrucción en el siguiente ciclo. Esto supone que las etapas anteriores también deben detenerse, impidiendo que modifiquen sus registros de segmentación de salida.

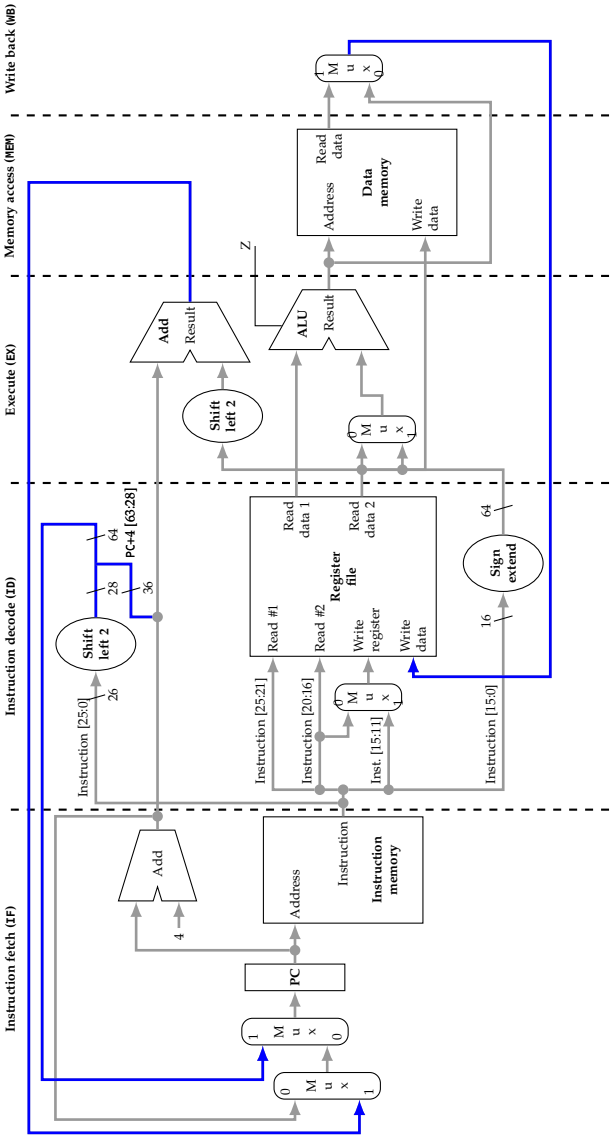


Figura 2.19: Camino de datos monociclo identificando las etapas del cauce segmentado

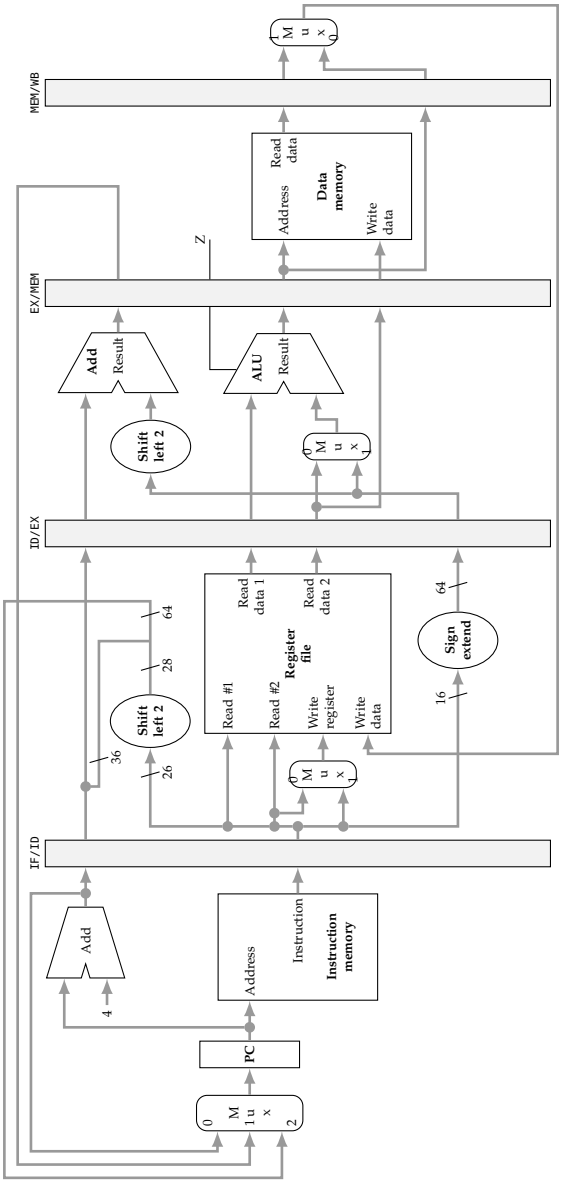


Figura 2.20: Camino de datos segmentado

En cambio, las instrucciones que se encuentren en etapas posteriores a la etapa que causó la detención sí podrán avanzar[3].

No debe confundirse el estado de detención de una etapa en un ciclo de reloj con el de inactividad. En el estado inactivo la etapa no dispone de información válida en su registro de segmentación de entrada y, por lo tanto, no tiene ninguna tarea que realizar. En esta situación la etapa no aparece en los cronogramas en el ciclo de reloj considerado.

Hay tres tipos de riesgos de la segmentación (*pipeline hazards*): estructurales, por dependencia de datos y de control. En los siguientes apartados se detallan cada uno de estos tipos de riesgos.

Riesgos estructurales

Se producen cuando el hardware no es capaz de ejecutar en paralelo una determinada combinación de instrucciones que requieren el uso del mismo recurso.

Es imposible que se produzcan riesgos estructurales en el camino de datos segmentado planteado en la figura 2.20, pues cada etapa utiliza recursos independientes del resto de etapas. Para ilustrar las consecuencias de un riesgo estructural se considera un hipotético camino de datos como el presentado, pero con una única memoria donde se almacenan datos e instrucciones, en lugar de dos, y el siguiente fragmento de código.

```
ld   r11, 120(r0)
dadd r2,  r3,  r6
xor  r15, r15, r15
dsub r4,  r1,  r9
and  r12, r13, r10
```

En la figura 2.21 se muestra el cronograma de ejecución del fragmento de código anterior considerando una única memoria[4]. Como se puede apreciar, la ejecución segmentada es la ideal hasta que llega el momento de proceder con la etapa `MEM` de la instrucción `ld r11, 120(r0)` en el ciclo 4, que requiere leer la memoria (el resto de instrucciones no tienen operandos en memoria). Si la memoria es única, no se podría realizar al mismo tiempo la etapa `IF` de la instrucción `dsub r4, r1, r9` para leer el código de la instrucción, con lo que se produce una detención, de forma que dicha etapa se pospone al siguiente ciclo de reloj. Las detenciones en el cauce (un único ciclo de reloj en el ejemplo) se propagan, ya que las instrucciones siguientes se han visto retrasadas. Por conveniencia las detenciones estructurales se representarán con un aspa, en lugar de mostrar la etapa repetida. El objetivo es simplemente distinguirlas fácilmente de otro tipo de detenciones. Puede verse también en la figura cómo la etapa `ID` está inactiva en el ciclo 5 (el posterior a la detención), de ahí que no aparezca en el cronograma.

La detección de los riesgos estructurales y la gestión de los mismos, deteniendo parte del cauce de ejecución, requiere del empleo de hardware adicional dentro de la microarquitectura MIPS64 estudiada. La forma precisa de hacerlo depende de la implementación particular y queda fuera del alcance de este texto.

[3] La etapa causante de la detención resetea su registro de segmentación de salida, de modo que la siguiente etapa estará inactiva en el siguiente ciclo de reloj.

[4] Esto implicaría modificar el camino de datos añadiendo un nuevo multiplexor para elegir la dirección a utilizar en los accesos a memoria: la que viene de `PC` o la dirección calculada en la etapa `EX`.

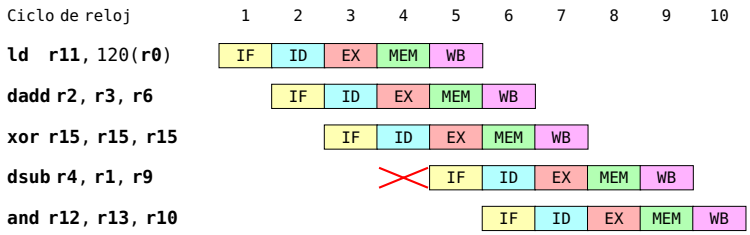


Figura 2.21: Ejecución segmentada asumiendo una única memoria

Riesgos por dependencia de datos

Se dice que existe una dependencia de datos cuando una instrucción referencia a un dato (registro o posición de memoria) de una instrucción anterior. Estas dependencias pueden conducir a detenciones del camino segmentado en el caso de una instrucción que depende del resultado de otra que todavía sigue en el cauce. Esto se aprecia claramente en el siguiente fragmento de código.

```
dadd r2,  r3,  r4
dsub r1,  r10, r2
or   r12, r11, r14
xor  r18, r18, r18
```

La primera instrucción suma los valores contenidos en los registros `r3` y `r4` y almacena el resultado en el registro `r2`. La siguiente instrucción utiliza este valor calculado para realizar la resta. El problema aparece cuando la segunda instrucción procede a leer el valor del registro `r2` (etapa `ID`), ya que el resultado de la suma todavía no ha sido escrito en el registro. Este resultado se escribe en el registro durante la etapa `WB` de la primera instrucción, de modo que es necesario detener el cauce de ejecución hasta que pueda leerse del registro `r2` el valor correcto.

La figura 2.22 muestra el cronograma de ejecución de las instrucciones teniendo en cuenta la detención por dependencia de datos. Se observa cómo la etapa `ID` de la instrucción `dsub` debe repetirse continuamente hasta que se lee el valor correcto de `r2` en el ciclo de reloj 5. Se produce simultáneamente la escritura del registro `r2` en la etapa `WB` de la instrucción `dadd` y la lectura del mismo registro `r2` en la etapa `ID` de la instrucción `dsub`. Este comportamiento es posible dado que la escritura en el archivo de registros se realiza en la etapa `WB` durante la primera mitad del ciclo de reloj y la lectura en la etapa `ID` durante la segunda mitad del ciclo de reloj. Sin embargo, este solapamiento de una escritura y una lectura en el mismo ciclo de reloj no es posible en el caso de la memoria.

Puede observarse también cómo la detención del cauce implica que la instrucción posterior (`or`) continúa en su etapa `IF` hasta que la instrucción `dsub` puede continuar.

Un matiz importante a tener en cuenta es que no es necesario que dos instrucciones que mantengan una dependencia de datos sean consecutivas para que se produzca la detención. En el caso anterior, se produciría dicha detención incluso aunque existiese entre ellas una instrucción. Las detenciones en el cauce por dependencia de datos entre instrucciones reducen la aceleración en el rendimiento que introduce la

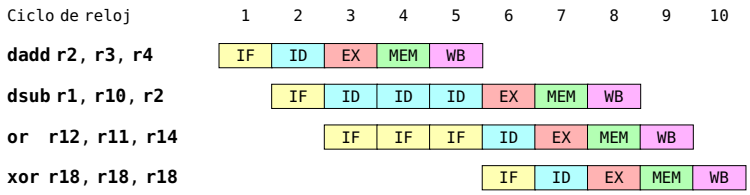


Figura 2.22: Ejecución segmentada con una detención por dependencia de datos

segmentación de instrucciones. En el ejemplo anterior, se necesitan 10 ciclos para ejecutar el fragmento de código en lugar de 8 si no existiese la detención.

La dependencia de datos mostrada es sólo una de las posibles. A continuación, se describen formalmente los tipos de dependencias de datos que pueden producirse entre instrucciones.

- Lectura después de escritura o RAW (*Read After Write*). Es la dependencia que aparece en el ejemplo visto en el listado anterior. Se produce cuando una instrucción lee un registro que ha sido previamente escrito por una instrucción anterior. A este tipo de dependencia también se la conoce como dependencia verdadera.

  La unidad de control debe detectar esta dependencia y verificar si existe un riesgo. En ese caso, debe detener el *pipeline* los ciclos necesarios hasta que el riesgo desaparezca. En la microarquitectura MIPS64, solo existen riesgos por dependencia de datos para el caso de operandos en registros, pero no para operandos en memoria, ya que solo se accede a memoria en las etapas `MEM` de las instrucciones de carga y almacenamiento (y no se pueden llevar a cabo dos etapas `MEM` al mismo tiempo). Se puede ver con el siguiente código de ejemplo.

```
sd r3, 120(r0)
ld r5, 120(r0)
```

  La instrucción **`ld`** lee la posición de memoria `120(r0)` después de ser escrita por la instrucción **`sd`**. No es necesario comprobar si esta dependencia conduce a detención, pues la lectura y escritura de la memoria de datos se llevan a cabo en la misma etapa, la etapa `MEM`, y siempre llegará a esta etapa en primer lugar la instrucción **`sd`** y a continuación la instrucción **`ld`**. Esta característica tan beneficiosa no es fruto de la casualidad; es una consecuencia de un juego de instrucciones RISC en el que solo acceden a memoria las instrucciones de carga y almacenamiento. Es una prueba de que el juego de instrucciones se diseña para simplificar la segmentación, lo que reduce el coste del procesador e incrementa su rendimiento.

- Escritura después de lectura o WAR (*Write After Read*). También se la conoce como antidependencia. Considérese ahora este otro fragmento de código.

```
dsub r1,  r10, r2
dadd r2,  r3,  r4
or   r12, r11, r14
```

De acuerdo a la semántica del programa, la escritura en el registro `r2` por parte de la instrucción `dadd` debe ser posterior a la lectura de dicho registro por parte de la instrucción `dsub`.

La escritura de un registro en la microarquitectura MIPS64 se hace en la etapa `WB` y la lectura en la etapa `ID`. Como la instrucción `dsub` va a llegar a la etapa `ID` antes que la instrucción `dadd` a la etapa `WB`, este riesgo de la segmentación no es posible.

En el caso de operandos de memoria la dependencia WAR nunca puede provocar detenciones de la segmentación, pues tanto la lectura como la escritura se llevan a cabo en la misma etapa, la etapa `MEM`.

Sin embargo, como se verá posteriormente, una de las técnicas de mejora de la segmentación es la ejecución fuera de orden. Empleando esta técnica podría comenzar la ejecución de la instrucción de escritura, antes que la de lectura, por lo que en ese caso sí sería necesario comprobar los riesgos que generan las dependencias WAR.

- Escritura después de escritura o WAW (*Write After Write*). Establece el orden en el que deben llevarse a cabo dos escrituras sobre el mismo dato. Recibe también el nombre de dependencia de salida. Considérese ahora el fragmento de código siguiente.

```
dsub r1, r10, r2
and  r1, r3,  r4
xor  r7, r1,  r6
```

Según la semántica del programa, la escritura de la instrucción `and` sobre el registro `r1` debe ser posterior a la escritura sobre ese mismo registro de la instrucción `dsub`, pues en caso contrario la instrucción `xor`, o cualquier otra posterior que use `r1`, recibirá un valor incorrecto. De nuevo, en el caso de la microarquitectura MIPS64 planteada se puede ignorar este tipo de dependencia, pues sólo hay una etapa que escribe en los registros: la etapa `WB`. En el caso de operandos de memoria ocurre lo mismo, sólo la etapa `MEM` escribe en memoria. Sin embargo, al igual que con la dependencia WAR, si se implementa la posibilidad de ejecución fuera de orden, esta dependencia produciría riesgos que deberían comprobarse.

Al igual que ocurre con la detección de los riesgos estructurales y la gestión de los mismos, el tratamiento de las detenciones por dependencias de datos requiere el empleo de hardware adicional dentro de la microarquitectura MIPS64 estudiada. La forma precisa de hacerlo depende de la implementación particular y queda fuera del alcance de este texto.

Riesgos de control

Los riesgos de control se producen en situaciones de cambios en el flujo de ejecución tales como llamadas a función, saltos, interrupciones o excepciones. Bajo estas condiciones, se interrumpe el flujo de ejecución secuencial, lo que impide a la CPU ejecutar las etapas iniciales de las siguientes instrucciones hasta que se resuelva el

destino de un salto, interrupción, excepción, etc. El siguiente listado de código ilustra esta situación.

```
      beq   r0, r3, loop
      daddi r1, r0, 1
      and   r3, r2, r4
      xor   r8, r8, r8
loop: dsub  r3, r1, r2
```

En primer lugar, aparece una instrucción de salto condicional. Dependiendo de si el salto se toma o no, la siguiente instrucción a ejecutar será **dsub** o la posterior a la instrucción de salto (**daddi**), respectivamente. El problema es que el registro PC se actualiza con la dirección de la siguiente instrucción a cargar en la etapa MEM de la instrucción **beq**. De esta forma, habría que detener el cauce de ejecución hasta determinar cuál será la siguiente instrucción a ejecutar. En la figura 2.23 se muestra el cronograma de ejecución del código anterior en los supuestos de que el salto se tome o no se tome.

Resulta conveniente indicar que cuando la condición no se cumple podría adelantarse un ciclo la etapa ID de la instrucción **daddi**, del ciclo 6 al ciclo 5, reduciendo la detención de 3 a 2 ciclos de reloj. Al final del ciclo 4 la etapa IF ya ha buscado la instrucción **daddi** y no sería necesario repetir la búsqueda de esta instrucción. No obstante, esta optimización complicaría la microarquitectura MIPS64 estudiada, pues sería necesario tener en cuenta si el salto se ha cumplido o no para adelantar en su caso la etapa ID. Por simplicidad esta optimización no se contemplará.

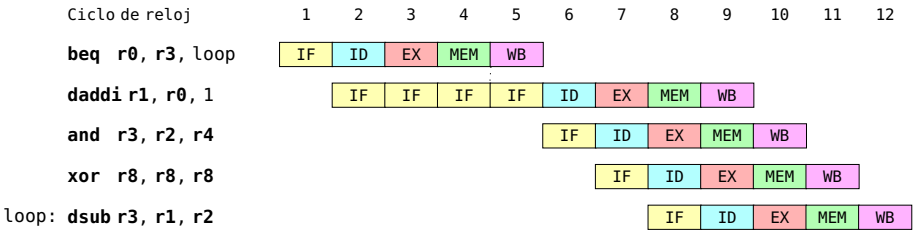
Para el caso de los saltos incondicionales, el registro PC se actualiza en la etapa ID, por lo que la duración de la detención sería menor: un único ciclo. Si se sustituye la instrucción **beq** del listado anterior por `j loop`, el cronograma de ejecución sería el mostrado en la figura 2.24.

Los riesgos de control son un caso particular de dependencia RAW relacionada con el registro PC. El cálculo de la dirección y la condición de salto se realiza en la etapa EX para los saltos condicionales. Sin embargo, no es hasta la etapa siguiente, la etapa MEM, cuando se actualiza el PC, por lo que a efectos prácticos la evaluación de un salto condicional se produce en la etapa MEM. La etapa de escritura en el PC, la etapa MEM, es posterior a la etapa de lectura del PC, la etapa IF, de ahí la dependencia por riesgo de datos (respecto al PC). En el caso de los saltos incondicionales ocurre lo mismo, pero la detención es menor, pues la modificación del PC se produce en la etapa ID.

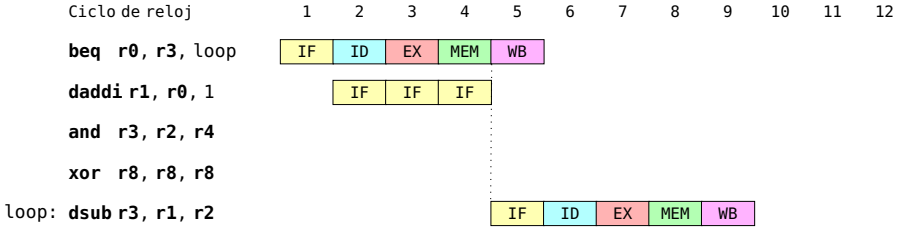
Al igual que ocurría con las detenciones por dependencias de datos RAW, la detención implica repetir la etapa que la sufre (IF) y las anteriores (que no existen) hasta que el valor del registro PC está disponible.

2.3.3. Operaciones multiciclo

Hasta ahora, se ha considerado que la microarquitectura MIPS64 en estudio implementa instrucciones que requieren un único ciclo en la etapa EX. Por ejemplo, la ALU de la etapa EX es capaz de realizar una suma y una resta de enteros en un ciclo de reloj. El problema reside cuando se pretenden implementar operaciones más complejas como son la multiplicación y división de enteros u operaciones de punto



(a) Salto condicional no tomado



(b) Salto condicional tomado

Figura 2.23: Ejecución segmentada con una detención por riesgo de control en un salto condicional

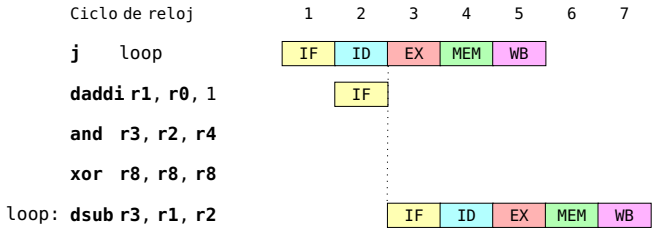


Figura 2.24: Ejecución segmentada con una detención por riesgo de control en un salto incondicional

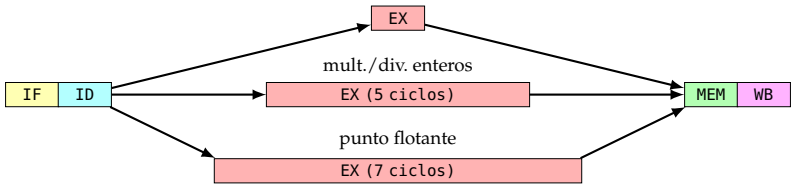


Figura 2.25: Introducción de operaciones multiciclo de enteros y punto flotante

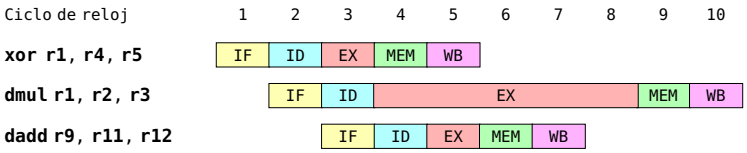


Figura 2.26: Mejoras de rendimiento empleando varias unidades de ejecución

flotante. Todas estas operaciones requieren más tiempo (teóricamente varios ciclos de reloj) para completarse.

Una solución obvia sería incrementar la duración del ciclo de reloj para que su duración permitiese realizar la operación más larga, pero sería inviable por el bajo rendimiento obtenido, ya que ralentizaría todo el cauce segmentado. Una solución más acertada consiste en añadir etapas EX multiciclo para la realización de operaciones complejas. Por ejemplo, podría añadirse una etapa EX multiciclo para la realización de multiplicaciones y divisiones de enteros y otra para operaciones de punto flotante, como se ejemplifica en la figura 2.25.

Si la instrucción realiza una multiplicación o división de enteros la etapa EX empleada será la etapa EX multiciclo de multiplicación/división de enteros de la figura. Si fuese una operación sobre operandos de punto flotante, se emplearía la etapa EX multiciclo de punto flotante. En cualquier otro caso, se emplearía la etapa EX monociclo habitual.

El empleo de varias unidades de ejecución abre la puerta a mejoras de rendimiento. Por ejemplo, una instrucción de suma de enteros puede ser emitida a su unidad de ejecución al mismo tiempo que una instrucción de multiplicación o división de enteros está siendo ejecutada en su unidad correspondiente. Para dar soporte a esta mejora cada unidad de ejecución debe tener su propio registro de segmentación de entrada. Esta situación se muestra en la figura 2.26 para el fragmento de código siguiente suponiendo que la etapa EX de multiplicación/división de enteros requiere cinco ciclos de reloj.

```
xor  r1, r4,  r5
dmul r1, r2,  r3
dadd r9, r11, r12
```

La instrucción **dadd** entra en la etapa EX (es emitida) antes de que la instrucción **dmul** que le precede abandone la suya, pues emplea la unidad de ejecución general, que

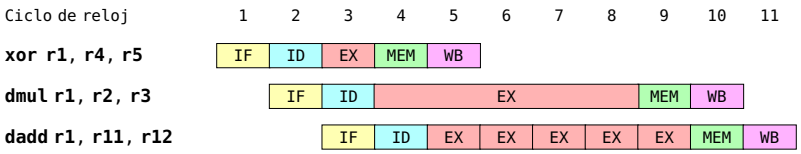


Figura 2.27: Ejemplo de detención WAW ocasionada por unidades de ejecución multiciclo

está disponible. Se dice que ambas instrucciones se emiten y ejecutan en orden (pasan a la etapa `EX` y comienzan la ejecución en esta etapa en el orden del programa)[5]. Además, la instrucción **`dadd`** termina antes que la instrucción **`dmul`** que le precede, por lo que se dice que ambas instrucciones terminan (o son retiradas) fuera de orden. En cualquier caso, las instrucciones siempre se cargan (`IF`) y se decodifican (`ID`) en orden, pero la emisión, la ejecución (`EX`) y la terminación (`MEM` y `WB`)[6] pueden realizarse fuera de orden.

La emisión, ejecución y terminación (o retirada) de instrucciones fuera de orden mejora el rendimiento de los programas a costa de complicar la microarquitectura, pues la gestión de las dependencias de datos se vuelve más compleja. Por ejemplo, si la instrucción **`dadd r9, r11, r12`** se sustituyese por la instrucción **`dadd r1, r11, r12`**, no podría terminar antes que la instrucción que la precede, pues hay una dependencia WAW entre ambas en el registro **`r1`**. En caso contrario el valor del registro **`r1`** al final del programa sería erróneo. La figura 2.27 muestra la detención de 4 ciclos WAW causada por la escritura del registro **`r1`**. Esta detención en última instancia ocasiona un incremento en el tiempo de ejecución de un ciclo de reloj.

La ejecución de operaciones multiciclo trae consigo otro efecto que no existía en la microarquitectura MIPS64: las detenciones por riesgos estructurales son ahora posibles en la etapa `EX`. Para ilustrarlo considérese el siguiente fragmento de código.

```
xor  r1, r4,  r5
dmul r1, r2,  r3
dmul r8, r9,  10
dadd r9, r11, r12
```

La segunda instrucción **`dmul`** no puede entrar en la etapa de ejecución hasta que la primera instrucción **`dmul`** la libere, lo que le supone una detención estructural de 4 ciclos. El cronograma de ejecución sería el mostrado en la figura 2.28. Esta detención supone una penalización de rendimiento, pues incrementa el CPI. Obsérvese cómo la detención estructural detiene el cauce y se propaga a las instrucciones siguientes, en este caso la instrucción **`dadd r9, r11, r12`**. La detención de una instrucción supone la detención de todas las que le siguen. Esta restricción será eliminada posteriormente en el apartado dedicado a la microarquitectura superescalar.

Para reducir o eliminar la detención estructural es posible segmentar la etapa `EX` de multiplicación/división de enteros en tantas etapas como ciclos necesite, de tal

[5]Técnicamente, la emisión es el envío de una instrucción a su unidad de ejecución correspondiente, mientras que la ejecución es el momento en el que la instrucción está ejecutándose en esta unidad. Pueden existir *buffers* que independicen estos dos procesos.

[6]Las etapas de terminación son aquellas que modifican el estado de la CPU y la memoria.

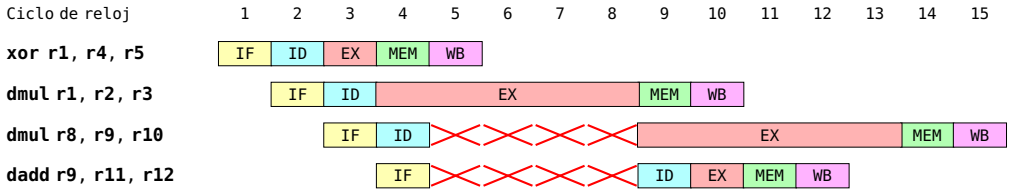


Figura 2.28: Ejecución de operaciones multiciclo no segmentadas

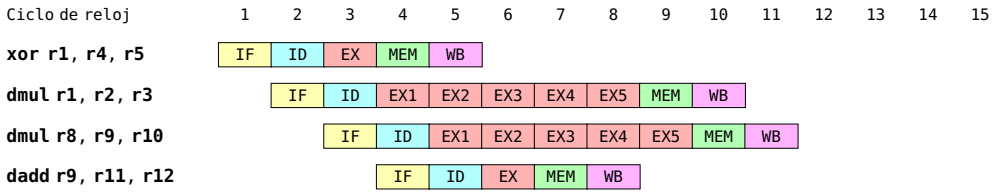


Figura 2.29: Ejecución de operaciones multiciclo segmentadas

forma que puedan trabajar en paralelo. La técnica de la segmentación puede aplicarse a cualquier sistema digital, no sólo a la CPU. La figura 2.29 muestra la ejecución una vez se ha segmentado la unidad de multiplicación/división de enteros. Puede observarse una mejoría significativa, pues ha desaparecido la detención por riesgo estructural entre las dos instrucciones de multiplicación.

2.3.4. Gestión de excepciones

Las excepciones producen cambios en el flujo normal de ejecución del programa. Dependiendo de la arquitectura también se utiliza el nombre de fallos o interrupciones. Una vez que ocurre una excepción, se detiene la ejecución del programa y pasa a ejecutarse una rutina del sistema operativo y, en general, prosigue la ejecución del programa donde se había dejado. Las excepciones pueden ocurrir durante la ejecución de las instrucciones o justo al acabar la mismas.

Por ejemplo, una instrucción de división por cero no puede ejecutarse, de igual forma que una instrucción cuyo código de instrucción sea incorrecto. En ambos casos se produce una excepción. En el contexto de la segmentación también se consideran excepciones las interrupciones solicitadas por la interfaz de un periférico.

En las microarquitecturas segmentadas se busca un comportamiento ante excepciones análogo al de las máquinas no segmentadas. Las instrucciones posteriores a la que generó la excepción que hubiesen entrado en el cauce deberían detenerse. Si la excepción ocurre en medio de una instrucción esta misma instrucción también debería detenerse. Las instrucciones anteriores por otra parte deberían completarse. Cuando se consigue este objetivo se dice que la microarquitectura segmentada soporta **excepciones precisas**.

Las CPU actuales soportan excepciones precisas, pero no así algunas CPU segmentadas antiguas. Estas requerían soporte por parte de las rutinas de servicio de

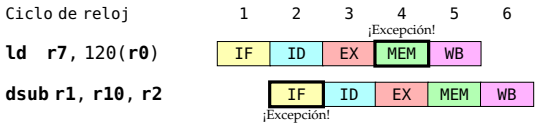


Figura 2.30: Múltiples excepciones en la ejecución segmentada

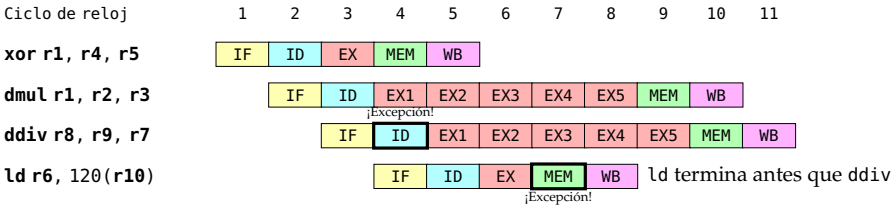


Figura 2.31: La terminación fuera de orden puede ocasionar excepciones no precisas

las excepciones para conseguir un funcionamiento correcto ante determinados tipos de excepciones. En esos casos recaía en el sistema operativo la responsabilidad de que la ejecución de las instrucciones fuese la correcta en presencia de excepciones.

El problema de las excepciones en microarquitecturas segmentadas es que se pueden producir en diferentes etapas por parte de diferentes instrucciones. Por ejemplo, el programa siguiente podría dar lugar a una excepción en la etapa `MEM` de la instrucción **`ld`** y a una excepción en la etapa `IF` de la instrucción **`dsub`**, tal como se muestra en la figura 2.30.

```
ld   r7, 120(r0)
dsub r1, r10, r2
```

La excepción de la instrucción **`ld`** ocurre después que la excepción de la instrucción **`dsub`**, pero debería procesarse antes, pues la instrucción **`ld`** aparece antes en el programa. Si las excepciones se procesasen justo en el momento en el que aparecen no serían precisas.

La solución consiste en procesar las excepciones no en el momento en que se producen, sino más tarde cuando la ejecución de la instrucción llega a la última etapa: la etapa `WB`. De esta forma, llegará antes la excepción de la instrucción **`ld`** que la excepción de la instrucción **`dsub`**. Para que la instrucción avance por el cauce hasta llegar a la etapa `WB` es necesario que el cauce no se detenga, lo que obliga a desactivar las escrituras en registros y memoria para la instrucción de la excepción y todas las que le siguen en el cauce, evitando de esta forma que modifiquen el estado de la máquina.

Otro problema asociado a las excepciones es la terminación de instrucciones fuera de orden, pues puede llegar antes a la etapa `WB` una instrucción posterior a la que generó la excepción. La figura 2.31 ilustra esta situación.

Para implementar excepciones precisas es necesario que las instrucciones terminen en orden. La terminación o retirada de las instrucciones se lleva a cabo cuando escriben y por lo tanto modifican el estado de la máquina, esto es, en las etapas `MEM`

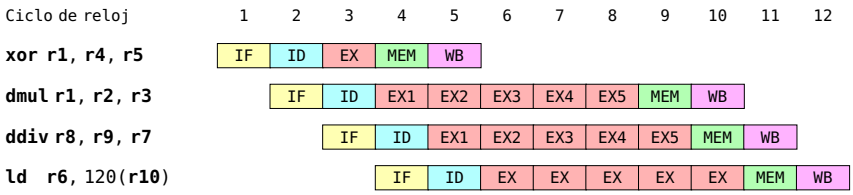


Figura 2.32: Las excepciones precisas requieren terminación en orden

y WB. La figura 2.32 ilustra la terminación en orden para el mismo ejemplo anterior y cómo esta restricción induce una pérdida de rendimiento.

Aunque el soporte de excepciones precisas exige la terminación de instrucciones en orden, la emisión y la ejecución de instrucciones puede ser fuera de orden, proporcionando oportunidades de mejora del rendimiento, especialmente en el caso de las CPU superescalares, que se estudiarán en la sección 2.4.2.

2.3.5. Reducción de detenciones por dependencias de datos

Las CPU segmentadas son capaces de retirar un máximo de una instrucción por cada ciclo de reloj en condiciones ideales. No obstante, como se ha visto, existen riesgos de la segmentación que impiden alcanzar este valor en la práctica, por lo que el CPI de estas CPU será siempre mayor que 1. Para el caso de la microarquitectura MIPS64 segmentada en estudio, solo son posibles detenciones por dependencias de datos y por riesgos de control. Se pueden producir penalizaciones de hasta tres ciclos de reloj, lo que ocasiona un incremento del CPI medio de los programas.

En este apartado se muestran las técnicas básicas empleadas para reducir el impacto de las dependencias de datos entre instrucciones para acercar el rendimiento del *pipeline* a su máximo teórico, tomando como base la microarquitectura MIPS64 estudiada. Las técnicas para minimizar las detenciones por riesgos de control se verán en el siguiente apartado. Se plantearán tres técnicas para reducir o eliminar las detenciones por dependencias de datos:

- La planificación de instrucciones, que puede realizarse por parte del compilador, en cuyo caso no requiere cambios en la microarquitectura, o por la microarquitectura directamente.
- Las rutas de reenvío, que se implementan por hardware y son transparentes al software.
- El renombrado de registros, que también se implementa por hardware y es transparente al software.

Todas estas técnicas tienen en común que eliminan o reducen las detenciones sin alterar la semántica del programa, esto es, sin que los resultados del programa varíen. El renombrado de registros busca eliminar dependencias, mientras que las otras dos técnicas mantienen las dependencias pero buscan reducir las detenciones que ocasionan.

En este punto conviene recordar los diferentes tipos de dependencias de datos:

- Lectura después de escritura o RAW (*Read After Write*). Una posición de memoria o un registro no puede leerse hasta que una instrucción anterior haya escrito su valor.
- Escritura después de lectura o WAR (*Write After Read*). Una instrucción de lectura de una posición de memoria o registro debe ejecutarse antes que una instrucción de escritura posterior. Debe evitarse que por causa de una ejecución fuera de orden la instrucción de escritura se ejecute antes que la de lectura.
- Escritura después de escritura o WAW (*Write After Write*). Si una instrucción de escritura de una posición de memoria o registro precede a otra instrucción de escritura, no puede ocurrir que por una ejecución fuera de orden la primera instrucción de escritura se ejecute después de la última.

La dependencia de datos RAW recibe el nombre de dependencia verdadera, porque supone una transferencia de datos de una instrucción a otra dentro de la semántica del programa, a diferencia de lo que ocurre con las dependencias WAR y WAW. Estas dos últimas son dependencias artificiales que se producen por la reutilización de registros por parte del compilador.

Planificación de instrucciones

La planificación de instrucciones (*instruction scheduling*) es una técnica que implica reordenar las instrucciones a ejecutar de forma que las dependencias de datos no produzcan detenciones y la semántica del programa no cambie. Esto supone mover instrucciones para situarlas entre instrucciones que mantengan una dependencia para «alejarlas» y así reducir o evitar la detención, respetando en todo caso las dependencias de datos del programa original.

Considérese parte del fragmento de código empleado para ilustrar las dependencias RAW en la microarquitectura MIPS64, el cual da lugar a una detención de dos ciclos de reloj.

```
dadd r2,  r3,  r4
dsub r1,  r10, r2
or   r12, r11, r14
```

Las dos primeras instrucciones mantienen una dependencia sobre el registro `r2`. La instrucción `or r12, r11, r14` es independiente de las dos que mantienen la dependencia, pues utiliza registros distintos a los usados por estas. Por esta razón, esta instrucción podría ubicarse entre las instrucciones dependientes sin que cambie la funcionalidad del programa, tal como puede verse a continuación.

```
dadd r2,  r3,  r4
or   r12, r11, r14
dsub r1,  r10, r2
```

La detención por la dependencia de datos pasaría de dos ciclos a un ciclo, como puede observarse en la figura 2.33, en la que la etapa `ID` de la instrucción `dsub` se repite una vez. De hecho, si se pudiesen colocar dos instrucciones independientes entre las instrucciones `dadd` y `dsub`, no existiría detención.

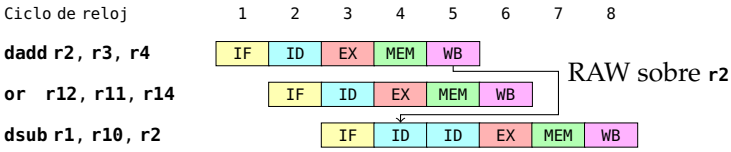


Figura 2.33: Efecto de la reordenación de instrucciones

Esta técnica puede implementarse a nivel software en el compilador o a nivel hardware. En el primer caso, se habla de planificación estática de instrucciones, ya que se realiza en tiempo de compilación. El inconveniente de la implementación por parte del compilador no es solo el incremento de su complejidad, sino que además puede ser necesario recompilar los programas para adaptarlos a cada microarquitectura, incluso cuando la arquitectura del juego de instrucciones de la CPU no cambia.

Cuando la planificación de instrucciones se implementa a nivel hardware es la microarquitectura la que toma las decisiones de reordenamiento de las instrucciones de forma dinámica según se van ejecutando y apareciendo dependencias. En ese caso se habla de planificación dinámica de instrucciones.

Rutas de reenvío

Las rutas de reenvío, adelantamiento o *forwarding*, son conexiones en el camino de datos que se emplean para reducir las detenciones por dependencias de datos RAW, que en la microarquitectura MIPS64 descrita solo son posibles por dependencias entre registros (nunca de memoria). La escritura de registros se lleva a cabo en la primera mitad de la etapa `WB` y la lectura de los mismos en la segunda mitad de la etapa `ID`. La mayor penalización por la detención es de dos ciclos y aparece cuando la instrucción de escritura y la de lectura del registro son consecutivas, tal como ocurre en el siguiente fragmento de código, representado en la figura 2.22 de la página 51.

```
dadd r2,  r3,  r4
dsub r1,  r10, r2
or   r12, r11, r14
xor  r18, r18, r18
```

El valor del registro `r2` que necesita la instrucción **`dsub`** se calcula cuando la instrucción **`dadd`** pasa por la etapa `EX`, pero se escribe en el registro `r2` en la primera mitad del ciclo de la etapa `WB`. Si de alguna forma fuese posible llevar el resultado de la etapa `EX` a la entrada de esta misma etapa para ser usado en el siguiente ciclo de reloj, en lugar del proporcionado por la etapa `ID`, la instrucción **`dsub`** podría entrar en el siguiente ciclo de reloj en la etapa `EX` y ejecutarse sin detenciones.

La figura 2.34 muestra la ejecución del fragmento anterior cuando se dispone de una ruta de reenvío desde la salida de la etapa `EX` a su entrada.

El camino de datos incluyendo esta ruta de reenvío es fácilmente implementable. Basta incluir un multiplexor que lleve a la entrada de la etapa `EX`, o bien el valor del registro presente a la salida de la etapa `ID` (salida del registro de segmentación `ID/EX`), o bien a la salida de la etapa `EX` (salida del registro de segmentación `EX/MEM`).

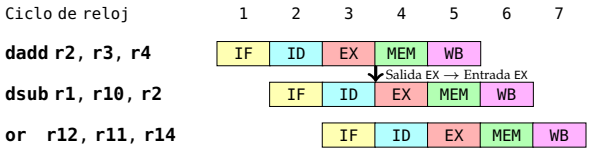


Figura 2.34: Ejecución con la ruta de reenvío Salida EX → Entrada EX

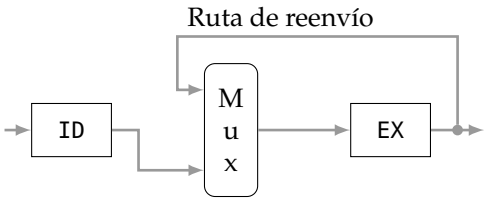


Figura 2.35: Implementación de la ruta de reenvío Salida EX → Entrada EX

La figura 2.35 muestra conceptualmente cómo debe modificarse el camino de datos para soportar esta ruta de reenvío.

Cuando las instrucciones dependientes no son consecutivas es necesario definir otra ruta de reenvío. Por ejemplo, en el fragmento siguiente hay una instrucción independiente entre las dos instrucciones dependientes anteriores.

```
dadd r2,  r3,  r4
xor  r6,  r11, r10
dsub r1,  r10, r2
or   r12, r11, r14
```

Cuando la instrucción **dsub** está a punto de entrar en la etapa EX la ruta de reenvío que va desde la salida de la etapa EX a la entrada de la misma etapa debe estar desactivada, pues el valor del registro **r2** no está a la salida de la etapa EX, sino a la salida de la etapa siguiente, la etapa MEM. Debe tenerse en cuenta que a medida que pasan los ciclos de reloj los resultados de la etapa EX se mueven a través de los registros de la segmentación EX/MEM y MEM/WB para llegar finalmente a la etapa WB.

La figura 2.36 muestra la ejecución del fragmento anterior cuando se dispone de una ruta de reenvío desde la salida de la etapa MEM (salida del registro de segmentación MEM/WB) a la entrada de la etapa EX. En este caso se ha pasado de una detención de un ciclo de reloj a que no exista detención, pues el valor de **r2** está disponible al comienzo de la etapa EX.

En el caso de que hubiese dos instrucciones entre las instrucciones dependientes la situación sería análoga a la del fragmento de código siguiente:

```
dadd r2,  r3,  r4
xor  r6,  r11, r10
ld   r7,  120(r15)
dsub r1,  r10, r2
or   r12, r11, r14
```

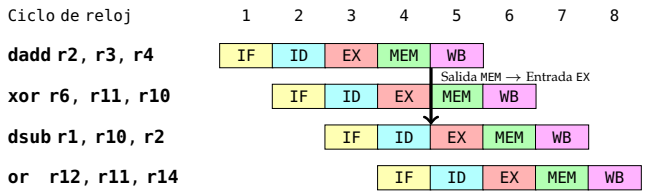


Figura 2.36: Ruta de reenvío Salida MEM → Entrada EX

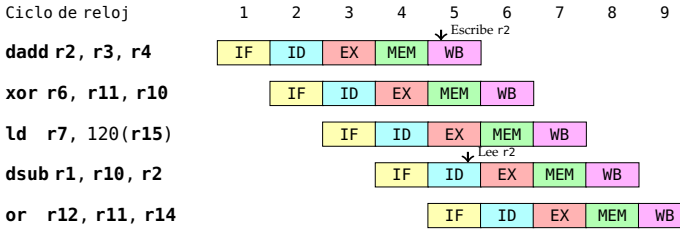


Figura 2.37: La ruta de reenvío Salida WB → Entrada EX no es necesaria

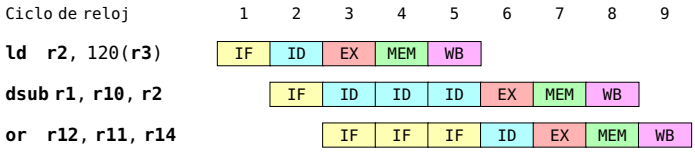
En este caso no es necesario disponer de una nueva ruta de reenvío desde la salida de la etapa WB a la entrada de la etapa EX, pues la instrucción **dsub** leerá el valor correcto del registro **r2** durante la segunda mitad del ciclo de su etapa ID, ya que este habrá sido previamente escrito en la primera mitad del ciclo en la etapa WB de la instrucción **dadd**. La figura 2.37 ilustra esta situación.

Las rutas de reenvío consideradas hasta el momento suponen que la instrucción que escribe en el registro es una instrucción aritmética, la instrucción **dadd r2, r3, r4**. Las rutas de reenvío mostradas son válidas también con otros tipos de instrucciones que escriben en registro, con la excepción de las instrucciones de carga. Si se trata de una instrucción de carga, el valor del registro destino está disponible como muy pronto al finalizar la etapa MEM, por lo que la única ruta de reenvío que puede activarse es Salida MEM → Entrada EX. Un ejemplo de activación de esta ruta de reenvío se muestra en la figura 2.38, lo que permite reducir la detención de dos ciclos de reloj a solo uno para el fragmento de código siguiente.

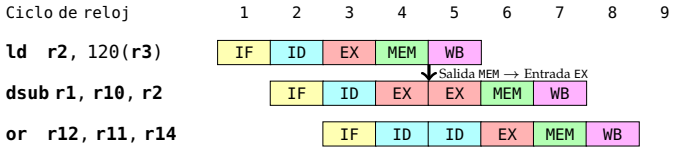
```
ld   r2,  120(r3)
dsub r1,  r10, r2
or   r12, r11, r14
```

Un matiz que puede pasar desapercibido a primera vista en la figura 2.38 es la detención en la etapa EX en lugar de la etapa ID por la dependencia de datos. La razón fundamental reside en la elevada complejidad que supondría a la unidad de control detectar cuándo debe detener la etapa ID en microarquitecturas que van más allá de la microarquitectura segmentada básica.

Cuando existen rutas de reenvío las detenciones por dependencias de datos se producen en la etapa EX en lugar de en la etapa ID. Esto resulta en una implementación mucho más simple. La etapa EX se detiene cuando no ha recibido alguno de

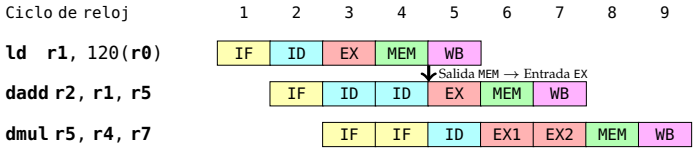


(a) Dependencia RAW con instrucción inicial de carga

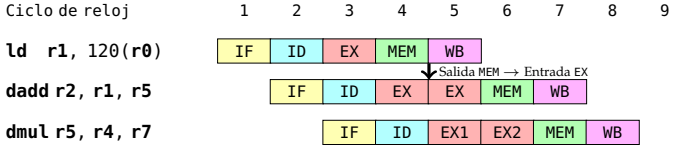


(b) Ruta de reenvío Salida `MEM` → Entrada `EX` con instrucción inicial de carga

Figura 2.38: Reducción de la detención RAW con instrucción inicial de carga



(a) Detención en la etapa `ID`



(b) Detención en la etapa `EX`

Figura 2.39: Mejora de rendimiento con detenciones en la etapa `EX`

los registros que necesita de su registro de segmentación de entrada o a través de las rutas de reenvío.

Una consecuencia adicional de llevar las detenciones por dependencias de datos a la etapa `EX` es la mejora de rendimiento cuando se tienen varias unidades de ejecución. La detención de una instrucción en su etapa `EX` no detiene la ejecución de instrucciones posteriores que utilizan otras unidades de ejecución, tal como se muestra en la figura 2.39, en la cual se asume que la instrucción **`dmul r5, r1, r7`** emplea una unidad de ejecución de dos ciclos segmentada.

Al no producirse la detención en la etapa `ID` de **`dmul r5, r1, r7`**, sino en su etapa `EX1`, la instrucción siguiente **`dadd r2, r4, r5`**, la cual utiliza otra unidad de ejecución, puede entrar en la etapa `ID` y a continuación ejecutarse, sin tener que esperar a que termine la detención de la instrucción `add` **`r2, r1, r5`**.

Hasta ahora se ha considerado el caso en el que se activa una única ruta de reenvío durante la ejecución de una instrucción, pero en general, una instrucción con

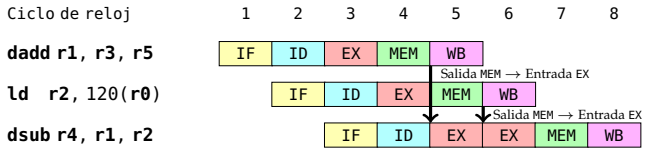
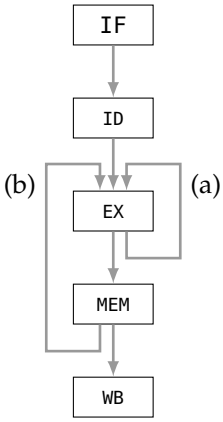


Figura 2.40: Activación de dos rutas de reenvío durante la ejecución de una instrucción



(a) Instrucciones de escritura sin carga con instrucción de lectura consecutiva

(b) Tres opciones:

- Instrucciones de escritura sin carga y lectura separadas por una instrucción
- Instrucciones de carga y lectura consecutivas**
- Instrucciones de carga y lectura separadas por una instrucción

** En este caso persiste una detención de 1 ciclo

Figura 2.41: Resumen de las rutas de reenvío

dos registros fuente puede requerir la activación de dos rutas de reenvío en el mismo instante, o incluso en instantes diferentes. Esta última situación se ejemplifica en la figura 2.40. La etapa `EX` no puede progresar en el ciclo 5, pues para hacerlo requiere el valor del registro `r2`, que no está disponible hasta el final del ciclo. Sin embargo, a pesar de detenerse, debe capturar el valor del registro `r1` a través de la ruta de reenvío Salida `MEM` → Entrada `EX` que se activa al principio del ciclo.

En resumen, implementando las rutas de reenvío Salida `EX`→Entrada `EX` y Salida `MEM`→Entrada `EX` se consiguen eliminar las detenciones por dependencias de datos RAW en casi todos los casos. La figura 2.41 muestra de forma esquemática las rutas de reenvío para una instrucción de escritura que no sea de carga y para una instrucción de carga.

Renombrado de registros

A diferencia de la planificación de instrucciones y las rutas de reenvío, el renombrado de registros no busca minimizar el efecto de las dependencias de datos, sino directamente eliminarlas. Concretamente, busca eliminar las dependencias no verdaderas, esto es, las dependencias de tipo WAR y WAW, que podían producir detenciones cuando la CPU ejecutaba instrucciones fuera de orden.

Las dependencias WAR y WAW no forman parte de la semántica del programa. Son creadas por el compilador al aprovechar un número de registros muy limita-

```
a = 1;          (1) ori  r5, r0, 1  ; r5 <- a
b = 3 * a;      (2) ori  r6, r0, 3  ; r6 <- 3
c = 7;          (3) dmul r4, r6, r5 ; r4 <- b
d = 6;          (4) ori  r4, r0, 7  ; r4 <- c
                (5) ori  r6, r0, 6  ; r6 <- d
```

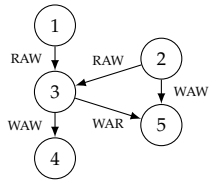


Figura 2.42: Ejemplo de dependencias producidas por el reciclaje de registros por parte del compilador

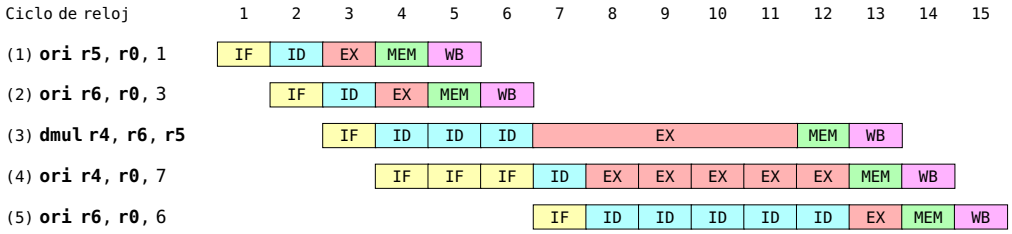


Figura 2.43: Ejecución con reciclaje de registros

do de la arquitectura a la hora de asociar almacenamiento a las variables[7]. Siempre que pueden, los compiladores emplean registros en lugar de posiciones de memoria, pues el acceso es mucho más rápido, lo que redunda en un menor tiempo de ejecución. Por esta razón, cuando el compilador emplea un registro para una variable y detecta que a partir de un cierto punto del programa esa variable ya no se usa, lleva a cabo el reciclaje del registro asociado, es decir, lo usa para almacenar otra variable del programa, lo que puede conducir a la aparición de dependencias WAR y WAW.

Por ejemplo, la figura 2.42 muestra un fragmento de un programa fuente junto a su traducción a ensamblador y el grafo de dependencias resultante. En el grafo se indica con un arco una dependencia, donde la instrucción apuntada depende de la instrucción de la que parte el arco. Además de las dependencias verdaderas (RAW), se pueden observar dependencias WAR y WAW, ocasionadas por el reciclaje de los registros `r4` y `r6`. El registro `r4` se emplea inicialmente para almacenar la variable `b` y a continuación se recicla para almacenar la variable `c`. El registro `r6` se emplea inicialmente para almacenar la constante 3 y más adelante se recicla para almacenar la variable `d`.

La ejecución del programa anterior por parte de una CPU MIPS64 sin excepciones precisas es la mostrada en la figura 2.43. Se ha supuesto que la unidad de ejecución de multiplicación de enteros requiere 5 ciclos de reloj. Entre otras, se observa una detención ocasionada por una dependencia WAW; la instrucción `ori r4, r0, 7` no puede terminar hasta que lo haga la instrucción `dmul r4, r6, r5` anterior.

Si el compilador no hubiese reciclado ningún registro, por ejemplo almacenando la variable `c` en el registro `r7` y la variable `d` en el registro `r8`, el programa y el grafo de

[7] Por ejemplo, cuando se aplican optimizaciones como el desenrollado de bucles que replican el cuerpo de los bucles para evitar saltos y por ende riesgos de control.

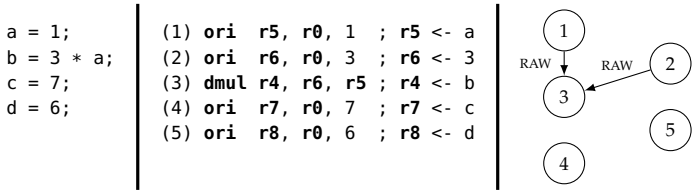


Figura 2.44: Eliminación de dependencias WAR y WAW evitando el reciclaje de registros

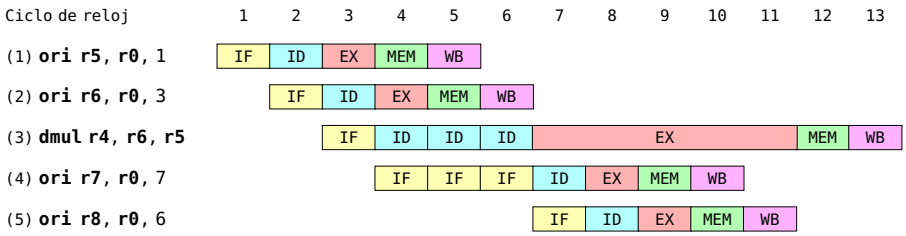


Figura 2.45: Ejecución sin reciclaje de registros

dependencias pasarían a ser los mostrados en la figura 2.44, lo que redundaría en un menor número de dependencias entre las instrucciones. Al eliminar el reciclaje de registros y desaparecer la dependencia WAW entre las instrucciones `dmul r4, r6, r5` y `ori r4, r0, 7`, desaparece la detención asociada, pasando la ejecución de 14 a 13 ciclos de reloj, tal como se muestra en la figura 2.45.

En cualquier caso, el reciclaje de registros es inevitable, pues las CPU disponen habitualmente de un conjunto de registros reducido. A priori, una solución al problema sería incrementar el número de registros de la CPU para evitar el reciclaje de registros. Sin embargo esta solución no es viable, pues un incremento en el número de registros significa un cambio importante en la arquitectura del juego de instrucciones que puede acarrear problemas de compatibilidad del software. Además, incluso en el caso de nuevas arquitecturas que no están condicionadas por mantener compatibilidades, disponer de un elevado número de registros es desaconsejable, pues la identificación de los mismos en las instrucciones requiere más bits y obligaría a códigos de instrucción más largos. Por ejemplo, en la arquitectura MIPS64 pasar de 32 registros a 256 registros supone pasar de 5 a 8 bits para identificar un registro. Si fuese necesario identificar 3 registros, tal como ocurre en muchas instrucciones, harían falta 24 bits solo para el direccionamiento y obligaría a pasar el código de las instrucciones de 32 a 64 bits.

La solución habitual al problema del reciclaje de registros consiste en la aplicación de la técnica de renombrado de registros a nivel hardware. La idea es que los registros de la arquitectura, de `r0` a `r31` en el caso de MIPS64, son registros simbólicos que se asocian a registros físicos. De esta forma la arquitectura no cambia, pues los registros que ve el programador son los registros arquitectónicos, pero se puede sacar provecho de un conjunto de registros físicos más amplio. A modo de ejemplo,

Registro arquitectónico	Registro físico	Contador
r0	rr0	0
.....		
r31	rr31	0
-	rr32	0
.....		
-	rr63	0
f0	rf0	0
.....		
f31	rf31	0
-	rf32	0
.....		
-	rf63	0

Tabla 2.4: Estado de la tabla de renombrado al comienzo de la ejecución

se supondrá que la CPU MIPS64 dispone de 64 registros físicos para enteros que van de `rr0` a `rr63` y otros 64 de punto flotante que van de `rf0` a `rf63`.

La CPU dispone de una estructura de datos asociada al renombrado de registros, a la que se denomina tabla de renombrado, que evoluciona durante la ejecución de las instrucciones. Puede verse el estado inicial de la tabla de renombrado para una microarquitectura MIPS64 en la tabla 2.4.

La tabla relaciona cada registro arquitectónico con un registro físico. Además, para cada registro físico hay un campo `Contador` que indica el número de instrucciones pendientes de terminar que emplean el registro físico como operando fuente. Los registros físicos pueden encontrarse en tres estados:

- Asociado y en uso. En este caso el registro físico es utilizado como operando fuente en al menos una instrucción que no ha terminado. Este estado se distingue porque el campo `Contador` del registro físico es mayor que cero. Además, en este estado el registro físico está asociado a un registro arquitectónico, por lo que no está disponible para futuros renombrados.
- Asociado y en desuso. En este caso el registro físico está asociado a un registro arquitectónico y no es operando fuente de instrucciones pendientes de terminar. Este estado se distingue porque el campo `Contador` del registro físico es cero. Al estar asociado a un registro arquitectónico no está disponible para futuros renombrados. En este estado se encuentran inicialmente los registros físicos `rr0` a `rr31` y `rf0` a `rf31`.
- Disponible. El registro físico está disponible para el renombrado de registros arquitectónicos en próximas instrucciones. Se distingue porque su campo `Contador` está a cero y no está asociado a ningún registro arquitectónico. En

este estado se encuentran inicialmente los registros físicos rr32 a rr63 y rf32 a rf63.

A partir de la tabla anterior el funcionamiento del renombrado de registros sería el siguiente:

1. Al arrancar la CPU los registros arquitectónicos **r0** a **r31** y **f0** a **f31** están asociados a los registros físicos del mismo índice, que están en desuso, ya que tendrán su campo Contador a 0. Por ejemplo, inicialmente **r6** está asociado a rr6 y no hay ninguna instrucción que lea rr6 pendiente de terminar. Los demás registros físicos están disponibles (rr32 a rr63 y rf32 a rf63), por lo que su campo Contador está a cero y no están asociados a ningún registro arquitectónico. Esta situación es la mostrada en la tabla 2.4.

2. Los registros físicos disponibles se organizan en dos colas FIFO, la primera de ellas inicialmente contiene los registros rr32 a rr63, mientras que la segunda contiene los registros rf32 a rf63.

3. En la segunda mitad de la etapa ID se identifican los registros arquitectónicos fuente y destino durante la decodificación de la instrucción.

   Los registros arquitectónicos fuente se sustituyen por los registros físicos asociados, obtenidos a partir de la tabla de renombrado. Además, se incrementa su campo Contador para reflejar que una nueva instrucción los referencia. Si aparece el mismo registro físico varias veces como operando fuente en la instrucción sólo se incrementará en una unidad su Contador, pues este hace referencia al número de instrucciones pendientes de terminar que lo leen.

   El registro arquitectónico destino se renombra asociándolo a un nuevo registro físico que se extrae de la cola FIFO correspondiente. Al mismo tiempo debe borrarse su asociación con el registro físico anterior, pues un registro arquitectónico debe estar obligatoriamente asociado a un único registro físico. Es importante tener en cuenta que el registro **r0** nunca se renombra, ya que no puede actuar como operando destino en las instrucciones. Está programado por hardware con un valor 0.

4. Cuando termina la ejecución de una instrucción, concretamente en la primera mitad de su etapa WB, se decrementa el campo Contador de los registros físicos fuente de la instrucción. Si el registro físico aparece más de una vez, sólo se decrementa en una unidad. Si el campo Contador del registro físico pasa a tomar el valor cero y no tiene asociado un registro arquitectónico, este registro físico quedará disponible para ser usado en un futuro renombrado y se añade a la cola FIFO correspondiente.

Para ejemplificar el funcionamiento del renombrado de registros, la figura 2.46 ilustra el renombrado de registros sobre el mismo programa de la figura 2.42. Debe tenerse en cuenta que en un ejemplo tan sencillo sería fácil evitar el reciclaje de registros en tiempo de compilación, tal como se ilustró en la figura 2.44. Sin embargo, no siempre es posible realizar el renombrado de esta forma, ya que el número de registros disponibles resulta muy limitado para los programas actuales y algunas

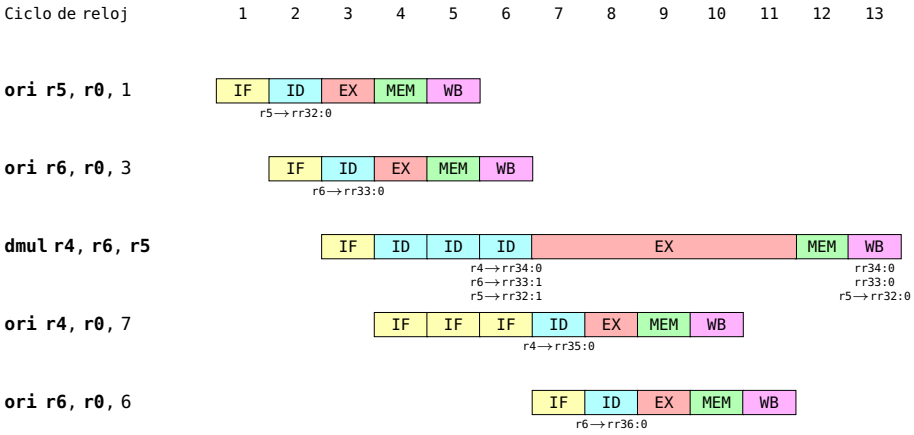


Figura 2.46: Ejecución con renombrado de registros a nivel hardware

dependencias se producen en función de saltos condicionales previos, que pueden ser imposibles de calcular en tiempo de compilación.

Para describir el funcionamiento del renombrado se emplean expresiones del tipo `ri→rrj:k` para indicar cambios en la tabla de renombrado. La expresión anterior indica que el registro arquitectónico `ri` está asociado al registro físico `rrj` y que este último tiene un valor de contador igual a `k`. Cuando un registro físico está disponible para futuros renombrados no aparece el registro arquitectónico en la expresión y el contador de uso es cero, indicándose con la expresión `rrj:0`.

Para analizar el renombrado de registros en el ejemplo es necesario tener en cuenta las etapas `ID` y `WB` de las instrucciones siguiendo el orden temporal marcado por los ciclos de reloj (no el de las instrucciones), partiendo del estado inicial indicado en la tabla 2.4.

- Ciclo de reloj 2. Etapa `ID` de la instrucción **`ori r5, r0, 1`**. El registro arquitectónico destino **`r5`** debe ser renombrado, por lo que dejará de estar asociado al registro físico `rr5` y pasará a estar asociado al primer registro físico disponible en la cola FIFO. De esta forma, se actualiza la tabla de renombrado con `r5→rr32:0`. A partir de este ciclo, cualquier referencia al registro fuente **`r5`** se convierte en una referencia al registro físico asociado `rr32`.
- Ciclo de reloj 3. Ocurre algo similar con el registro **`r6`** durante la etapa `ID` de la instrucción **`ori r6, r0, 3`**, por lo que la tabla de renombrado se actualiza con `r6→rr33:0`.
- Ciclo de reloj 4. Durante la etapa `ID` de la instrucción **`dmul r4, r6, r5`** habría que renombrar el registro destino e incrementar los contadores de los registros físicos asociados a los registros arquitectónicos fuente. Sin embargo, dado que se identifica una dependencia RAW sobre el registro **`r6`** que conduce a una detención del cauce de ejecución, las modificaciones en la tabla de renombrado se retrasan hasta el ciclo de reloj 6, una vez el registro físico `rr33` es escrito por la instrucción **`ori r6, r0, 3`**.

- Ciclo de reloj 5. Termina la instrucción **ori r5**, **r0**, 1, por lo que habría que decrementar en una unidad los contadores de sus registros físicos fuente. Dado que en este caso no hay registros físicos en uso (no se considera el registro **r0**), no se modifica la tabla de renombrado.

- Ciclo de reloj 6. Termina la instrucción **ori r6**, **r0**, 3, que como la instrucción **ori r5**, **r0**, 1 tampoco usa registros fuente.

  En este ciclo también termina la detención por la dependencia RAW, por lo que ahora sí se modifica la tabla de renombrado durante la etapa ID de la instrucción **dmul r4**, **r6**, **r5**. En primer lugar, se renombra el registro destino, por lo que **r4** pasa a estar asociado a rr34 (r4→rr34:0). En segundo lugar, deben incrementarse los contadores de los registros físicos asociados a **r6** y **r5**, esto es, rr33 y rr32 respectivamente (r6→rr33:1 y r5→rr32:1).

- Ciclo de reloj 7. Durante la etapa ID de la instrucción **ori r4**, **r0**, 7 se renombra el registro **r4**. Se actualiza la tabla de renombrado con r4→rr35:0.

- Ciclo de reloj 8. En la etapa ID de la instrucción **ori r6**, **r0**, 6 se renombra el registro **r6** (r6→rr36:0).

- Ciclo de reloj 13. Termina la instrucción **dmul r4**, **r6**, **r5**, por lo que se decrementa en una unidad el contador de los registros físicos que utiliza como fuente, esto es, los registros rr33 y rr32. Los registros físicos rr34 y rr33 ya no están asociados a ningún registro arquitectónico y pasan a tener un contador de cero (no hay ninguna instrucción pendiente que los lea), por lo que pasan a estar disponibles y se añaden a la cola FIFO. El contador del registro rr32 también pasa a tomar el valor cero, pero a diferencia de los dos anteriores está asociado a un registro arquitectónico, el registro **r5**, por lo que no está disponible.

En definitiva, el código ejecutado realmente sobre la microarquitectura es el mostrado a continuación:

```
ori  rr32, rr0,  1
ori  rr33, rr0,  3
dmul rr34, rr33, rr32
ori  rr35, rr0,  7
ori  rr36, rr0,  6
```

Puede verificarse que en este código solo existen dependencias de datos verdaderas (RAW), ya que las dependencias WAR y WAW se han eliminado gracias al renombrado de registros. Esto ha posibilitado reducir el tiempo de ejecución de 14 ciclos de reloj (figura 2.43) a 13 ciclos de reloj (figura 2.46). No parece mucho, pero debe tenerse en cuenta el pequeño número de instrucciones del ejemplo; se ha ganado un ciclo en 5 instrucciones.

Sin embargo, aunque el número de registros físicos suele ser elevado, puede haber programas complejos para los que resulte imposible eliminar todas las dependencias WAR o WAW por el alto número de renombrados necesarios.

2.3.6. Reducción de detenciones por riesgos de control

La ejecución de las instrucciones de salto condicional requiere, por un lado, calcular la dirección destino del salto, y por otro evaluar la condición. El nuevo valor del contador de programa se establece en la etapa `MEM` si se cumple la condición. En cambio, en el caso de los saltos incondicionales, el nuevo valor del `PC` se establece en la etapa `ID`.

Cuando se decodifica una instrucción de salto condicional en la etapa `ID`, se impide que las instrucciones que la siguen entren en el cauce de ejecución hasta que esta abandone la etapa `MEM`. No obstante, no se puede impedir que la instrucción siguiente a la de salto entre en la etapa `IF`, pues en ese momento aún no se ha identificado la instrucción de salto, por lo que posteriormente el resultado de esa etapa es desechado. En total, se produce una detención de tres ciclos de reloj. Esta detención se muestra en el ejemplo de la figura 2.23.

A continuación, se estudian algunas técnicas para la reducción de las detenciones debidas a riesgos de control.

Evaluación agresiva de saltos

La *evaluación agresiva de saltos* es una técnica para reducir la penalización en los saltos condicionales que consiste en adelantar la evaluación de los saltos (cálculo de la dirección y evaluación de la condición) en el cauce segmentado, por ejemplo hasta la etapa `ID`, lo que resultaría en una detención de un ciclo de reloj, igual que para los saltos incondicionales.

Para ello, es necesario introducir algunos cambios en el camino de datos. En primer lugar, debe moverse la lógica para el cálculo de la dirección destino del salto desde la etapa `EX` a la etapa `ID`. En segundo lugar, debe añadirse un comparador para verificar si la condición del salto se cumple. Con estos cambios, la parte afectada del camino de datos para dar soporte a la instrucción **`beq`** quedaría según se muestra en la figura 2.47. La figura 2.48 muestra un ejemplo de ejecución con evaluación agresiva de saltos.

Una cuestión importante a tener en cuenta es que en la segunda mitad del ciclo de reloj se leen los registros en la etapa `ID` (se escriben en la primera mitad en `WB`). Con la implementación mostrada en la figura 2.47, la etapa `ID` debe realizar más trabajo durante la segunda mitad del ciclo. Deben compararse los registros leídos, por lo que podría implicar una mayor duración del ciclo de reloj de la CPU para acomodar esta nueva lógica. Por esta razón, esta técnica es difícil de implementar de forma eficiente, ya que suele implicar una mayor duración del ciclo de reloj. La clave está en acomodar la lógica de evaluación de los saltos lo más pronto posible en el cauce sin aumentar la duración del ciclo de reloj. Debe recordarse que la duración del ciclo de reloj coincide con la duración de la etapa más larga.

Otro problema que lleva aparejado la evaluación agresiva de saltos es que es necesaria una nueva ruta de reenvío. La instrucción de salto condicional necesita los valores de los registros en la etapa `ID`, y estos pueden ser generados en las etapas `EX` y `MEM` como resultado de operaciones aritmético-lógicas y de carga de memoria respectivamente. Para el segundo caso, el registro se escribiría en la etapa `WB` durante

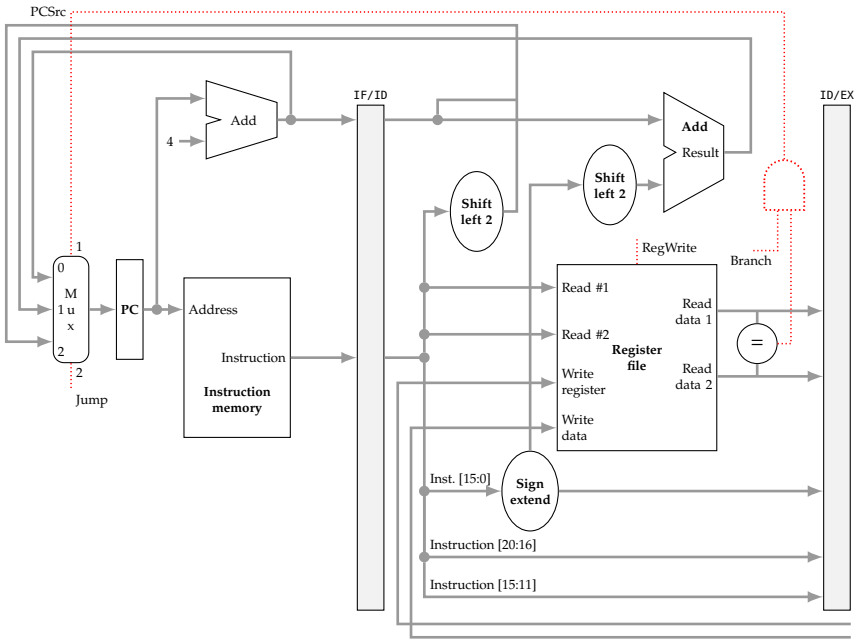


Figura 2.47: Etapas `IF` e `ID` para la evaluación agresiva del salto **`beq`**

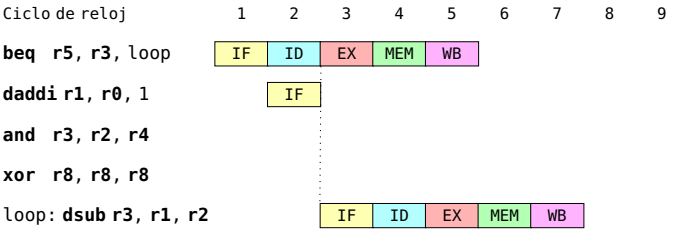


Figura 2.48: Detención por riesgo de control con evaluación agresiva de saltos suponiendo que el salto se toma

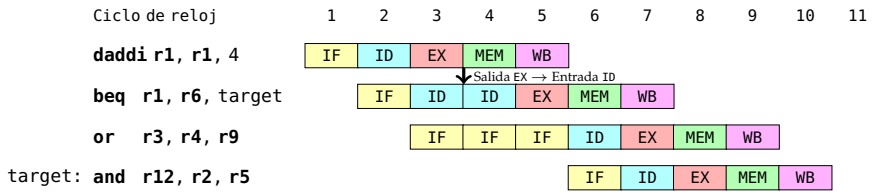


Figura 2.49: Nueva ruta de reenvío Salida `EX` → Entrada `ID` con evaluación agresiva de saltos. El salto no se toma.

la primera parte del ciclo siguiente y se podría leer en la segunda parte del ciclo. Sin embargo, para el primer caso es necesaria una nueva ruta de reenvío: Salida `EX` → Entrada `ID`. La figura 2.49 muestra un ejemplo de utilización de esta nueva ruta de reenvío. Nótese cómo a pesar de utilizar la ruta de reenvío es necesario detener el *pipeline* un ciclo.

Aunque la técnica de evaluación agresiva de saltos es aplicable a microarquitecturas sencillas con pocas etapas, como es el caso de la microarquitectura MIPS64 en estudio, es poco efectiva en el caso de microarquitecturas con mayor número de etapas.

Predicción de saltos

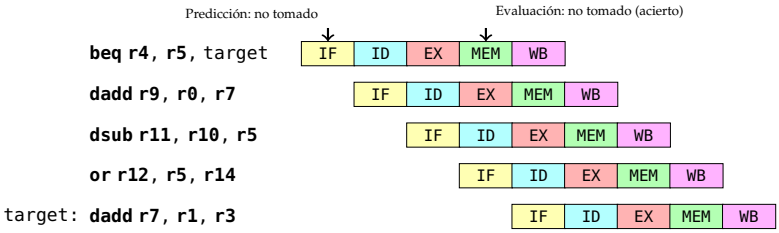
La predicción de saltos es la técnica de reducción de detenciones por riesgos de control más empleada. Consiste en predecir si el salto se cumple o no, de tal forma que al final de la etapa `IF` el contador de programa contenga la dirección de memoria de la instrucción que previsiblemente se debería ejecutar a continuación.

Se trata de una técnica de ejecución especulativa, es decir, trata de predecir el resultado del salto y en base a esa predicción ejecuta instrucciones que de antemano no sería posible ejecutar hasta más adelante cuando se evaluase la condición del salto y su dirección de destino. Si la predicción resulta correcta se habrá mejorado el rendimiento, pues en el mejor de los casos, no habrá sido necesario detener el cauce esperando por el resultado del salto. Por el contrario, si la predicción ha sido incorrecta, se han ejecutado instrucciones de forma indebida, por lo que sus efectos deben revertirse.

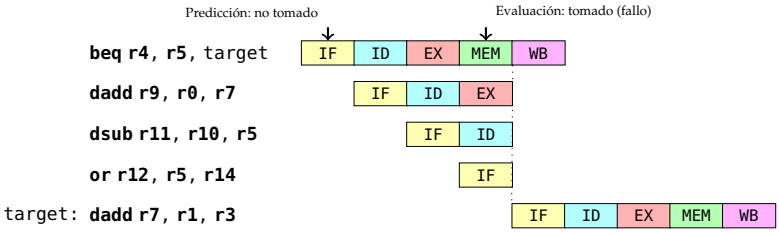
Para ilustrar la predicción de saltos se empleará este fragmento de código.

```
  beq  r4,  r5,  target
  dadd r9,  r0,  r7
  dsub r11, r10, r5
  or   r12, r5,  r14
target:
  dadd r7,  r1,  r3
```

La técnica más básica de predicción de saltos se denomina «siempre no tomado» o *always not taken* y consiste en suponer que las condiciones de salto nunca se cumplen. Una vez se llega a una instrucción de salto condicional, el hardware de control de la CPU supone siempre que la condición de salto no se cumple y sigue ejecutando



(a) Acierto



(b) Fallo

Figura 2.50: Comportamiento del predictor «siempre no tomado»

las instrucciones siguientes a la del salto sin detenerse, lo que además simplifica la gestión de los saltos. Una vez la instrucción de salto condicional llega a la etapa `MEM` (o la etapa `ID` si se adopta una evaluación agresiva de saltos) y se evalúa la condición del salto caben dos posibilidades[8]:

- La condición de salto no se cumple. En este caso la predicción fue acertada y la ejecución de las instrucciones siguientes a la del salto es correcta. No se ha detenido el cauce, por lo que la ganancia es importante; se pasa de una detención de 3 ciclos a no detener el cauce. Esta situación se muestra en la figura 2.50a.
- La condición de salto se cumple. En este caso la predicción fue fallida. Las instrucciones **`dadd r9, r0, r7`**, **`dsub r11, r10, r5`** y **`or r12, r5, r14`** deben dejar de progresar en el cauce, por lo que son desechadas (se vacía el cauce). En este caso la detención sigue siendo de tres ciclos como en el caso del cauce de ejecución original. Esta situación se muestra en la figura 2.50b.

Por ejemplo, si se cumple la condición en la mitad de los saltos condicionales de un programa, la detención media pasa de tres ciclos de reloj a justo la mitad, con la consiguiente reducción del CPI.

Una característica interesante de la microarquitectura MIPS64 en estudio es que las etapas `IF`, `ID` y `EX` no modifican ni el estado de los registros ni el estado de la memoria, por lo que en caso de fallo de predicción en el salto no es necesario revertir

[8] Es necesario recordar que la evaluación de la condición se realiza en la etapa `EX`, pero la modificación del registro `PC` se realiza en la etapa `MEM`, por lo que en la práctica los saltos se toman o no en esta última etapa.

escrituras, que es la parte más complicada. Recuérdese que las escrituras en memoria y registros se llevan a cabo más adelante en las etapas `MEM` y `WB`, respectivamente. De nuevo esta característica tan favorable no es fruto de la casualidad sino de un diseño muy cuidadoso de la microarquitectura. En cauces con un alto número de etapas las instrucciones que se ejecutan especulativamente se etiquetan como tales y no se les permite realizar escrituras definitivas en registros o en memoria. Solo cuando se resuelve el salto se realizan las escrituras si la predicción fue acertada o se descartan si fue errónea.

La técnica de predicción de saltos «siempre no tomado» es muy básica y se emplea poco actualmente. Estudios experimentales han demostrado que, en promedio, de tres saltos condicionales que se ejecutan, dos saltos se toman y uno no, por lo que la eficacia de este heurístico sería del orden del 33 %. Por supuesto, estos datos son sólo orientativos.

Las CPU segmentadas modernas emplean muchas más etapas que la microarquitectura segmentada de MIPS64 vista con el objetivo de mejorar el rendimiento. A medida que aumenta el número de etapas, las consecuencias de una predicción errónea son mayores. Implican más ciclos de detención por las instrucciones que hay que descartar y mayor consumo de energía por la ejecución de estas instrucciones innecesarias. Por esta razón, se emplean técnicas de predicción de saltos sofisticadas que van mucho más allá de suponer que la condición de salto nunca se cumple.

La predicción de saltos es un tema de gran importancia en el diseño de las CPU por la influencia que tiene sobre el rendimiento. Ha habido y sigue habiendo mucho trabajo e investigación sobre este tema. Los algoritmos de predicción de saltos de los procesadores actuales son muy complejos, por lo que quedan fuera de los objetivos de este texto. No obstante, resulta interesante mostrar algún algoritmo de predicción de saltos mucho más eficaz que el simple «siempre no tomado».

En primer lugar, conviene meditar un poco acerca de la naturaleza del algoritmo de predicción «siempre no tomado». Pertenece a la familia de algoritmos denominados estáticos, pues una vez llegan a una instrucción de salto su comportamiento es siempre el mismo para esa instrucción, ya sea para saltar o no saltar. El mayor inconveniente de los algoritmos de predicción estáticos es que no tienen en cuenta la historia de la instrucción de salto, es decir, si en instantes anteriores en que fue ejecutada se produjo el salto o no. Por ejemplo, considérese este bucle de 100 repeticiones:

```
  .....
  ori   r1, r0, 100 ; r1 = 100
startf:
  daddi r1, r1, -1  ; r1 = r1 - 1
  bnez  r1, startf  ; jump to startf if r1 not zero
endf:
  .....
```

En las 99 primeras ejecuciones de la instrucción de salto **`bnez`** se cumple la condición, pero en la ejecución 100 no se cumple y termina el bucle. Si se tuviese en cuenta la historia del salto, se podría fallar en la predicción de la primera ejecución o incluso la segunda, pero más adelante, una vez se analiza la historia del salto, está claro que la mejor predicción es suponer que se cumplirá la condición. En cambio, en otro bucle con distinta estructura, podría darse la situación contraria y que lo mejor analizando la historia del salto fuese suponer que el salto no se cumple. Esta es la idea

clave que incorporan los predictores dinámicos: tienen en cuenta la historia del salto para predecir si se va a cumplir o no la condición del salto en la ejecución actual.

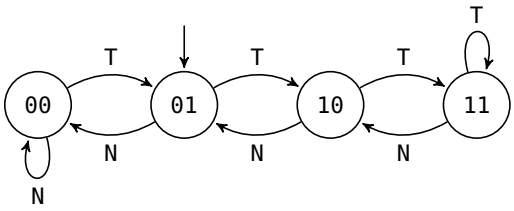
Uno de estos predictores dinámicos es el predictor de saltos de dos bits, que utiliza una tabla, denominada BHT (*Branch History Table*) o BPB (*Branch Prediction Buffer*), donde se almacena un histórico con el comportamiento pasado de los saltos condicionales. También es necesario disponer de una segunda tabla, denominada BTB (*Branch Target Buffer*) donde se almacenan las direcciones de salto tanto de los saltos condicionales como de los incondicionales, de forma que pueda ejecutarse de forma especulativa la rama «salto tomado» en caso de que la predicción así lo determine.

Por simplicidad, de ahora en adelante se asume que la información contenida en el BTB se incluye en la BHT, de forma que solo es necesaria una única tabla, la BHT, para el funcionamiento del predictor. Cada vez que se ejecuta una instrucción de salto por primera vez, se asocia al salto una entrada en la tabla BHT que contiene:

- La dirección de memoria de la instrucción de salto (o sus bits menos significativos). Coincide con el valor del `PC` al comienzo de la etapa `IF` de la instrucción de salto. Este campo se emplea para obtener la entrada en la BHT asociada al salto durante la etapa `IF` a la vez que se lee el código de la instrucción de la memoria de instrucciones. Por lo tanto, la lectura de la BHT no acarrea ninguna penalización en el rendimiento, pues se hace en paralelo con la lectura de la memoria de instrucciones.
- Dirección destino del salto. En la microarquitectura MIPS64 vista, la dirección destino del salto condicional se calcula en la etapa `EX` (sin evaluación agresiva), pero no se hace efectiva hasta la etapa `MEM`, que es en la que se actualiza el `PC`. Es en esta última etapa donde se escribe la dirección de destino en la BHT la primera vez que se ejecuta la instrucción de salto. En el caso de los saltos incondicionales, o los saltos condicionales con evaluación agresiva, la escritura se realiza en la etapa `ID`.
- Bits de predicción. Se utilizan para predecir si el salto se toma o no en base al comportamiento pasado del salto.

El predictor de saltos de dos bits almacena en cada entrada de la BHT dos bits asociados a cada salto. Estos dos bits pueden considerarse un contador que se incrementa cada vez que se salta y se decrementa cuando no se salta, saturando por debajo a `00` y por encima a `11`. En base al valor de este contador son posibles estos cuatro estados:

- `00` (*strong not taken*).
- `01` (*weak not taken*).
- `10` (*weak taken*).
- `11` (*strong taken*).



BHT

Dirección	Destino del salto	Historial
1010101...	0011101...	01
0111010...	1100010...	10
1010101...	0111011...	11

Figura 2.51: Máquina de estados y memoria de saltos de un predictor de 2 bits

El bit más significativo de los dos que utiliza el predictor determina la predicción sobre el salto. Si el bit vale `0` (estados *strong not taken* y *weak not taken*) la predicción es que el salto no se toma. Si, por el contrario, el bit vale `1` (estados *strong taken* y *weak taken*) la predicción es que el salto se toma.

La parte izquierda de la figura 2.51 muestra el diagrama de estados correspondiente al predictor, en el cual se puede apreciar que el estado inicial es el `01` (*weak not taken*). La parte derecha de la figura muestra la BHT.

Para implementar el predictor de 2 bits son necesarios varios cambios en la microarquitectura. En cada ciclo de reloj la etapa `IF` comprueba si hay una entrada en la BHT con la dirección actual del `PC`. Si la hay, se dice que se produce un acierto de BHT, mientras que si no la hay se dice que se produce un fallo de BHT.

Si se produce un fallo de BHT y se trata de un salto incondicional se incurrirá en una detención de un ciclo, pues no se conoce la dirección destino del salto hasta que termina la etapa `ID`. Al final de la etapa `ID` se actualiza la BHT para almacenar la dirección destino del salto.

Si el fallo de BHT ocurre con una instrucción de salto condicional se asume que el salto no se toma. Si la predicción es correcta no hay detención, pero si es incorrecta habrá que vaciar el cauce y se incurre en una detención de uno o tres ciclos de reloj, dependiendo de si se usa o no una evaluación agresiva de saltos. En cualquiera de los casos anteriores, se actualiza la tabla BHT, ya sea en la etapa `MEM` o en la etapa `ID`.

Si se produce un acierto de BHT y se trata de un salto incondicional, la dirección que se escribe en el `PC` es la que figura como dirección de salto en la entrada de la BHT. Puesto que tanto la comprobación de la BHT como la escritura del `PC` se producen en la etapa `IF`, no se incurre en detención.

Si el acierto de BHT ocurre con una instrucción de salto condicional, el bit más significativo de los bits de predicción determina la rama a tomar. Si se predice que el salto es tomado, la dirección que se escribe en el `PC` es la almacenada en la entrada de la BHT. Si se predice que el salto no es tomado, se escribe en el `PC` el valor `PC+4`. Todo este proceso se realiza en la etapa `IF`. Posteriormente, en la etapa `ID` o `MEM`, dependiendo de si se emplea o no evaluación agresiva de saltos, se comprueba si la predicción fue correcta. Si la predicción fue correcta se continúa sin detención. Por el contrario, si fue incorrecta, es necesario desechar las instrucciones que han entrado erróneamente en el cauce y se incurrirá en una detención de uno o tres ciclos de reloj,

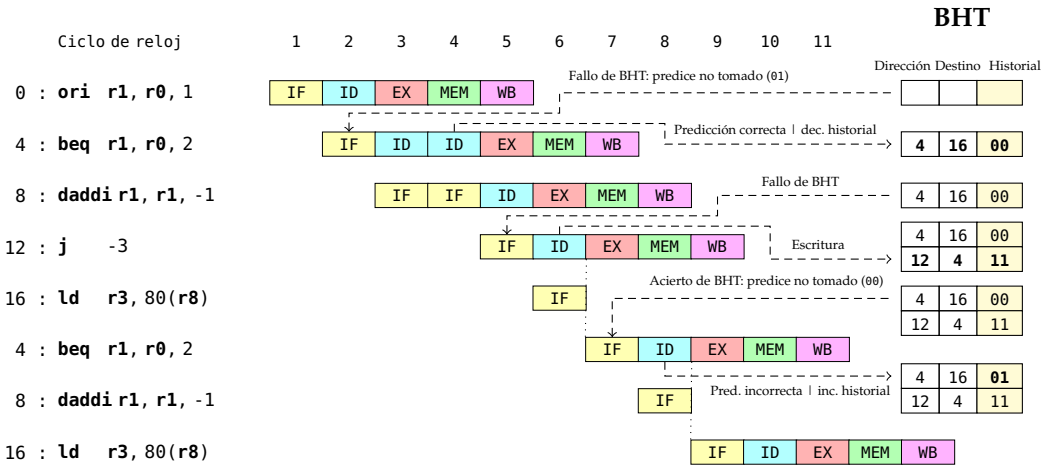


Figura 2.52: Funcionamiento del predictor de saltos de 2 bits

dependiendo de si se usa o no una evaluación agresiva de saltos. En cualquiera de los casos anteriores se actualiza la BHT, ya sea en la etapa `ID` o en la etapa `MEM`.

El funcionamiento del predictor de 2 bits se ilustra con el siguiente ejemplo, cuyo cronograma se muestra en la figura 2.52. En el ejemplo se asume que se utilizan una evaluación agresiva de saltos en la etapa `ID` y rutas de reenvío.

```
  ori   r1, r0, 1 ; r1 = 1
startf:
  beq   r1, r0, endf
  daddi r1, r1, -1
  j     startf
endf:
  ld    r3, 80(r8)
```

Por simplicidad, en la figura 2.52 solo se muestran las consultas BHT en las instrucciones de salto, si bien se realizarían en todas[9], ya que en la etapa `IF` todavía no se ha decodificado la instrucción.

- Como se aprecia en la figura, la primera vez que se ejecuta la instrucción **`beq`** la tabla está vacía, por lo que se produce un fallo de BHT. En ese caso, se asume una predicción «salto no tomado» y el `PC` se carga con la dirección de la siguiente instrucción (ubicada a partir de la dirección 8), que entra en el cauce en el ciclo 3. En el ciclo 4, una vez resuelta la dependencia RAW sobre **`r1`**, asumiendo la evaluación agresiva del salto, se evalúa y se comprueba que la predicción era correcta, esto es, el salto no se toma. Además, se escribe en la tabla BHT la entrada correspondiente al salto. El histórico del salto tomará el valor `00` (*strong not taken*) al no tomarse el salto (se decrementa el contador).
- En el salto incondicional también se produce un fallo de BHT, por lo que hay que procesar el salto como es habitual en la etapa `ID`. Una vez calculada la

[9] Las instrucciones que no sean saltos generarán siempre fallo de BHT.

dirección de salto, se actualiza el PC y se escribe en la entrada correspondiente de la tabla BHT la dirección de la instrucción de salto, la dirección de destino y 11 (*strong taken*) para el historial, de forma que la predicción sea saltar en el salto incondicional[10].

- En la segunda iteración se llega de nuevo a la instrucción de salto condicional, pero esta vez se produce un acierto de BHT. Como el estado almacenado es *strong not taken*, se predice que el salto no se toma. Se comprueba en el siguiente ciclo que la predicción fue incorrecta, por lo que se elimina la instrucción que ha entrado en la etapa IF incorrectamente (`daddi r1, r1, -1`) y se copia en el PC la dirección de salto. Se incurre en una detención de un ciclo de reloj. También se incrementa el contador del historial de la entrada en la tabla BHT.

Estudios experimentales empleando programas típicos han mostrado que un predictor de dos bits como el anterior logra predecir correctamente el 90 % de los saltos condicionales aproximadamente. Sin embargo, las CPU segmentadas actuales requieren unos porcentajes aún mayores que los logrados con ese predictor para no penalizar en exceso el CPI. El comportamiento de muchos saltos condicionales depende de otros saltos diferentes que le precedieron en la ejecución, por lo que los predictores más modernos tienen en cuenta no sólo la instrucción de salto objeto de la predicción sino otras instrucciones de salto.

2.3.7. Profundidad de la segmentación

La segmentación de instrucciones es una técnica que explota el paralelismo que existe entre las instrucciones para ejecutar varias al mismo tiempo, tantas como etapas del cauce segmentado. Es por esto que la independencia de las instrucciones de un programa es clave para lograr un alto grado de paralelismo y un buen rendimiento en las CPU segmentadas como consecuencia de ello.

Una posible vía para aumentar el rendimiento es aumentando la profundidad de la segmentación, esto es, incrementando el número de etapas del cauce. Se consigue así reducir el periodo de reloj y por lo tanto el tiempo de CPU. Por ejemplo, si se multiplica por dos el número de etapas, idealmente las etapas durarán la mitad, duplicando la frecuencia y reduciendo a la mitad el tiempo de ciclo, suponiendo como es habitual que todas las etapas tienen la misma duración. De acuerdo a la ley de hierro, el tiempo de CPU se reduciría a la mitad. A priori, cuanto mayor sea el número de etapas, tanto más se reducirá el tiempo de CPU y tanto mayor será el rendimiento de la CPU. Sin embargo, a medida que aumenta el número de etapas aparecen efectos negativos que tienden a disminuir la mejora en el rendimiento:

- Se produce un incremento del tiempo de ejecución de una instrucción ocasionado por un hardware de CPU más complejo. Entre las etapas del cauce se disponen registros de segmentación para enlazar unas con otras. Estos registros introducen un retardo debido al tiempo necesario para ser escritos. Por

[10] No tiene sentido predecir el resultado de saltos incondicionales, pues siempre se toman. Esto es consecuencia de combinar las tablas BHT y BTB en una sola. En los saltos incondicionales solo se utiliza la tabla BTB.

otra parte, al incrementar el número de etapas hay más etapas que requieren acceso simultáneo al fichero de registros y a la memoria, lo que implica el uso de ficheros de registros y memorias multipuerto. Los retardos de acceso, así como el coste de los ficheros de registros y memorias multipuerto, se incrementan rápidamente con el número de puertos.

- La penalización asociada a las detenciones de control ocasionadas por los saltos se incrementa rápidamente a medida que aumenta el número de etapas. En la microarquitectura MIPS64 segmentada vista, los saltos condicionales se evalúan en la etapa `MEM` (sin tener en cuenta su evaluación «agresiva»), por lo que el coste de una detención de control alcanza los tres ciclos de reloj en este caso: la distancia entre las etapas `IF` y `MEM`. Si se incrementa el número de etapas del cauce esta distancia aumentará y con ella la penalización en ciclos de reloj.
- Al aumentar el número de etapas aumenta la frecuencia de reloj, pues todas las etapas requieren un ciclo de la señal de reloj y estas son más cortas. Un pequeño aumento de la frecuencia de reloj trae consigo un gran aumento de la potencia disipada y, en la actualidad, existen unos límites sobre la potencia que puede disipar una CPU marcados por lo que se conoce como muro de potencia o energía (*power wall*). El apartado 2.5 proporciona más detalles sobre estos límites.

Teniendo en cuenta todas estas limitaciones, las CPU de propósito general actuales emplean entre 10 y 20 etapas.

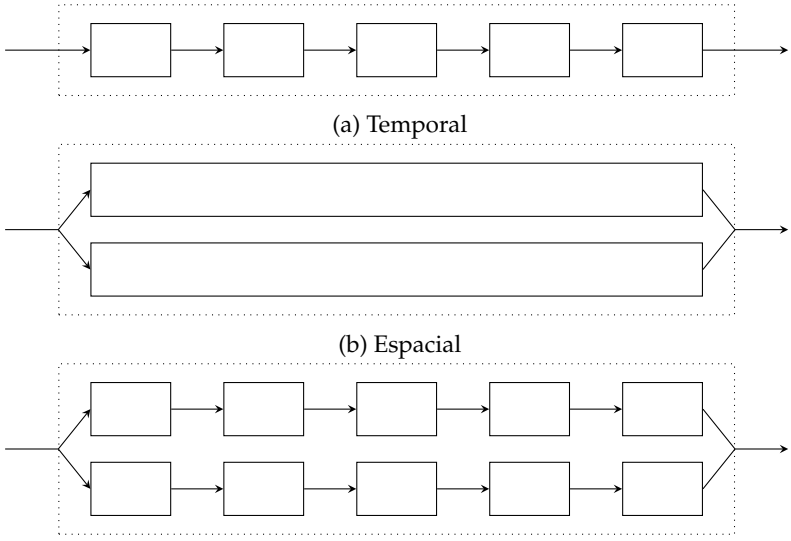
2.4. Emisión múltiple de instrucciones

Como se ha visto en la sección anterior, una CPU segmentada alcanza en condiciones ideales un CPI igual a 1, o dicho de otra forma, consigue una productividad de una instrucción por ciclo de reloj. Para mejorar el rendimiento de la CPU se debe actuar reduciendo el producto de los tres factores que proporcionan el tiempo de CPU en la ley de hierro:

$$\mathrm{T_{CPU}} = \text{Instrucciones del programa} \times \text{CPI} \times T$$

Para una determinada arquitectura del juego de instrucciones, el número de instrucciones del programa depende del compilador y se puede asumir constante para el mismo programa. Por tanto, las opciones de mejora son reducir el periodo de reloj o idear microarquitecturas capaces de conseguir valores de CPI inferiores a uno, es decir, ejecutar más de una instrucción cada ciclo de reloj. En el apartado anterior se indicó que existen unos límites prácticos en el periodo de reloj derivados del muro de potencia. Teniendo en cuenta estos condicionantes, la opción que queda para mejorar el rendimiento consiste en ejecutar más de una instrucción por ciclo de reloj.

Al número de instrucciones por ciclo de reloj se le denomina instrucciones por ciclo, o IPC, y está relacionado directamente con el CPI, pues se cumple $\text{IPC} = 1/\text{CPI}$. El interés se centrará entonces en diseñar microarquitecturas para conseguir valores



(a) Temporal

(b) Espacial

(c) Temporal y espacial

Figura 2.53: Tipos de paralelismo en la ejecución de instrucciones

de IPC mayores que uno. Para ello, es necesario explotar más todavía el paralelismo de las instrucciones.

La segmentación es una de estas organizaciones, en la que cada etapa del camino de datos está especializada en una parte de la instrucción, consiguiendo ejecutar varias instrucciones simultáneamente. La segmentación emplea un tipo de paralelismo denominado paralelismo temporal, esquematizado en la figura 2.53a.

Otra forma de paralelismo, denominada paralelismo espacial, consiste en replicar el camino de datos de la CPU para permitir la ejecución de varias instrucciones completas a la vez. La figura 2.53b muestra este tipo de organización.

Finalmente, se pueden combinar el paralelismo temporal y espacial replicando el camino de datos segmentado, dando lugar a varios cauces segmentados de ejecución, tal como se muestra esquemáticamente en la figura 2.53c. Esta técnica se conoce como emisión múltiple de instrucciones y consigue que la CPU ejecute más de una instrucción por ciclo. Al número de instrucciones que pueden procesarse en la misma etapa al mismo tiempo se le denomina ancho de emisión. Por ejemplo, para el camino de datos de la figura 2.53c, el ancho de emisión es 2, lo que permitiría alcanzar un IPC de 2 (o un CPI de 0.5) en condiciones ideales, sin necesidad de doblar la frecuencia de reloj.

Aunque la figura 2.53c representa conceptualmente una CPU con emisión múltiple de instrucciones, no se corresponde con la organización real de este tipo de CPU. Antes de mostrar una organización más realista resulta conveniente establecer los objetivos y restricciones de las CPU de emisión múltiple de instrucciones:

1. Planificación de instrucciones. Al existir varios cauces de ejecución posibles, incluso con diferentes unidades de ejecución, es necesario planificar correctamente la ejecución de instrucciones. No sólo es necesario decidir a qué cauce

se lleva cada instrucción, sino resolver las dependencias de datos cruzadas que puedan aparecer entre las instrucciones de los diferentes cauces. La función de la planificación de instrucciones puede recaer en el compilador, la CPU o ambos.

2. Cada etapa del cauce debe ser capaz de procesar al menos tantas instrucciones como el ancho de emisión de la CPU.

Existen básicamente dos formas de implementar la emisión múltiple de instrucciones, que se diferencian en cómo se reparten la responsabilidad de lidiar con las anteriores cuestiones el compilador y la CPU: emisión múltiple con planificación estática de instrucciones y emisión múltiple con planificación dinámica de instrucciones.

En la emisión múltiple con planificación estática, el compilador empaqueta en un código de instrucción muy largo varias operaciones, cada una asociada a una de las unidades de ejecución de la CPU. Por ejemplo, si la CPU dispone de 4 unidades de enteros, la instrucción podría incluir cuatro operaciones independientes de suma de registros. Es función del compilador llevar a cabo el empaquetado de las operaciones para aprovechar el paralelismo del programa respetando las dependencias entre instrucciones. A las microarquitecturas que siguen este enfoque se las conoce como **VLIW** (*Very Long Instruction Word*). En este tipo de microarquitecturas el compilador debe ajustarse minuciosamente a la microarquitectura empleada, por lo que cambios en la microarquitectura, incluso manteniendo la misma arquitectura del juego de instrucciones, requieren un esfuerzo importante para la adaptación del compilador.

En la emisión múltiple con planificación dinámica se descarga al compilador de la toma de decisiones en cuanto a la planificación y la resolución de riesgos, siendo la CPU responsable única. No obstante todavía, en menor medida, la colaboración del compilador puede ayudar a mejorar el rendimiento. Esta microarquitectura recibe el nombre de **superescalar**[11], en contraposición con una microarquitectura escalar, que puede ejecutar a lo sumo una instrucción por ciclo de reloj. La principal diferencia respecto a las microarquitecturas VLIW es que las superescalares garantizan la correcta ejecución de las instrucciones sin ayuda del compilador. En una CPU VLIW no está garantizada la correcta ejecución de las instrucciones sin la ayuda del compilador.

Actualmente, la mayor parte de las CPU modernas son superescalares, por lo que se centrará la atención en las mismas.

El paralelismo a nivel de instrucción es un parámetro clave de los programas que permite sacar partido de las microarquitecturas de emisión múltiple y en particular de las microarquitecturas superescalares. Por esta razón se trata en más detalle a continuación.

[11] También existen CPU superescalares con planificación estática, pero no se considerarán, pues el funcionamiento es similar a las CPU VLIW. En este texto se utilizará el término CPU superescalar para referirse a una CPU superescalar con planificación dinámica de instrucciones.

```
(1) dadd r1, r3, r4      (1) dadd r3, r2, r1
(2) xor r2, r5, r1       (2) xor r6, r4, r1
(3) dsub r2, r4, r7      (3) dsub r8, r4, r7
(4) movz r7, r4, r6      (4) movz r9, r4, r5
(5) slt r9, r7, r6       (5) slt r10, r7, r5
```

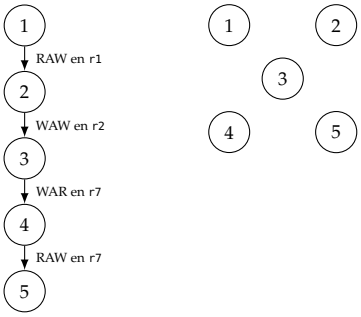


Figura 2.54: Ejemplos de paralelismo a nivel de instrucción

2.4.1. Paralelismo a nivel de instrucción

La independencia de las instrucciones de un programa es fundamental para obtener un buen rendimiento con las CPU segmentadas, pero lo es aún más en el caso de las CPU con emisión múltiple de instrucciones. Los riesgos de control se pueden reducir empleando predicción de saltos, los estructurales replicando unidades o implementando dispositivos multipuerto, pero los riesgos de datos forman parte de la semántica del programa. Cuando hay muchas instrucciones cercanas con dependencias de datos el cauce sufre continuas detenciones que penalizan el rendimiento.

El paralelismo a nivel de instrucción o **ILP** (*Instruction Level Parallelism*) se emplea para referirse al nivel de independencia entre las instrucciones de un programa. La figura 2.54 muestra dos fragmentos de código con dos situaciones opuestas en cuanto a paralelismo a nivel de instrucción. En el fragmento de la izquierda todas las instrucciones salvo la primera podrían teóricamente sufrir detenciones por dependencias de datos (aunque algunas son evitables usando renombrado de registros). Justo debajo se encuentra un grafo que representa estas dependencias entre las instrucciones. En esta situación, el rendimiento de una CPU con emisión múltiple de instrucciones sería muy deficiente, es decir, el valor de IPC sería muy inferior al ancho de emisión de la CPU.

En la parte derecha de la figura 2.54 se muestra un fragmento de código sin dependencias de datos, lo que permite aprovechar al máximo una CPU con emisión múltiple de instrucciones y hacer que su rendimiento se acerque al ideal. El grafo correspondiente aparece justo debajo y se observa que todos los nodos están desconectados al no existir dependencias.

Los grafos de dependencias proporcionan información visual no sólo sobre las dependencias entre las instrucciones, sino además sobre el mínimo tiempo posible de ejecución de un programa. La ruta más larga dentro del grafo indica de forma

aproximada el tiempo mínimo de ejecución posible. Aunque el hardware disponible fuese ilimitado, esto es, se dispusiese de una CPU con un ancho de emisión arbitrario, el tiempo de ejecución no podría reducirse por debajo del asociado a la ruta más larga, a la que se denomina ruta crítica.

En resumen, el grado de ILP de los programas, o de forma análoga, las dependencias entre las instrucciones, fijan unos límites teóricos en cuanto al mínimo tiempo de ejecución de un programa incluso disponiendo de hardware ilimitado. Al fin y al cabo, si unas instrucciones tienen que esperar por otras no es posible ejecutarlas en paralelo.

2.4.2. Microarquitectura superescalar

La figura 2.55 muestra a nivel de bloques una microarquitectura superescalar para MIPS64 de ancho 2. La organización de una CPU de mayor ancho sería similar. La ejecución de las instrucciones de divide en cinco etapas:

- Lectura de instrucciones (`IF`).
- Decodificación (`ID`).
- Distribución (`DT`).
- Ejecución (`EX`).
- Retirada (`RT`).

Las etapas no coinciden exactamente con las de la microarquitectura segmentada estudiada, pero la mayor parte son similares conceptualmente. Las etapas `IF`, `ID` y `EX` son análogas a las de la microarquitectura segmentada en cuanto a las funciones realizadas. La etapa de retirada, `RT`, realiza una función análoga a la de las etapas `MEM` y `WB` en conjunto. La etapa de distribución `DT` es nueva y no coincide con ninguna de las etapas de la CPU segmentada. Todas las etapas son capaces de procesar dos instrucciones a la vez, excepto la etapa `EX` que dispone de 6 unidades de ejecución: dos de enteros y operaciones lógicas, una de carga y almacenamiento, una de evaluación de saltos, una de punto flotante y otra de multiplicación y división de enteros. Estas unidades de ejecución requieren diferente número de ciclos. En el caso de necesitar más de uno, las unidades se segmentan para mejorar el rendimiento. Dependiendo de la combinación de instrucciones podría haber hasta 6 instrucciones simultánemente en la etapa `EX`.

Entre todas las etapas se encuentran *buffers* multipuerto que permiten llevar a cabo varias lecturas y escrituras en paralelo. Se trata de registros de más capacidad y más complejos que los existentes entre las etapas de las CPU segmentadas.

Es importante reseñar que una CPU superescalar se divide en tres secciones, cada una de las cuales abarca una o varias etapas:

- Sección inicial en orden. Las instrucciones se leen de la memoria, se decodifican y se distribuyen en el orden del programa.

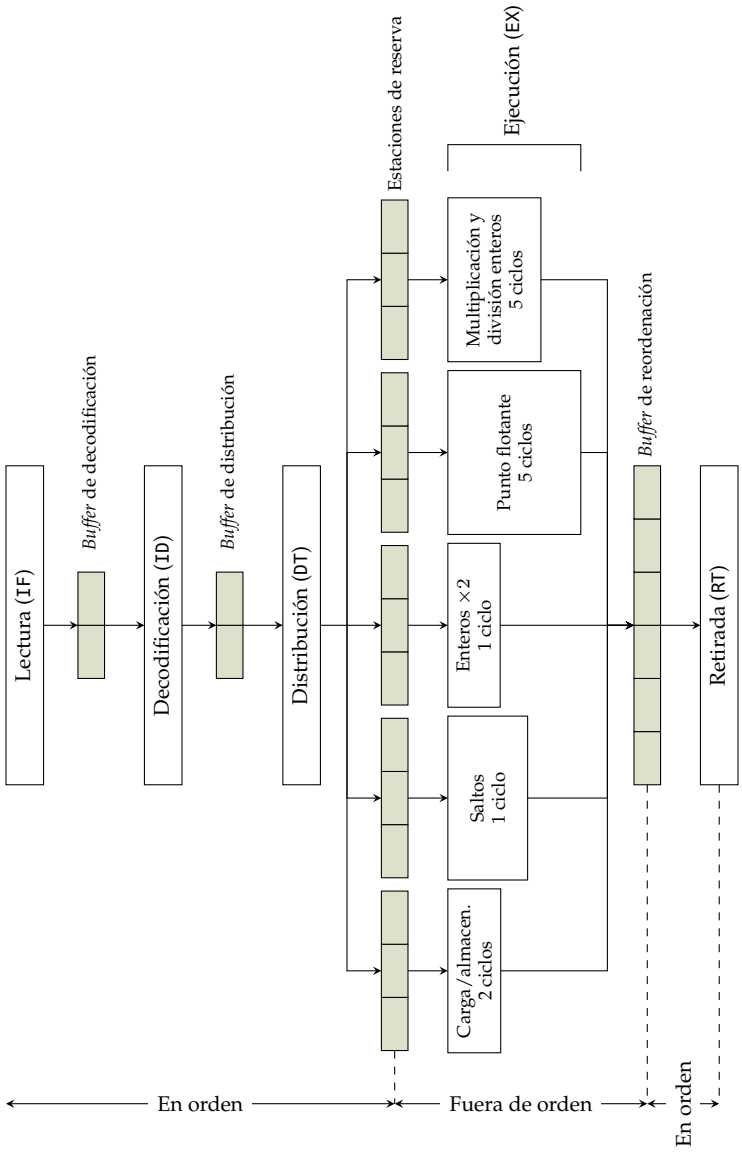


Figura 2.55: Organización de una CPU superescalar

- Sección de emisión y ejecución fuera de orden. Las instrucciones se emiten y ejecutan en la unidad de ejecución correspondiente fuera del orden especificado en el programa para aprovechar al máximo el paralelismo a nivel de instrucción.
- Sección final en orden. Las instrucciones se retiran en el orden indicado en el programa para que las escrituras en memoria y registros ocurran en el orden adecuado y dar el soporte necesario a las excepciones precisas.

A continuación se describirán en detalle las etapas de la CPU superescalar.

- La etapa `IF` lee de la memoria de programa un número de instrucciones igual al ancho de emisión de la CPU superescalar, en este caso 2. La lectura de las dos instrucciones se hace a la vez, por lo que se lee una doble palabra (64 bits). Esto significa que se leen dos instrucciones que se encuentran en direcciones de memoria consecutivas a partir de la dirección indicada por el `PC`.

  Los accesos a la memoria de programa están alineados a doble palabra, es decir, 8 bytes. Esto tiene sus implicaciones durante el cambio del flujo del programa pues las direcciones destino de salto están alineadas a palabra, es decir, 4 bytes.

  Durante esta etapa también se realiza la predicción de saltos tal como se ilustró en la sección 2.3.6.

- La etapa `ID` es una de las más complejas. Decodifica al mismo tiempo dos códigos de instrucción (tantos como el ancho de la CPU), lee los registros fuente disponibles de ambas instrucciones (hasta 4 registros simultáneamente), etiqueta los que faltan (porque estén afectados por una dependencia RAW) y renombra los registros destino (para evitar riesgos por dependencias de datos WAR y WAW).

- La etapa `DT` extrae hasta dos instrucciones del *buffer* de distribución y las distribuye a unos *buffers*, denominados estaciones de reserva, de las unidades de ejecución correspondientes. Típicamente, hay una estación de reserva asociada a cada tipo de unidad de ejecución. Cada una de las estaciones de reserva puede almacenar varias instrucciones que esperan el procesamiento por parte de una unidad de ejecución asociada. Si este *buffer* está lleno cuando se distribuye una instrucción se produce una detención por riesgo estructural.

  Las estaciones de reserva son unos *buffers* multipuerto que requieren especial atención. Cada estación de reserva recibe instrucciones de un cierto tipo. Junto con la instrucción se almacena el valor de los registros disponibles e identificadores para los registros no disponibles todavía. Por ejemplo, en la figura 2.55 aparecen estaciones de reserva para instrucciones de carga/almacenamiento, otra para saltos, otra para operaciones de aritmética sobre enteros y operaciones lógicas, otra para instrucciones de punto flotante y otra para multiplicación/división de enteros. Las estaciones de reserva marcan la frontera entre la primera parte del *pipeline* en orden y la emisión fuera de orden.

  Las instrucciones se mantienen a la espera en su estación de reserva hasta que están listos todos sus operandos. Mientras una instrucción se encuentra esperando por alguno de los operandos se está comprobando continuamente el bus

de reenvío, en el que las instrucciones publican el valor de los registros que modifican al salir de su etapa de ejecución. Una vez la entrada de la estación de reserva detecta en el bus una coincidencia entre un registro publicado y alguno de los registros pendientes de la instrucción, el valor del registro pasa a almacenarse junto a la instrucción en la entrada de la estación de reserva. Cuando la instrucción tiene todos sus operandos se emite para ejecutarse tan pronto como haya una unidad de ejecución disponible.

- Etapa EX. Las CPU superescalares disponen de varias unidades de ejecución especializadas. Las instrucciones son ejecutadas fuera de orden en estas unidades. El orden viene dado por el momento de emisión y la disponibilidad de una unidad de ejecución. Una vez las instrucciones son ejecutadas se almacenan en el *buffer* de reordenación y se publica en el bus de reenvío el valor del registro que modifica. Además, si se ha producido una excepción en la ejecución de la instrucción, se etiqueta la instrucción dentro del *buffer* de reordenación.

  La instrucción al atravesar esta etapa todavía no ha acabado, pero la publicación del registro modificado permite que las instrucciones con dependencia RAW sobre ese registro puedan emitirse. Como puede observarse, se trata de una generalización de las rutas de reenvío.

  En el caso de instrucciones de carga o almacenamiento, en esta etapa se calcula la dirección de memoria y se almacena la operación de acceso memoria en el *buffer* de reordenación junto con otra información de la instrucción, como los registros fuente o destino.

- La etapa de RT extrae las instrucciones del *buffer* de reordenación en el orden marcado por el programa. De esta forma, además de soportar excepciones precisas, se garantiza el cumplimiento de las dependencias RAW, WAW y WAR entre operandos de memoria[12].

  Una instrucción para poder ser retirada tiene que estar marcada como no especulativa. Cuando se hace una predicción de salto las instrucciones que siguen a la de salto se marcan como especulativas. Más tarde cuando se evalúa la condición de salto, si la predicción fue correcta las instrucciones ejecutadas pasan de especulativas a no especulativas y pueden ser retiradas. Si la predicción fue incorrecta, las instrucciones especulativas son desechadas.

  En el caso particular de las instrucciones de almacenamiento, las escrituras no se llevan directamente a la memoria de datos, sino que se llevan a un *buffer* de almacenamiento dentro de la CPU en el orden del programa. Aunque el funcionamiento del *buffer* está fuera de los objetivos de este capítulo, conviene comentar que su objetivo es acelerar las lecturas y escrituras en memoria.

Para comprender las implicaciones en la ejecución de instrucciones en una microarquitectura superescalar, la figura 2.56 muestra la ejecución de las 8 instrucciones de un programa en la CPU MIPS64 superescalar planteada:

[12] Con la ejecución de instrucciones fuera de orden, el cumplimiento de las dependencias de memoria ya no es trivial como ocurría en la microarquitectura segmentada MIPS64 vista.

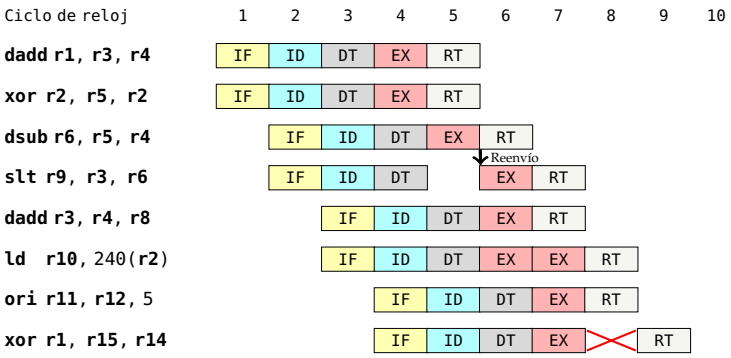


Figura 2.56: Ejecución superescalar

- Las dos primeras instrucciones son leídas al mismo tiempo desde memoria en la etapa `IF`. Se leen 64 bits que se corresponden con los códigos de estas dos instrucciones.

  Las dos instrucciones se decodifican al mismo tiempo en el ciclo 2, se comprueba que no existen dependencias entre ellas, se renombran los registros destino y se leen los 4 registros fuente desde el fichero de registros. Si alguno dependiese del resultado de una instrucción previa se etiquetaría para indicar esta situación (no es el caso).

  Las instrucciones se distribuyen a las estaciones de reserva en el ciclo 3. En este caso, al tratarse de una operación aritmética sobre enteros y de una operación lógica, ambas instrucciones se distribuyen a la estación de reserva de las unidades de ejecución de enteros.

  En el ciclo 4 ambas instrucciones son emitidas y ejecutadas por las unidades de enteros (una en cada una de las dos unidades disponibles). Al final de este ciclo las instrucciones, ya ejecutadas, se almacenan en el *buffer* de reordenación a la salida de la etapa `EX` para mantener el orden del programa. También se publican los valores calculados para `r1` y `r2` en el bus de reenvío por si alguna instrucción esperando en una estación de reserva los necesitase.

  Durante el ciclo 5 los resultados de ambas instrucciones son escritos en el fichero de registros en el orden del programa.

- Las siguientes dos instrucciones se leen en el ciclo 2. Se trata de dos instrucciones aritméticas sobre enteros, por lo que en principio podrían ejecutarse al mismo tiempo. Sin embargo existe una dependencia RAW entre ambas sobre el registro `r6`. Por este motivo, la instrucción `slt r9, r3, r6` debe esperar en la estación de reserva hasta que la instrucción `dsub r6, r5, r4` compute el valor de `r6` en el ciclo 5.

  En el ciclo 6 el bus de reenvío posibilita que esté disponible el valor del registro `r6`, por lo que la instrucción `slt r9, r3, r6` se emite y ejecuta en una unidad de ejecución de enteros. Esta última será retirada (etapa `RT`) por tanto un ciclo después.

- Las siguientes instrucciones se cargan en el ciclo 3. Se trata de una instrucción aritmética sobre enteros y una instrucción de carga, por lo que pueden ejecutarse al mismo tiempo; una sobre una unidad de enteros y otra sobre la unidad de carga/almacenamiento. Esta última unidad requiere 2 ciclos, mientras que la unidad de enteros termina su trabajo en un único ciclo. De ahí que una instrucción sea retirada en el ciclo 7 y la otra en el 8.

  Es importante destacar cómo en el ciclo 6 hay tres instrucciones en la etapa EX; un número mayor que el ancho de emisión de la CPU. Dos instrucciones están utilizando sendas unidades de ejecución de enteros y la tercera está en la unidad de carga/almacenamiento. Ocurre lo mismo en el ciclo 7.

- Las siguientes dos instrucciones se cargan en el ciclo 4. Son dos instrucciones lógicas, por lo que podrían ejecutarse al mismo tiempo sobre las dos unidades de ejecución de enteros disponibles. Sin embargo, en el ciclo 8 es retirada la instrucción de carga `ld r10, 240(r2)`, por lo que solo la primera de estas dos nuevas instrucciones puede ser retirada en ese mismo ciclo. La instrucción `xor r1, r15, r14` debe esperar al ciclo 9 para ser retirada. Se produce una detención estructural, pues la unidad de retirada ya está trabajando sobre dos instrucciones en el ciclo 8.

Ignorando el transitorio inicial de 4 ciclos, para la ejecución de las 8 instrucciones anteriores son necesarios 5 ciclos (del 5 al 9), lo que resulta en IPC $= 8/5 = 1.6$, o lo que es lo mismo, CPI $= 0.625$, significativamente inferior al límite mínimo teórico de 1 de una CPU escalar segmentada.

Influencia de la ejecución especulativa en la seguridad

Como se ha comentado, en las arquitecturas actuales es habitual utilizar técnicas de ejecución especulativa tales como la predicción de saltos y la ejecución fuera de orden para obtener un rendimiento elevado. Esto supone que en ocasiones deben descartarse los efectos de instrucciones ejecutadas especulativamente que no deberían haberse ejecutado. Por ejemplo, ante un fallo en la predicción de un salto o una excepción en una instrucción ejecutada fuera de orden. Esto puede ser muy complicado y, de hecho, ha sido la causa de una serie de vulnerabilidades en los principales procesadores (sobre todo de Intel y AMD, aunque también de ARM). Las primeras en encontrarse, en 2017, fueron denominadas Meltdown y Spectre, y combinan la ejecución especulativa con otros conceptos arquitectónicos como las cachés, los niveles de privilegio y la gestión de excepciones.

Aunque el detalle de cómo se explotan es distinto en cada caso, la idea general es siempre la misma: intentar adivinar valores (como por ejemplo contraseñas o números de tarjetas de crédito) que no deberían ser accesibles a un programa ejecutando especulativamente instrucciones que acceden a direcciones de memoria prohibidas para ese programa; cuando se descubre que esas instrucciones no deberían ejecutarse, se deshacen sus efectos, pero dejan un rastro de su ejecución especulativa. Este rastro puede ser el cambio en el tiempo de ejecución del programa. Midiendo los cambios en el tiempo de ejecución se pueden llegar a obtener valores en teoría no

accesibles, a través de lo que se denomina un «canal lateral» (*side channel*), denominación utilizada porque no se trata de acceder a la información por el canal principal sino hacerlo de manera indirecta.

Estas vulnerabilidades están obligando a los diseñadores de CPU a modificar sus diseños para intentar evitar estos ataques. Por ejemplo, uno de los ataques se basa en, desde un hilo de un programa, entrenar incorrectamente las tablas de predicción de saltos para obtener los datos de un hilo de otro programa midiendo los cambios en tiempos de ejecución. Una solución a este problema es vaciar las tablas de predicción cada vez que se cambia de hilo en ejecución. Otra alternativa es tener tablas de predicción de saltos independientes para cada hilo. En cualquier caso, las dos alternativas plantean penalizaciones de rendimiento o incremento de costes, lo que supone nuevos y muy complejos problemas para los diseñadores de CPU.

2.5. Ley de Moore

Las mejoras microarquitectónicas tratadas en los apartados anteriores y las mejoras en el proceso de fabricación de las CPU persiguen un objetivo común: conseguir incrementos cada vez mayores en el rendimiento de la CPU. Desgraciadamente, la tecnología actual está rozando unos límites que impiden seguir mejorando el rendimiento. Este apartado describe estos límites.

Se conoce como «Ley de Moore» (*Moore's law*) a la idea de que la potencia computacional se incrementará drásticamente, a la vez que se reducirá su coste, a un ritmo exponencial. Se basa en una predicción hecha por Gordon Moore (cofundador de Intel) en 1965.

La predicción parte de las mejoras en el proceso de fabricación de procesadores[13]. Los procesadores se fabrican mediante circuitos integrados o chips, que se construyen a partir de obleas de cristal de silicio (*silicon wafers*). Los componentes del circuito, tales como transistores o cables, son estampados en la superficie de la oblea utilizando técnicas de grabado químico. En una sola oblea se estampan tantos chips como es posible para aprovechar al máximo su superficie y ahorrar costes.

A lo largo de los años se ha ido mejorando lo que se conoce como «tecnología de fabricación», que hace referencia al tamaño del componente más pequeño que puede ser estampado en la superficie de la oblea. Las mejoras en el proceso de fabricación permiten fabricar transistores más pequeños y rápidos, e incluir más transistores por unidad de superficie. La figura 2.57 muestra la evolución en el número de transistores de los procesadores a lo largo de los años en escala logarítmica.

Estas mejoras en el proceso de fabricación aparecen reflejadas en los factores de la ley de hierro que proporcionan el tiempo de CPU. A continuación se analizarán en detalle.

$$\mathrm{T_{CPU}} = \text{Instrucciones del programa} \times \text{CPI} \times T$$

[13] En este texto se utiliza el término CPU para referirse a una unidad de procesamiento, mientras que el término procesador se refiere al dispositivo que tiene su propio encapsulado y que incluye una o más CPU. En el caso de incluir una única CPU, es común utilizar los términos procesador y CPU indistintamente.

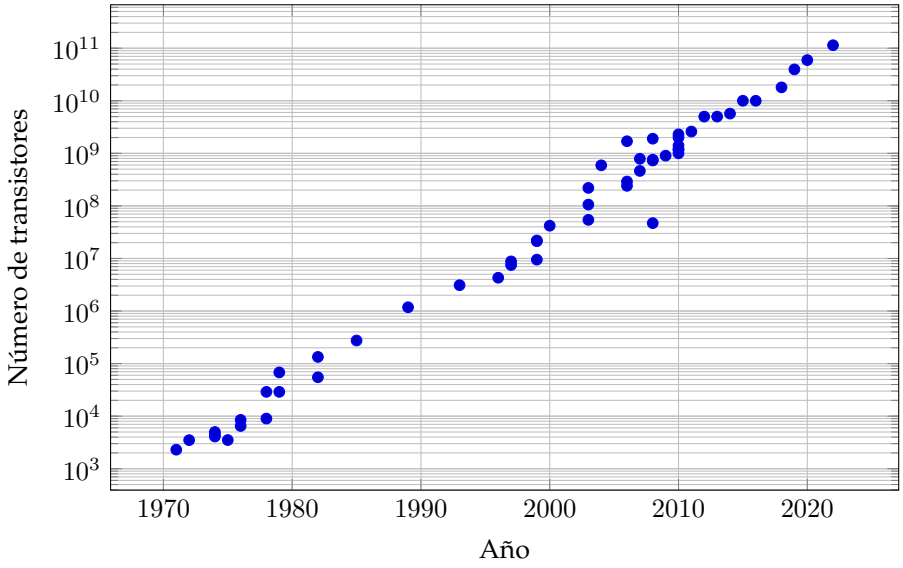


Figura 2.57: Número de transistores en los procesadores en las últimas décadas

Disponer de más transistores proporciona la oportunidad de introducir mejoras en la microarquitectura, como las que se han mostrado en los apartados anteriores. Durante los primeros años de desarrollo de las CPU superescalares se emplearon los transistores extra disponibles para diseñar CPU superescalares con un ancho de emisión cada vez mayor hasta agotar el grado de paralelismo de los programas. El problema es que los programas tienen un nivel de ILP limitado, por lo que llega un momento en el que aumentar el ancho de emisión no reduce el CPI y, por tanto, no mejora el rendimiento. La consecuencia desde el punto de vista del fabricante de la CPU es que ya no le interesa complicar más la microarquitectura, pues no redunda en mejoras de rendimiento. Las CPU superescalares actuales tienen habitualmente un ancho de emisión igual o inferior a cuatro.

Disponer de transistores más rápidos permite incrementar la frecuencia de trabajo y el rendimiento para una microarquitectura, pues reduce el periodo de reloj. El inconveniente es que el consumo energético del procesador se incrementa exponencialmente con la frecuencia de trabajo. Mayor frecuencia de trabajo implica mayor potencia consumida y por consiguiente mayor potencia disipada. La disipación de potencias superiores a 100 W en dispositivos de un centímetro cuadrado aproximadamente plantea problemas que requieren soluciones tecnológicas muy costosas.

En la figura 2.58 se muestra el aumento de la frecuencia de los procesadores en las últimas décadas con escala logarítmica. Debido al consumo y a la disipación de calor, el incremento de la frecuencia se detuvo aproximadamente en 2005. El límite tecnológico que impide aumentar la frecuencia de forma continuada para aumentar la potencia computacional se denomina muro de potencia (*Power Wall*). Este muro de potencia impidió que las frecuencias aumentasen más allá de los 4 GHz.

Teniendo en cuenta el ILP máximo de los programas y la imposibilidad de incrementar la frecuencia de trabajo, el crecimiento exponencial de la potencia compu-

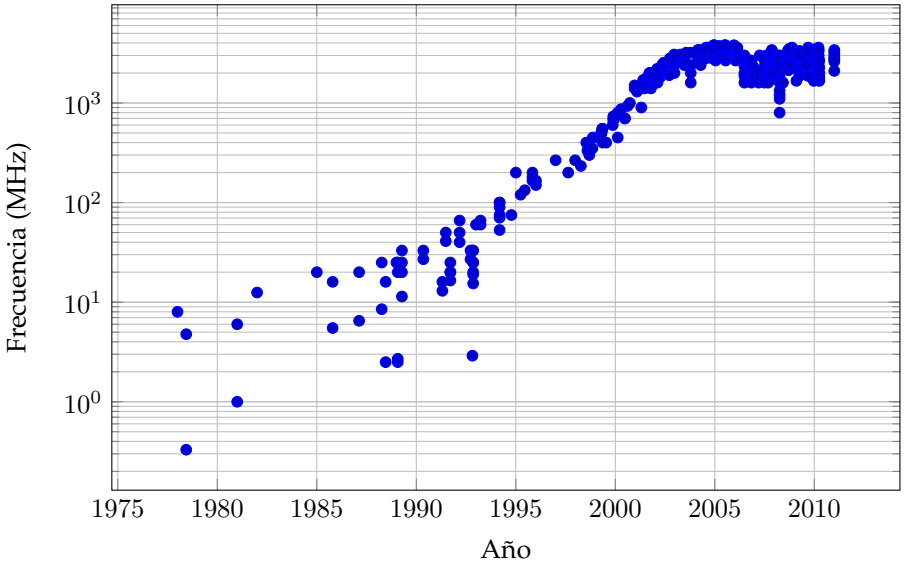


Figura 2.58: Frecuencia de los procesadores Intel en las últimas décadas

tacional se ha ralentizado en los últimos años. Más aún, la introducción de nuevas tecnologías de fabricación requiere de inversiones cada vez más grandes por parte de las compañías, por lo que estas nuevas tecnologías aparecen cada vez más espaciadas en el tiempo. Esto no significa que no se haya incrementado el rendimiento de las CPU desde entonces. Sí que ha aumentado, pero de forma mucho más pausada, empleando otras formas de paralelismo que se describen en el apartado 2.6.

A partir de las limitaciones del ILP y potencia se plantean fundamentalmente tres formas de aprovechar las mejoras de fabricación de los circuitos integrados, ilustradas en la figura 2.59:

1. Producir la misma funcionalidad en menor superficie y mantener la frecuencia. La reducción de la superficie supone una ventaja desde el punto de vista del coste, ya que se pueden producir más procesadores por oblea. Además, tiene otra importante ventaja: se reduce el consumo energético porque éste es proporcional a la superficie. El consumo energético es un parámetro clave en los procesadores de dispositivos móviles y de IoT (*Internet of Things*), por lo que esta alternativa es interesante en la construcción de los procesadores.

2. Producir la misma funcionalidad en menor superficie e incrementar la frecuencia. La idea es aprovechar la reducción de consumo que se obtiene al reducir la superficie para incrementar la frecuencia, lo que de nuevo aumenta el consumo, hasta que se compense la reducción de un lado con el incremento del otro. El problema de esta alternativa es que las reducciones de consumo al reducir la superficie son rápidamente absorbidas por pequeños incrementos de frecuencia. La reducción de la potencia es proporcional a la reducción de la superficie, pero crece de forma exponencial con la frecuencia. Por esta razón,

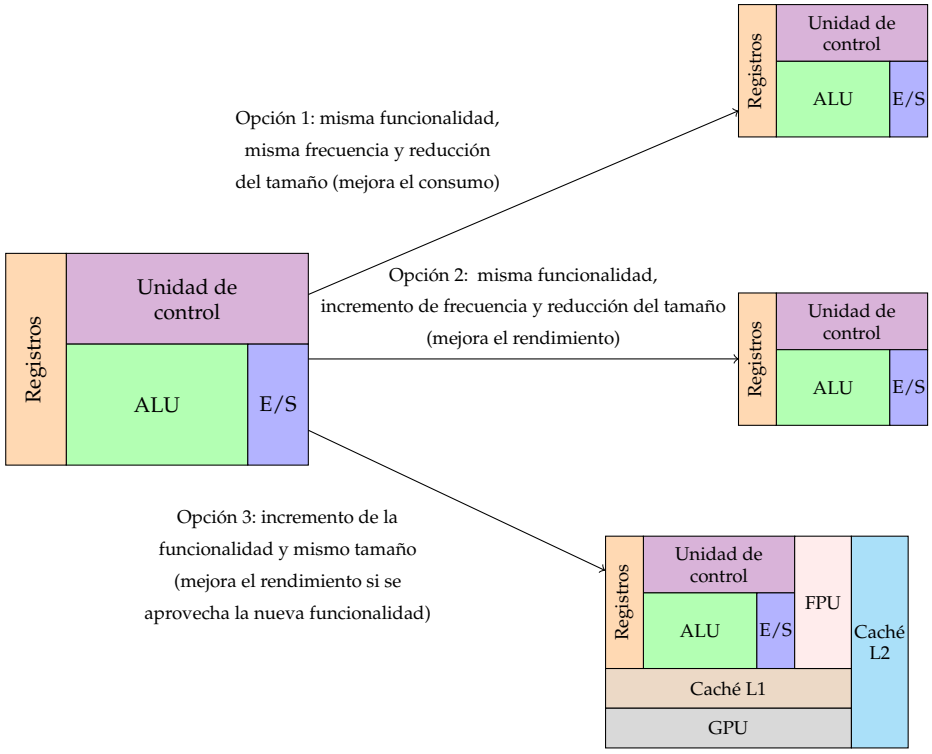


Figura 2.59: Posibilidades de mejora tras reducir la tecnología de fabricación

los incrementos de frecuencia posibles con las mejoras de fabricación son muy bajos.

3. Añadir más funcionalidad para la misma superficie y mantener la frecuencia. El incremento del número de transistores permite integrar dentro del chip del procesador otros componentes del computador como dispositivos de E/S, memorias, etc. Con esto se consiguen incrementos de rendimiento, pues las comunicaciones entre estos componentes y el procesador son más rápidas. Por ejemplo, la integración de parte de la memoria dentro del chip del procesador reduce los tiempos de acceso a memoria. Otra posibilidad de añadir más funcionalidad consiste en integrar varias CPU dentro del mismo chip dando lugar a configuraciones multinúcleo. Este tipo de configuraciones se describen en el apartado 2.6.

En resumen, las mejoras de fabricación tradicionalmente se aprovechaban para implementar microarquitecturas cada vez más sofisticadas e incrementar la frecuencia de reloj. Una vez que este camino ya no permite seguir mejorando el rendimiento, se ha pasado a emplear nuevas técnicas, que si bien consiguen incrementos de rendimiento más modestos, permiten seguir progresando. El apartado siguiente describe alguna de estas técnicas.

2.6. CPU multihilo

La forma habitual de incrementar el rendimiento más allá de los límites impuestos por la tecnología de fabricación consiste en incrementar el nivel de paralelismo del sistema. Hasta ahora se ha aprovechado el paralelismo a nivel de instrucción para conseguir mejoras de rendimiento, pero existen otras formas de paralelismo, descritas en la taxonomía de Flynn.

2.6.1. Taxonomía de Flynn

La taxonomía de Flynn es una clasificación de arquitecturas de computadores basada en el número de instrucciones concurrentes y los flujos de datos disponibles, propuesta en la década de 1970 y que aún se utiliza a día de hoy. A partir de este criterio los arquitecturas pueden clasificarse en cuatro grandes grupos: SISD, SIMD, MISD y MIMD. A continuación se presentan las principales características de cada grupo.

SISD (*Single Instruction*, *Single Data*). Describe la organización de un computador que ejecuta un flujo de instrucciones único sobre un flujo de datos único. Este es el caso de las arquitecturas von Neumann o Harvard consideradas hasta ahora.

SIMD (*Single Instruction*, *Multiple Data*). Describe la organización de un sistema que ejecuta un único flujo de instrucciones sobre múltiples flujos de datos, es decir, ejecuta la misma instrucción sobre varios datos a la vez, lo que redunda en una sensible mejora de rendimiento. Por ejemplo, si es necesario realizar la suma de los elementos de igual índice en dos vectores de cuatro elementos para obtener un tercer vector de cuatro elementos con la suma, en lugar de emplear un bucle de cuatro repeticiones que sume cada pareja de elementos de igual índice, se puede emplear una unica instrucción SIMD que realice las cuatro sumas a la vez. Esto supone un gran ahorro en el tiempo de búsqueda y ejecución de instrucciones.

Un ejemplo de sistema SIMD es la unidad de procesamiento gráfico (GPU) de un computador personal, ya que en el tratamiento de gráficos es habitual tener que hacer la misma operación sobre varios píxeles de una imagen. Las CPU empleadas en los computadores personales actuales también disponen de instrucciones SIMD. Las aplicaciones multimedia y de procesamiento de señal son habitualmente las que más partido sacan a las instrucciones SIMD por exhibir este tipo de paralelismo.

MISD (*Multiple Instruction*, *Single Data*). Describe la organización de un sistema que ejecuta múltiples flujos de instrucciones sobre un flujo de datos único.

Los sistemas MISD no emplean el paralelismo para conseguir mejoras de rendimiento, sino para ser tolerantes a fallos, pues ejecutan la misma instrucción en paralelo sobre los mismos datos para detectar errores en tiempo de ejecución. Se emplean en sistemas críticos, como por ejemplo en sistemas de aviación.

MIMD (*Multiple Instruction*, *Multiple Data*). Describe la organización de un sistema que ejecuta múltiples flujos de instrucciones sobre múltiples flujos de datos. Se trata de la generalización natural de la arquitectura von Neumman o Harvard.

Existen dos alternativas para implementar un sistema MIMD con varias CPU y varios dispositivos de memoria:

- MIMD de memoria distribuida. Consiste en replicar los pares CPU/memoria y conectarlos mediante una red de comunicación. Cada memoria sólo puede ser accedida por la CPU que tenga asignada.
- MIMD de memoria compartida. Consiste en construir un conjunto de varias CPU y dispositivos de memoria de forma que cualquier CPU pueda acceder a cualquier dispositivo de memoria a través de un mecanismo de interconexión.

La mayor parte de los procesadores actuales optan por un sistema MIMD de memoria compartida. La memoria principal es compartida entre todas las CPU del sistema. Este es el caso de los sistemas multinúcleo que se presentan en el apartado 2.6.3. Con esto se consigue incrementar el rendimiento más allá de los límites marcados por el ILP de los programas y el muro de potencia. La idea clave es que con esta organización se pueden ejecutar varios programas (o hilos) de forma concurrente.

2.6.2. Paralelismo a nivel de hilo de ejecución

Un hilo de ejecución es la unidad mínima que puede planificar un sistema operativo. Cada programa tiene al menos un hilo de ejecución, pero algunos programas pueden tener varios hilos. En ese caso, una forma de mejorar el rendimiento es ejecutando varios hilos compartiendo la CPU en lo que se conoce como *multithreading*. Esta técnica aprovecha el paralelismo que existe a nivel de hilo de ejecución, o *Thread Level Parallelism* (TLP). La CPU ejecuta un solo hilo en cada instante, por lo que los hilos deben compartir el tiempo de CPU. Para que esta técnica resulte eficiente el cambio de un hilo de ejecución a otro dentro de la CPU debe ser muy rápido.

Este tipo de paralelismo a nivel de hilo es distinto del paralelismo a nivel de instrucción que explotan técnicas como el *pipeline* o las arquitecturas superescalares. Dos diferencias fundamentales entre ambos tipos de paralelismo son:

- El paralelismo a nivel de instrucción mejora el rendimiento de cada hilo de ejecución individual, independientemente del número de hilos. De esta forma, se mejora el rendimiento de un programa monohilo automáticamente, sin necesidad de cambios en el paradigma de programación.
- El paralelismo a nivel de hilo de ejecución, en cambio, solo mejora el rendimiento cuando se ejecutan varios hilos a la vez. A diferencia del anterior, el paralelismo a nivel de hilo de ejecución no es transparente a los programas; requiere transformar los programas monohilo en multihilo para poder aprovechar las mejoras de rendimiento. La programación multihilo es significativamente más compleja que la monohilo, ya que requiere tener en cuenta la sincronización entre múltiples hilos.

Una evolución de *multithreading* utilizada en muchas CPU actuales es *simultaneous multithreading* (SMT). La idea subyacente es que las CPU ofrecen un número elevado de unidades funcionales paralelas y no siempre se usan todas al mismo tiempo, ya sea por el tipo de instrucciones que se están ejecutando, o porque el programa tiene un reducido ILP. De esta forma, sería posible ejecutar simultáneamente varios hilos en lugar de que compartan el tiempo de CPU. El problema es que no están replicadas todas las unidades funcionales necesarias para poder ejecutar cualquier combinación de instrucciones en multiples hilos. La implementación más conocida es la tecnología *HyperThreading* de Intel, que permite ejecutar dos hilos simultáneamente en cada CPU, pero sólo en determinadas circunstancias, por lo que el incremento de rendimiento es limitado.

2.6.3. Procesadores multinúcleo

Tal y como se ha indicado anteriormente, cuando a principios de la primera década del siglo XXI los grandes incrementos en frecuencia dejaron de ser significativos por problemas de energía y disipación de calor, el método más usado para incrementar el rendimiento fue pasar de arquitecturas con una única CPU a arquitecturas MIMD de memoria compartida siguiendo el esquema mostrado en la figura 2.60. Todas las CPU de este tipo de arquitecturas comparten una misma memoria principal, de tal forma que el tiempo de acceso a memoria es similar para todas ellas. Por esta

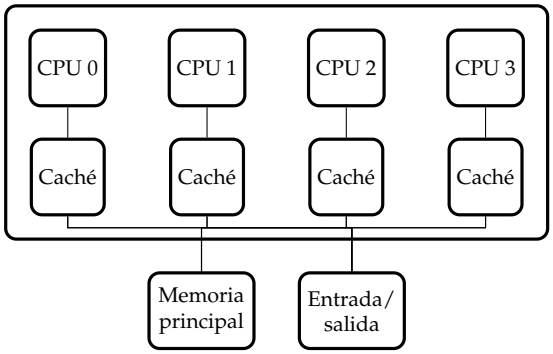


Figura 2.60: Arquitectura básica de un procesador multinúcleo

razón, a este tipo de arquitecturas se las suele denominar *Shared Memory Processors* (SMP) o *Uniform Memory Access* (UMA).

Se denomina procesador multinúcleo o *multi-core* a un procesador que contiene dos o más CPU, o *cores*. De esta forma los transistores extra proporcionados por las sucesivas mejoras en la tecnología de fabricación se emplean para integrar varias CPU dentro del mismo chip, obteniendo nuevas mejoras de rendimiento a pesar del máximo ILP de los programas y las limitaciones en la disipación de energía.

Actualmente, casi todos los procesadores multinúcleo son procesadores SMP, dado que por lo general comparten un único espacio de direcciones físicas. La tecnología multinúcleo es común en todo tipo de procesadores para computadores de sobremesa y portátiles, y está presente incluso en teléfonos móviles y tabletas.

Los procesadores multinúcleo pueden además implementar técnicas SMT, con lo que cada núcleo del procesador puede ejecutar más de un hilo simultáneamente. Por ejemplo, un procesador con cuatro núcleos que implemente SMT de tal forma que pueda llegar a planificar dos hilos en cada núcleo ofrece al sistema operativo ocho hilos de ejecución simultáneos. Hay que tener en cuenta que estos ocho hilos de ejecución no son iguales, en el sentido de que no es lo mismo planificar dos hilos dentro del mismo núcleo, donde en ocasiones van a poder ejecutarse a la vez y en otras ocasiones no (cuando necesiten una misma unidad funcional no replicada), que planificarlos en dos núcleos distintos donde no van a competir por las mismas unidades funcionales.

Aunque la implementación de procesadores multinúcleo ha supuesto un avance en el rendimiento de los computadores desde el punto de vista de su productividad, esta técnica tiene sus limitaciones. El uso de muchos núcleos hace surgir un nuevo muro: el muro de la memoria. Muchos núcleos accediendo a la memoria simultáneamente hacen que la memoria se convierta en un cuello de botella. Este efecto se ve paliado con la utilización de memorias caché. Tal y como se muestra en la figura 2.60, habitualmente cada núcleo tiene asociado uno o más niveles de memoria propia, denominada memoria caché. Estas cachés son fundamentales para reducir la dependencia de la memoria. No obstante, debido al ritmo de crecimiento actual del número de núcleos, se estima que en unos años se «chocará contra el muro de memoria». La memoria caché se estudia en detalle en el capítulo 3.

De forma general, existen tres grandes muros o límites de rendimiento: el de potencia (*power wall*), el de memoria (*memory wall*), y el de paralelismo a nivel de instrucción (*ILP wall*). La combinación de estos tres muros se la conoce como *Brick Wall*, y es una forma de describir las limitaciones tecnológicas que impiden mantener el crecimiento de potencia computacional al mismo ritmo que en el pasado.

2.7. Soporte a los SO multitarea

Hasta ahora, en este capítulo se ha presentado un ejemplo de arquitectura del juego de instrucciones, la arquitectura MIPS64, y sobre ella se han desarrollado diferentes microarquitecturas con el objetivo de mejorar el rendimiento dentro de las limitaciones tecnológicas. Si bien el rendimiento es una parte muy importante de una CPU, hay otras también muy importantes relacionadas con su funcionalidad. Uno de los requerimientos de cualquier CPU moderna es el soporte de su arquitectura a sistemas operativos multitarea. *Grosso modo*, la funcionalidad está ligada a la arquitectura del juego de instrucciones, esto es, indica qué puede hacer la CPU, mientras que el rendimiento hace referencia a la velocidad con la que se lleva a cabo dicha funcionalidad y depende de la microarquitectura. No obstante, algunos de los mecanismos de soporte a los sistemas operativos multitarea tienen consecuencias desde el punto de vista del rendimiento.

Antes de presentar el soporte de la arquitectura a los sistemas operativos multitarea, resulta conveniente introducir los conceptos básicos de un sistema multitarea.

2.7.1. Introducción a los SO multitarea

Habitualmente, el usuario de un computador no se encuentra con un computador «desnudo», es decir, un computador sin ningún tipo de software instalado. Por el contrario, lo habitual es que el usuario se encuentre un computador con un sistema operativo instalado. La unión de un computador más el sistema operativo constituye una máquina que proporciona una serie de servicios a las aplicaciones de usuario, las cuales interactúan con el hardware a través de una API (*Application Programming Interface*) perfectamente definida. Esto facilita mucho la programación de aplicaciones, pues el sistema operativo proporciona una interfaz independiente del hardware. El sistema operativo se convierte además en un gestor de los recursos hardware del computador para proveer servicios a las aplicaciones. La relación entre el hardware, el sistema operativo y las aplicaciones aparece esquematizada en la figura 2.61.

Prácticamente, todos los sistemas operativos actuales son multitarea. Esto significa que el conjunto formado por el computador y el SO soporta la multiprogramación, es decir, puede ejecutar concurrentemente varios programas.

A pesar de las funciones específicas que cumple un sistema operativo multitarea, se trata al fin y al cabo de un programa, si bien tiene acceso a todos los recursos de la máquina y durante su ejecución se encarga de servir al resto de programas, a los que denominaremos tareas[14]. En un momento dado, suponiendo un procesador con una

[14]El término tarea se emplea como una abstracción de los conceptos de proceso e hilo, comúnmente empleados en los sistemas operativos.

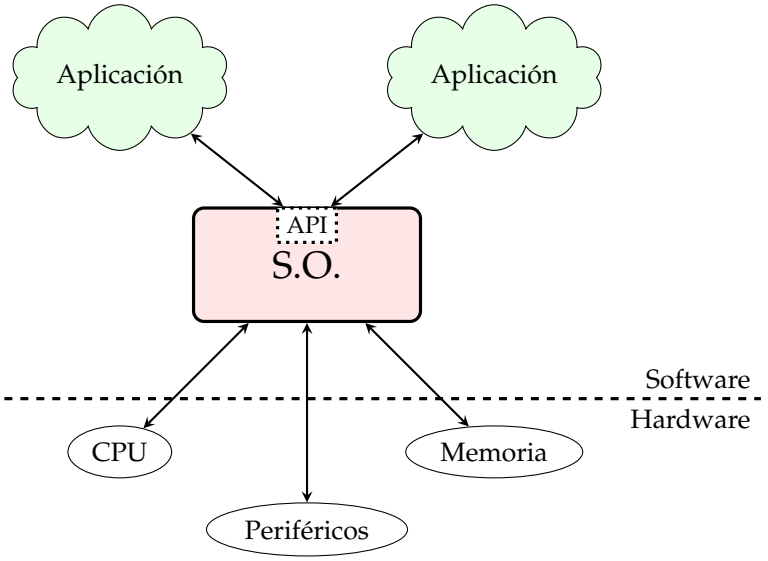


Figura 2.61: Relación entre el hardware, el sistema operativo y las aplicaciones

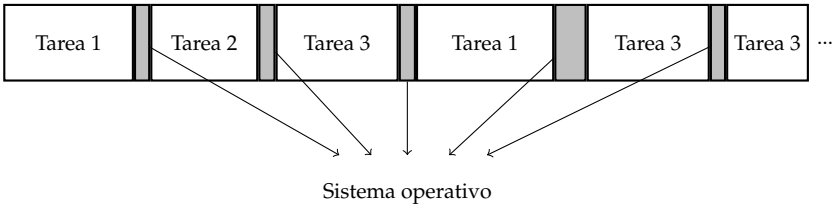


Figura 2.62: Cronograma de ejecución del SO y tres tareas

única CPU o núcleo, se estará ejecutando una tarea o el sistema operativo. La figura 2.62 muestra un ejemplo de ejecución de tres tareas y el sistema operativo. Como se puede observar, el tiempo de ejecución del sistema operativo es muy bajo en comparación con el de las tareas, tal como suele ocurrir habitualmente[15]. A lo largo de 50 milisegundos se ejecutan de dos a tres veces cada una de las tareas. Esto lo percibe el usuario como la ejecución simultánea de las tres tareas, aunque en realidad en cada instante se ejecuta una única tarea o el sistema operativo, dado que no es capaz de identificar los cambios de ejecución de una tarea a otra. El proceso de cambiar la tarea en ejecución por otra distinta recibe el nombre de cambio de contexto.

En la figura 2.62 se observa asimismo que para ejecutar una tarea diferente, esto es, realizar el cambio de contexto, es necesario transferir el control al sistema operativo, lo que implica pasar a ejecutar el sistema operativo. Un sistema operativo multitarea recibe el control de la máquina cuando ocurre una de estas tres situaciones:

[15]Una excepción a esto ocurre cuando el sistema operativo está ejecutando el código de un servicio complejo solicitado por una tarea.

1. Una tarea realiza una llamada a un servicio del sistema operativo.
2. Se produce una interrupción.
3. Se produce una excepción.

En cualquiera de estas situaciones se ejecuta un manejador (o rutina de servicio) que forma parte del sistema operativo. Antes de estudiar cada uno de los mecanismos de transferencia anteriores es conveniente tener en cuenta las siguientes consideraciones:

- Cada vez que se produce una llamada al sistema, una interrupción o una excepción, el manejador puede modificar el estado de las tareas. Por ejemplo, puede retirar el control de la CPU a una tarea para cedérselo a otra. Esto puede observarse en la figura 2.62.
- Las llamadas al sistema, interrupciones y excepciones pueden anidarse. Por ejemplo, puede generarse una excepción durante la ejecución de una instrucción del manejador de una interrupción, o interrumpirse el manejador de una interrupción para ejecutar el manejador de una interrupción más prioritaria que acaba de producirse.
- Cuando se trabaja con las CPU reales es necesario tener cuidado con la nomenclatura empleada, pues no hay un convenio común a todas ellas en la literatura técnica. En algunas CPU las llamadas al sistema, interrupciones y excepciones se engloban bajo el término genérico de excepciones, tal como se ha hecho en este texto en las secciones anteriores. Otras utilizan el término genérico *traps*. Incluso hay otras CPU en las cuales las llamadas a servicios se las conoce como interrupciones software, como por ejemplo en las CPU x86. En lo que resta del libro se empleará la nomenclatura de *llamadas al sistema*, *interrupciones* y *excepciones*.
- Es necesario disponer de un temporizador que genere una interrupción cada cierto periodo de tiempo prefijado, T. Esto asegura que el sistema operativo recupera el control de la máquina en el peor de los casos cada T unidades de tiempo. De esta forma, se evita que una tarea monopolice la CPU mientras no se produzcan excepciones y no haya interfaces de periféricos que generen interrupciones.

En los apartados siguientes se describen los tres mecanismos de transferencia de control al sistema operativo.

Llamada a los servicios del sistema operativo

Consiste en que una aplicación solicite a una rutina del sistema operativo que le preste algún servicio. Por ejemplo, cuando un programa desea abrir un fichero de disco llama a un servicio del sistema que le permite llevar a cabo esta tarea. Debe tenerse en cuenta que una aplicación no puede acceder directamente al disco; debe hacerlo siempre a través del sistema operativo.

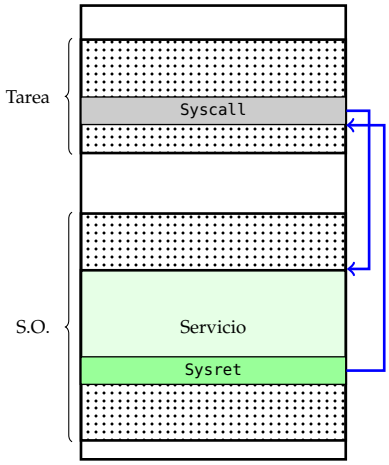


Figura 2.63: Llamada y retorno de un servicio del sistema operativo

Hay dos tipos de instrucciones relacionadas con la llamada a servicios del sistema operativo. Genéricamente, reciben el nombre de `Sysret` y `Syscall`, si bien su nombre real es dependiende de la arquitectura.

- `Syscall`. Instrucción de llamada a un servicio del sistema operativo a través de su API. Cuando una tarea ejecuta esta instrucción pasa el control al sistema operativo.
- `Sysret`. Instrucción de retorno de un servicio. El sistema operativo ejecuta esta instrucción justo antes de devolver el control a la tarea que solicitó el servicio (u otra tarea si se produce un cambio de contexto). Entre la ejecución de las instrucciones `Syscall` y `Sysret` el sistema operativo accede al espacio de memoria de la tarea. Por ejemplo, para el caso de la apertura de un fichero escribe en una estructura de datos de la tarea asociada al fichero si la apertura se ha completado con éxito o no. Después de ejecutar la instrucción `Sysret`, se ejecuta la instrucción siguiente a `Syscall`.

La figura 2.63 muestra cómo se produce la llamada y el retorno de un servicio del sistema operativo.

Interrupciones

Las interrupciones son un mecanismo hardware mediante el cual un periférico, a través de su interfaz, puede avisar a la CPU de que interrumpa momentáneamente la ejecución en curso para ejecutar un manejador asociado al mismo. Este manejador, o rutina de tratamiento, forma parte del sistema operativo. Al igual que ocurre con las llamadas a servicios del sistema operativo, la última instrucción del manejador es la instrucción `Sysret`, tras la cual la CPU continúa la ejecución interrumpida donde la había dejado.

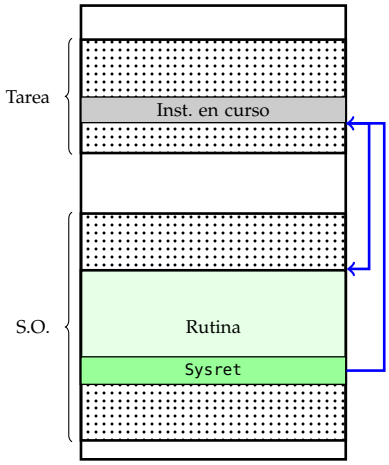


Figura 2.64: Flujo de ejecución durante una interrupción

Por ejemplo, ocurren interrupciones cada vez que se recibe un paquete de datos a través de una interfaz de red, cada vez que el usuario pulsa una tecla del teclado, o cada vez que expira el temporizador del sistema.

La figura 2.64 muestra el flujo de ejecución de la CPU durante una interrupción. Los pasos que ocurren son básicamente los siguientes:

- La interrupción se puede recibir en cualquier instante, pues la CPU desconoce a priori cuándo va a recibir una petición de interrupción.
- Se termina la ejecución de la instrucción en curso y se salta a la rutina de tratamiento, que forma parte del sistema operativo.
- Se ejecuta la rutina de tratamiento de la interrupción hasta que se alcanza la instrucción `Sysret`.
- A continuación de `Sysret` se ejecuta la instrucción siguiente a la que se estaba ejecutando en el momento en el que se produjo la interrupción.

Excepciones

El objetivo de las excepciones es pasar el control al sistema operativo cuando ocurre una situación anómala durante la ejecución de una instrucción. Al contrario de lo que sucedía con las interrupciones, no es un periférico el que origina la excepción, sino que es la propia CPU. Hay muchas situaciones que pueden dar lugar a una excepción. Por ejemplo, cuando se ejecuta una instrucción que intenta llevar a cabo una división por cero, o cuando se intenta ejecutar un código de instrucción inválido.

La figura 2.65 muestra el flujo de ejecución de la CPU durante una excepción. Los pasos que ocurren durante una excepción son básicamente los siguientes:

- Durante la ejecución de una instrucción se alcanza un estado de excepción.

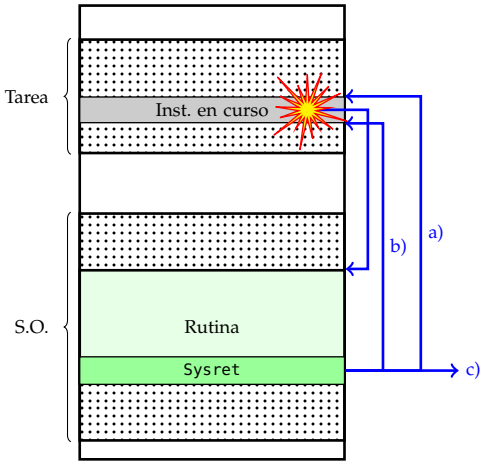


Figura 2.65: Flujo de ejecución durante una excepción

- Se transfiere el control a la rutina de tratamiento de la excepción, que forma parte del sistema operativo.
- Se ejecuta la rutina de tratamiento de la excepción hasta que se alcanza la instrucción `Sysret`.
- `Sysret` típicamente retorna a:
  a) la propia instrucción que causó la excepción, o
  b) la instrucción siguiente a la que causó la excepción, o
  c) no puede retornar.

2.7.2. Mecanismos de soporte a los SO

El soporte de la CPU a los sistemas operativos multitarea establece una serie de requerimientos funcionales mínimos que es necesario implementar en la arquitectura. A continuación se enumeran de forma justificada:

1. En el juego de instrucciones hay instrucciones privilegiadas que sólo debería poder ejecutar el sistema operativo. Se trata de instrucciones que modifican el modo de funcionamiento de la CPU o cambian su configuración. Un ejemplo claro es una instrucción que desactive las interrupciones. Si una tarea pudiese ejecutar esta instrucción privilegiada el sistema en su conjunto dejaría de responder a eventos externos, como las interrupciones solicitadas por la interfaz del teclado cuando se pulsa una tecla. En cambio, el sistema operativo debe poder ejecutar cualquier instrucción para realizar su labor de administración de los recursos de la máquina.

2. Debe protegerse la memoria del sistema operativo de lecturas y escrituras por parte de las tareas. El sistema operativo es un programa más que se encuentra en la memoria del computador. Si una tarea de usuario pudiese escribir en el área de memoria asociada al sistema operativo podría modificar su funcionamiento, tomar el control de la máquina o incluso provocar un mal funcionamiento. Si una tarea pudiese leer la memoria del sistema operativo podría acceder a información privada de otro usuario o llevar a cabo lecturas sobre las interfaces de los periféricos, mapeadas en el área de memoria del sistema operativo.

3. Debe protegerse la memoria de una tarea del acceso por parte de otra tarea. Una tarea no debe tener acceso al área de memoria de otra tarea, pues en ese caso, por ejemplo, la tarea de un usuario podría acceder a datos privados que forman parte de la tarea de otro usuario. Lógicamente tampoco debe permitirse la escritura en la memoria de otra tarea, pues puede modificar el funcionamiento de esta última.

4. Cada vez que se produce una interrupción, llamada a servicio del sistema operativo o excepción, la CPU debe proporcionar mecanismos que permitan guardar el estado de ejecución. De esta forma se podrá proseguir la ejecución donde se había dejado una vez termine la rutina de tratamiento.

2.7.3. Soporte a los SO multitarea en MIPS64

Una vez se han presentado las necesidades básicas a cubrir por una arquitectura para dar soporte a los sistemas operativos multitarea se proporciona un ejemplo en una arquitectura real, en este caso la arquitectura MIPS64. Como se verá, aparecerán nuevas instrucciones y registros que hasta ahora no habían sido necesarios.

Niveles de privilegio

El nivel (o modo) de privilegio de ejecución, o anillo de privilegio, es el estado de la CPU que establece qué instrucciones puede y no puede ejecutar. El nivel más privilegiado suele asociarse al sistema operativo, mientras que el menos privilegiado corresponde a las tareas de los usuarios. En el nivel privilegiado se pueden ejecutar todas las instrucciones del juego de instrucciones, mientras que en el nivel menos privilegiado sólo puede ejecutarse un subconjunto del juego de instrucciones.

El nivel de privilegio de una CPU está definido por uno o más bits de control. En el caso de la arquitectura MIPS64 hay fundamentalmente dos niveles de privilegio:

- Usuario. Nivel en el cual se ejecutan las tareas.
- *Kernel*. Nivel en el cual se ejecuta el sistema operativo.

El nivel de privilegio está definido por los bits `UM`, `EXL` y `ERL` del registro de estado. La figura 2.66 muestra a nivel de referencia estos bits en el registro de estado[16]. En

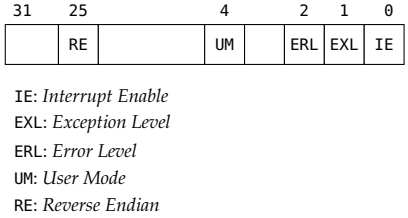


Figura 2.66: Algunos bits del registro de estado de la arquitectura MIPS64

modo usuario se tiene UM=1, EXL=0 y ERL=0. En modo *kernel* los bits pasan a ser UM=0, EXL=1 y ERL=1.

Una de las instrucciones privilegiadas en la arquitectura MIPS64 es la instrucción `di`, que desactiva las interrupciones. Esta instrucción puede ejecutarse en modo *kernel*, pero cuando trata de ejecutarse en modo usuario se produce una excepción.

La microarquitectura cambia el estado de los bits UM, EXL y ERL para pasar del modo usuario al modo kernel cuando se produce una interrupción, llamada a servicio, o excepción, durante la ejecución de una tarea. Para realizar una llamada a servicio del sistema operativo se utiliza la instrucción `syscall`.

Paralelamente, la instrucción que se ejecuta para finalizar la rutina de tratamiento de una interrupción, excepción o llamada a servicio es `eret`. Esta instrucción modifica el valor de los bits UM, EXL y ERL para conmutar la CPU al modo usuario.

Protección de memoria

Para proteger la memoria del sistema operativo de accesos por parte de las tareas, la arquitectura MIPS64 le reserva un rango de direcciones específico. Cualquier dirección en el rango 4000 0000 0000 0000h a FFFF FFFF FFFF FFFFh es accesible sólo en modo *kernel*, esto es, solo por el sistema operativo. Ubicando el sistema operativo en ese rango de direcciones está protegido de cualquier acceso por parte de las tareas. Las tareas tienen asociado el rango 0000 0000 0000 0000h a 3FFF FFFF FFFF FFFFh. Cuando una tarea intenta acceder a una dirección de memoria fuera de este rango, ya sea con una instrucción de carga o almacenamiento, se genera una excepción.

Para proteger la memoria de una tarea del acceso por parte de otras tareas las CPU MIPS64 incorporan una unidad de gestión de memoria que implementa un sistema de memoria virtual paginada. La memoria virtual paginada permite asociar a cada tarea rangos de direcciones exclusivos que no se solapan con los de otras tareas. Cuando una tarea intenta acceder a una dirección de un rango de direcciones que no le pertenecen se genera una excepción. El estudio de la memoria virtual paginada forma parte del capítulo 3.

[16] Se ha aprovechado la figura para mostrar el bit de habilitación de las interrupciones, IE, y el bit RE, que establece si la CPU trabaja en modo *little endian* o *big endian*.

Salvaguarda del estado de ejecución

Cuando se produce una interrupción, una llamada a servicio o una excepción es necesario guardar el estado de ejecución para poder ser retomado más adelante y continuar la ejecución. En el caso de la arquitectura MIPS64 hay dos registros relacionados, el `EPC` (también denominado *Exception PC*) y el *cause register*.

El registro `EPC` almacena la dirección de la instrucción a ejecutar cuando la rutina de tratamiento termine. Puede ser la dirección de la instrucción siguiente a la que causó la interrupción, llamada a servicio o excepción, o también en algunas excepciones la dirección de la misma instrucción que la causó.

El registro *cause register* almacena un valor que identifica la interrupción, la llamada a servicio o la causa de la excepción.

Cuando se produce una interrupción, llamada a servicio o excepción se desactivan las interrupciones para evitar que pueda llegar una interrupción y se sobrescriban los registros `EPC` y *cause register*. El tiempo durante el cual las interrupciones permanecen desactivadas debe ser tan corto como sea posible, por lo que la primera labor que debe realizar una rutina de servicio, que se ejecuta en modo *kernel*, es copiar a un lugar seguro de la memoria el valor de los registros `EPC` y *cause register* y volver a habilitar las interrupciones a continuación.

El retorno de la rutina de servicio se lleva a cabo escribiendo en el registro `EPC` la dirección de retorno previamente guardada y a continuación ejecutando la instrucción `eret`. A partir de ese momento, la CPU vuelve al modo usuario y retoma la ejecución en la instrucción cuya dirección se había salvado en el registro `EPC`.

2.8. Soporte a la virtualización

La virtualización es una técnica que permite ejecutar varios sistemas operativos con sus aplicaciones sobre la misma máquina. Tiene innumerables ventajas y se emplea tanto en servidores como en computadores de sobremesa. Siguiendo la misma línea del apartado anterior, en primer lugar se realiza una pequeña introducción a la virtualización, para finalmente ejemplificar la virtualización de la CPU sobre una arquitectura real, en este caso la arquitectura x86.

2.8.1. Introducción a la virtualización

La virtualización es un concepto general en el cual se presenta una entidad ficticia (virtual) como si fuese la real. En informática, se utiliza en muchos ámbitos; el más cercano a la arquitectura de computadores es lo que habitualmente se denomina virtualización de hardware, en la cual se crean máquinas virtuales, que son duplicados de la máquina real, o física, y sobre las que se puede instalar un sistema operativo como si se tratase de la máquina física. Esto tiene muchas utilidades, como ejecutar distintos sistemas operativos en el mismo hardware a la vez, aislar más las aplicaciones entre sí, o migrar fácilmente sistemas enteros (sistema operativo más aplicaciones) entre máquinas físicas distintas.

En un primer momento, las arquitecturas de computadores se diseñaban suponiendo que sólo había que dar soporte a un sistema operativo que administraría los recursos hardware entre distintas aplicaciones. Los sistemas operativos, a su vez, se diseñaban suponiendo que estaban gestionando un hardware que no compartían con otros sistemas operativos. Debido a esto, no era posible ejecutar dos sistemas operativos distintos a la vez sobre una máquina física única. Por lo tanto, fue necesario añadir una capa de abstracción que proporcionase la virtualización.

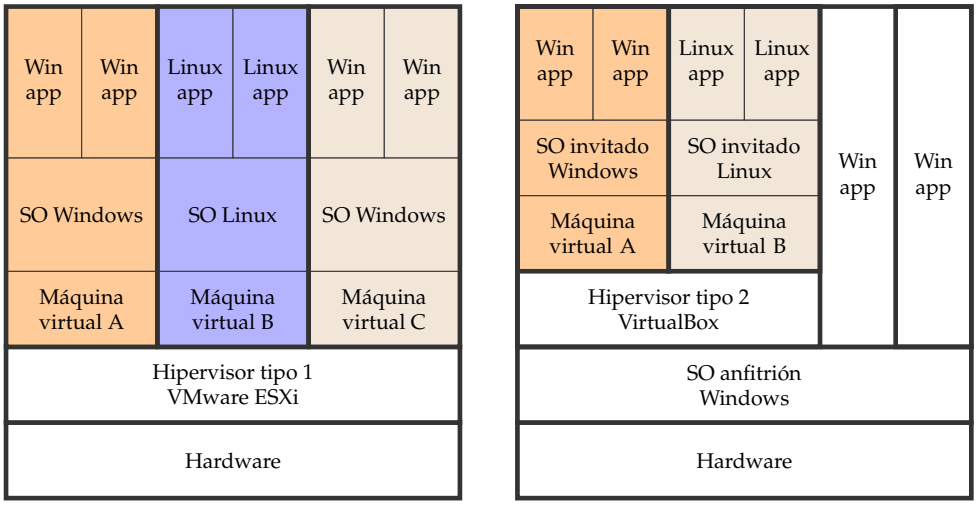
Se suele denominar hipervisor o monitor de máquinas virtuales (*Virtual Machine Monitor*, VMM) al componente software que interacciona directamente con el hardware real y presenta un hardware virtual para los sistemas operativos de las máquinas virtuales, multiplexando el hardware (real) entre varias máquinas virtuales. Hay dos tipos de hipervisor (ver figura 2.67):

- Tipo 1: el hipervisor es una capa de software pequeña que se ejecuta directamente sobre el hardware real y tiene como único cometido administrar los recursos entre las distintas máquinas virtuales; no se dedica a ejecutar otras aplicaciones.
- Tipo 2: el hipervisor es una aplicación que se ejecuta sobre un sistema operativo al que se denomina «anfitrión» (*host*), que gestiona el hardware real y puede ejecutar otras aplicaciones. El hipervisor se encarga de proporcionar las máquinas virtuales en las que se ejecutarán los otros sistemas operativos, a los que se denomina «invitados» (*guests*).

Los hipervisores de tipo 2 proporcionan un rendimiento inferior a los de tipo 1, pues el sistema operativo anfitrión representa una capa software adicional. Sin embargo, al contrario de lo que ocurre con los hipervisores de tipo 1 que se ejecutan sobre un hardware muy concreto, los hipervisores de tipo 2 tienen la ventaja de ejecutarse sobre casi cualquier hardware, pues el sistema operativo anfitrión proporciona el soporte necesario.

Aunque las versiones iniciales de muchas arquitecturas no soportaban la virtualización, aún en ese caso se podía construir un VMM menos eficiente, ejecutando los sistemas operativos invitados con el nivel de privilegio de las tareas y utilizando la técnica *trap-and-emulate*. De esta forma, cada vez que un sistema operativo invitado ejecutaba una instrucción privilegiada se producía una excepción que capturaba el hipervisor, emulando el comportamiento de la instrucción sobre el hardware real y devolviendo a continuación el control al sistema operativo invitado. Aun empleando la técnica *trap-and-emulate* había instrucciones cuya ejecución en modo no privilegiado no provocaba excepciones, pero cuyo funcionamiento cambiaba cuando se ejecutaban en modo privilegiado y en modo no privilegiado. A este tipo de instrucciones se las denomina «sensibles». Para solucionar este último problema fue necesario emplear además la técnica de traducción binaria que traducía las instrucciones «sensibles» en otras «no sensibles» antes de ser ejecutadas por la CPU. El principal problema de las técnicas de *trap-and-emulate* y traducción binaria es la considerable reducción de rendimiento en la ejecución de la máquina virtual.

Otra alternativa de virtualización más eficiente que el empleo de las técnicas de *trap-and-emulate* y traducción binaria es la paravirtualización. Básicamente consiste



(a) Tipo 1 (b) Tipo 2

Figura 2.67: Tipos de hipervisor

en modificar el sistema operativo anfitrión sustituyendo sus fragmentos de código privilegiados por llamadas a servicios del hipervisor. Por lo tanto, se establece una relación entre el hipervisor y el sistema operativo invitado análoga a la que existe entre un sistema operativo y una tarea de usuario del mismo. El mayor inconveniente de la paravirtualización es que exige modificar el sistema operativo invitado, lo cual es un trabajo ingente y, además, si no está disponible el código fuente, es inviable.

En la actualidad las CPU de propósito general proporcionan un soporte hardware a la virtualización de tal forma que ya no son necesarias las técnicas de *trap-and-emulate* y traducción binaria, lo que acerca considerablemente el rendimiento de la máquina virtual al de la máquina real. Aunque en este caso la paravirtualización no es necesaria, algunos sistemas operativos implementan una versión reducida de la misma, de tal forma que durante su ejecución pueden determinar si se están ejecutando directamente sobre el hardware o sobre un hipervisor paravirtualizado y optimizar su funcionamiento. Por ejemplo, el hipervisor de VirtualBox proporciona la opción de paravirtualización KVM que permite optimizar el rendimiento de máquinas virtuales que ejecutan versiones modernas de Linux.

2.8.2. Soporte a la virtualización en la arquitectura x86

Las versiones modernas de la arquitectura x86 han introducido el soporte hardware a la virtualización en forma de nuevas instrucciones y modos de funcionamiento. Las primeras mejoras aparecen englobadas bajo las extensiones de nombre VT-x, en el caso de la compañía Intel, y AMD-V, en el caso de la compañía AMD. Ambas extensiones son incompatibles, por lo que habitualmente la implementación de un hipervisor sobre la arquitectura x86 exige identificar previamente la extensión soportada. Básicamente estas extensiones introducen un nivel de privilegio adicional,

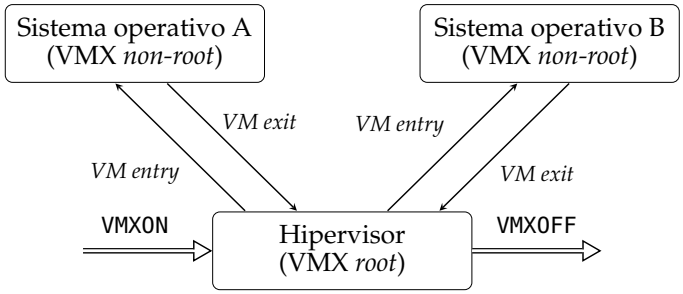


Figura 2.68: Transferencia de control con la extensión VT-x

más privilegiado que el nivel de privilegio asociado a los sistemas operativos, así como un conjunto de nuevas instrucciones y estructuras de datos relacionadas con la virtualización.

Se ilustrará como ejemplo el funcionamiento de la virtualización empleando la extensión VT-x de Intel. Cuando la arquitectura incluye esta extensión, la CPU implementa un nuevo modo de trabajo asociado a la virtualización, denominado VMX. La CPU entra en este modo ejecutando una nueva instrucción, la instrucción `VMXON`, y sale del mismo empleando otra nueva instrucción, la instrucción `VMXOFF`. Una vez la CPU se encuentra en el modo VMX puede encontrarse en dos submodos de operación:

- VMX *root*. Modo en el cual se ejecuta el hipervisor, con acceso al juego completo de instrucciones. En los sistemas que emplean hipervisores de tipo 2, el sistema operativo anfitrión también se ejecuta con este nivel de privilegio.
- VMX *non-root*. Modo en el cual se ejecutan los sistemas operativos de todas las máquinas virtuales, con acceso a un subconjunto del juego de instrucciones (más amplio que el de las tareas de usuario).

La figura 2.68 muestra esquemáticamente las transferencias de control empleando las nuevas instrucciones incluidas en la extensión VT-x.

En el modo VMX *root*, la CPU dispone de la instrucción `VMLAUNCH` para arrancar una máquina virtual, transfiriendo el control al sistema operativo correspondiente. La transferencia de control del hipervisor al sistema operativo se denomina VM *entry*. Una vez se ejecuta la instrucción `VMLAUNCH` la CPU entra en el modo VMX *non-root*.

Una vez toma el control el sistema operativo se ejecuta de la forma habitual hasta que ejecuta ciertas instrucciones. En ese momento se produce una transferencia de control del sistema operativo al hipervisor que se denomina VM *exit*. Una vez el hipervisor retoma el control, puede tratar adecuadamente la instrucción que provocó el VM *exit* empleando la técnica clásica de *trap-and-emulate*. A continuación el sistema operativo recupera el control de la máquina cuando el hipervisor ejecuta la instrucción `VMRESUME` que da lugar a un nuevo tipo de VM *entry*.

Las extensiones VT-x y AMD-V mejoran sensiblemente el rendimiento de las máquinas virtuales, mejorando la gestión de la CPU. Sin embargo existen otras elementos del computador que sufren problemas de rendimiento derivados de la virtualización que no son paliados con estas extensiones. Este es el caso de la memoria virtual y la entrada salida. En el caso de la memoria virtual las arquitecturas implementan nuevas extensiones, las cuales se muestran en el apartado 3.5, mientras que en el caso de la entrada/salida se muestran en el apartado 4.4.

Capítulo 3

La jerarquía de memoria

Este capítulo muestra cómo el sistema de memoria del computador se construye empleando diferentes tecnologías de almacenamiento, tratando de obtener un sistema de memoria rápido, barato y de gran capacidad. Actualmente, esto se consigue combinado tres tecnologías de memoria diferentes:

- Memoria SRAM. Empleada en la memoria caché.
- Memoria DRAM. Empleada en la memoria principal.
- Almacenamiento secundario. Gestionado por el mecanismo de la memoria virtual.

3.1. Introducción

Tras la CPU, el segundo elemento en importancia en el computador es el sistema de memoria. Su misión es servir como almacén para las instrucciones y datos que constituyen los programas. Por tanto, interesa que sea grande para poder almacenar muchos programas, muy grandes y con muchos datos. Además, interesa que sea rápida, ya que la CPU accede a memoria al menos una vez en cada instrucción: en la etapa de búsqueda del código de instrucción. Si además se trata de una instrucción de carga o almacenamiento, debe realizar un segundo acceso, esta vez para leer o escribir un dato respectivamente. En general, la CPU es más rápida que la memoria, por lo que las instrucciones que acceden a memoria pueden sufrir detenciones en el camino de datos esperando por memoria.

El problema de la rapidez de la memoria se ha ido acentuando con el paso del tiempo y el desarrollo tecnológico seguido por los distintos elementos del computador. La velocidad de la CPU ha experimentado un incremento muy importante, como también ha ocurrido con el número de núcleos de los procesadores. La memoria, en cambio, ha evolucionado principalmente en densidad, es decir, en capacidad

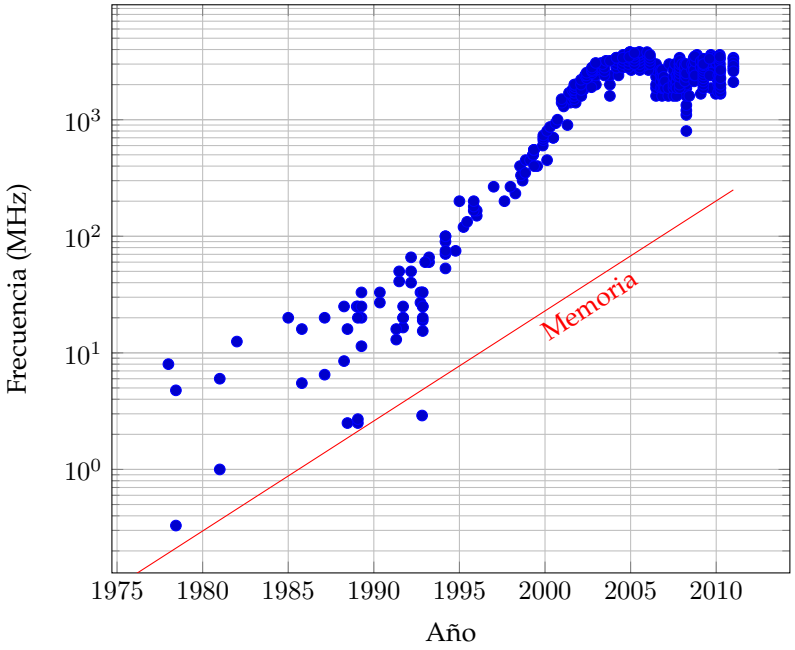


Figura 3.1: Evolución en velocidad de la CPU y la memoria

de almacenamiento, pero en menor medida en velocidad. La figura 3.1 muestra la evolución en velocidad seguida tanto por la CPU como por la memoria[1].

El coste de la espera de la CPU por la memoria es cada vez mayor, más si cabe con la tendencia a incluir cada vez un mayor número de núcleos dentro de los procesadores. Esto hace que el rendimiento efectivo de la CPU baje por la necesidad de esperar por la memoria.

A modo de aproximación, considérese una CPU segmentada que trabaja a una frecuencia de 1 GHz. Idealmente, esta CPU es capaz de ejecutar instrucciones con un CPI de 1, esto es, ejecuta 1000 MIPS. Supóngase también que el CPI real de esta CPU es 1.3 por la necesidad de esperar por memoria en las instrucciones de carga y almacenamiento (no se tienen en cuenta las lecturas de los códigos de instrucción de la memoria de instrucciones). En ese caso, la CPU real sería capaz de ejecutar únicamente 769.23 MIPS.

Para solucionar el problema de tamaño y velocidad se enumeran a continuación las opciones disponibles para construir una memoria:

- Tecnología de memoria RAM estática (SRAM). Su celda básica es un biestable.
- Tecnología de memoria RAM dinámica (DRAM). Su celda básica es un condensador y requiere refresco.
- Almacenamiento secundario (discos magnéticos y unidades de estado sólido).

[1] La figura es una mera aproximación, pues medir la velocidad en MHz es un poco engañoso. Las CPU pueden tener diferentes rendimientos a pesar de trabajar a la misma frecuencia de reloj.

De estas tres alternativas, la memoria SRAM es la más rápida de todas. Su tiempo de acceso llega a ser próximo al periodo del reloj de la CPU. La memoria DRAM es del orden de 5 veces más lenta que la SRAM. El almacenamiento magnético es extremadamente lento, hasta 10 millones de veces más lento. Por último, el almacenamiento basado en memoria *flash* (unidades SSD) es en torno a 100 veces más rapido que el almacenamiento magnético, pero sus tiempos de acceso todavía están lejos de los de las memorias SRAM y DRAM.

Atendiendo al criterio del tiempo de acceso no existe duda: la memoria del computador debería construirse con memoria SRAM. Sin embargo, existe una restricción adicional que aún no se ha mencionado: el coste debe ser limitado. Este viene determinado tanto por el tamaño físico de la celda básica (cantidad de material necesario para almacenar un bit), como por la energía que consume. Atendiendo al factor del tamaño físico el panorama se transforma tal como sigue:

- La tecnología SRAM tiene el tamaño de celda más grande; requiere al menos 6 transistores para construir la celda y es la que más energía consume.
- En la tecnología DRAM se requiere sólo un transistor para construir la celda básica e implica un menor consumo de energía.
- En la tecnología de almacenamiento magnético la celda básica está formada por unos pocos elementos de material magnético, permite unas densidades de almacenamiento muy grandes y son pasivos, de decir, no consumen energía para mantener el almacenamiento. Por su parte, las unidades de estado sólido se basan en tecnología *flash*, que también es pasiva.

Si en un gráfico en escala logarítmica se representan los dos criterios anteriores, en abscisas la rapidez de la memoria, expresada como latencia, y en ordenadas el coste, expresado en € por MiB, se obtiene el gráfico mostrado en la figura 3.2.

Para tener una idea de los valores reales, considérense las características de elementos representativos de las tecnologías vistas para el año 2017.

- Tecnología SRAM. La memoria SRAM, tal como se utiliza actualmente, viene incorporada en el procesador y su coste solo se podría evaluar por la diferencia de costes de dos procesadores idénticos salvo en el tamaño de memoria SRAM. Se consideran dos procesadores Intel Core i3 4170T y 4370T con 3 MiB y 4 MiB de caché (SRAM) respectivamente. La velocidad de acceso es del orden de 1 nanosegundo. Los costes son aproximadamente de 135 € y 163 € para cada procesador respectivamente. Esto hace que pueda valorarse el MiB de SRAM adicional en 28 €.
- Tecnología DRAM.- Se considera el módulo de memoria Kingston modelo HX421C14FB/8, cuya capacidad es de 8 GiB, con un tiempo de acceso de 6.6 nanosegundos y un coste aproximado de 75 €, lo que resulta en un coste de 0.0092 €/MiB.
- Almacenamiento magnético.- Disco Western Digital modelo WD10EZEX (Serial ATA/600) de 1 TB. Su tiempo medio de acceso en lectura es de 8 milisegundos y el coste es aproximadamente de 50 €, lo que resulta en un coste de 0.00005 €/MB.

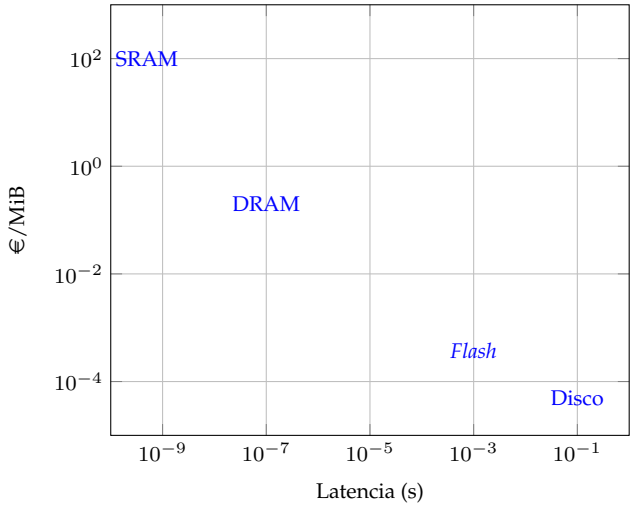


Figura 3.2: Relación coste y velocidad de las tecnologías de memoria

Considerando los precios anteriores, si se deseara construir una memoria de 16 GiB para un computador, existirían las siguientes alternativas:

- Si se construye con tecnología SRAM, se consigue una memoria extremadamente rápida con un tiempo de acceso de 1 nanosegundo y con un coste aproximado de 459 000 €.
- En el otro extremo está el caso del disco, por solo 50 € se podría obtener un disco de 1 TB o más. Sin embargo, el tiempo de acceso sería del orden de 8 milisegundos; 8 millones de veces más lento que la SRAM.
- Si la memoria se construye con memoria DRAM, el coste aproximado sería de 150 € y sería aproximadamente 6 veces más lenta que la memoria SRAM.

Se puede concluir que no existe ninguna tecnología de memoria que por sí sola satisfaga todos los requisitos deseables para la memoria. La tecnología SRAM es la más rápida, pero enormemente costosa; el almacenamiento magnético es muy económico y tiene una enorme capacidad, pero es extremadamente lento (lo mismo ocurre con el almacenamiento basado en memoria *flash*); la tecnología DRAM se encuentra en el punto intermedio, no es suficientemente rápida, ni suficientemente grande, pero tampoco excesivamente costosa. ¿Cómo construir entonces la memoria del computador? La solución pasa por combinar las tecnologías anteriores, tomando lo mejor de cada una en lo que se conoce como jerarquía de memoria.

3.2. Concepto de jerarquía de memoria

En la construcción del sistema de memoria de un computador se persiguen, entre otros, tres objetivos:

- Alta velocidad.
- Alta capacidad.
- Bajo coste.

El problema es que tal como se ha visto, estos objetivos son difícilmente alcanzables con una única tecnología. La solución consiste en combinar memorias rápidas y pequeñas con memorias lentas y grandes, de tal forma que:

- Las memorias rápidas y pequeñas contengan los datos con mayor probabilidad de acceso.
- Las memorias lentas y grandes contengan los datos con menor probabilidad de acceso.

Con esto se consigue que la capacidad del sistema de memoria sea elevada, pues contiene memorias grandes. Además, en la mayor parte de los casos el sistema de memoria será rápido, pues habitualmente se accederá a las memorias rápidas.

El tamaño resultante de combinar memorias de esta forma no es acumulativo. Las memorias rápidas y pequeñas contienen una copia de una parte de los datos almacenados en las memorias grandes, por lo que desde el punto de vista de la CPU, el tamaño del sistema de memoria viene definido por el tamaño de la memoria más grande.

Para poder llevar a la práctica la solución anterior es necesario que se cumplan dos condiciones:

- Conocer a priori los datos con mayor probabilidad de acceso futuro para así poder almacenarlos en las memorias pequeñas y rápidas.
- Que estos datos sean pocos y con una gran probabilidad de ser accedidos en el futuro. De esta forma cabrán en las memorias pequeñas y rápidas y además servirán la mayor parte de los accesos futuros, obteniéndose un sistema de memoria rápido en la mayor parte de los casos.

El cumplimiento de las condiciones anteriores es posible gracias al principio de localidad observado en la ejecución de programas por parte de cualquier CPU. Este principio se divide en dos:

- Principio de localidad espacial. Si en un momento dado se accede a una dirección de memoria, es probable que poco después se acceda a una dirección de memoria cercana. Un ejemplo de localidad espacial es el acceso al código durante la ejecución secuencial de un programa. Un ejemplo de localidad espacial en el acceso a datos se tiene en el acceso a los datos de un vector.

- Principio de localidad temporal. Si en un momento dado se accede a una dirección de memoria, es probable que poco después se acceda a esa misma dirección de memoria. Un ejemplo de localidad temporal en el acceso a código se encuentra en la ejecución de bucles, donde la misma instrucción se accede frecuentemente para ejecutarla varias veces. Un ejemplo de localidad temporal en el acceso a datos es el acceso a un contador de iteraciones durante la ejecución de un bucle.

Un símil de la jerarquía de memoria es el de un negocio (el sistema de memoria) destinado a la venta. Este negocio está formado por una tienda pequeña (memoria pequeña y rápida) en el centro de una ciudad en la que se atiende personalmente al cliente (la CPU) y de un gran almacén en las afueras (memoria grande y lenta). Si se conocen los gustos del cliente (principio de localidad) la tienda puede almacenar los artículos (posiciones de memoria) que probablemente solicitará el cliente y servirlos rápidamente. Además, el negocio dispone de un gran número de artículos, pues tiene un gran almacén[2].

La combinación de las diferentes tecnologías de memoria se hace en niveles de diferente capacidad y velocidad, formando lo que se conoce como jerarquía de memoria. La figura 3.3 muestra una jerarquía de memoria de tres niveles, que servirá de referencia a lo largo del presente capítulo. Los valores numéricos son sólo orientativos pues dependen de la tecnología. Por ejemplo, una CPU MIPS64 puede acceder a bytes, medias palabras, palabras o dobles palabras. En el caso de utilizar emisión múltiple, el ancho del canal de comunicación entre CPU y caché debe permitir leer simultáneamente tantas instrucciones como determine el ancho de emisión de la CPU. Además, con canales de comunicación con la caché anchos también pueden implementarse técnicas de *pre-fetching*, donde se leen varias instrucciones al mismo tiempo para aprovechar el citado principio de localidad espacial.

La memoria caché se construye empleando memorias SRAM, las cuales son muy rápidas pero de baja capacidad. En la práctica, el nivel de caché puede a su vez dividirse en subniveles.

La memoria principal se construye empleando memorias DRAM, que constituyen memorias de velocidad media y capacidad media.

El nivel más alejado de la CPU de la jerarquía se construye habitualmente empleando discos magnéticos o unidades de estado sólido, que son lentos pero de gran capacidad de almacenamiento. Para ello no se emplea el dispositivo completo, sino parte del mismo.

El principio de funcionamiento de la jerarquía de memoria es muy simple. Cuando la CPU desea leer el contenido de una dirección del sistema de memoria, accede al nivel de la memoria caché. Si el contenido de esa dirección se encuentra dentro de la memoria caché (acierto de caché), la CPU accede a ella de forma muy rápida. Esta es la situación más habitual, pues la mayor parte de los accesos de la CPU al sistema de memoria son servidos inmediatamente por la caché.

Si por el contrario el contenido de esta dirección de memoria no está dentro de la memoria caché (fallo de caché), se transfieren un conjunto de posiciones de memoria

[2]La solución ideal desde el punto de vista del rendimiento sería tener una tienda en el centro del tamaño del almacén, pero eso sería muy caro; lo mismo que sucede con las memorias.

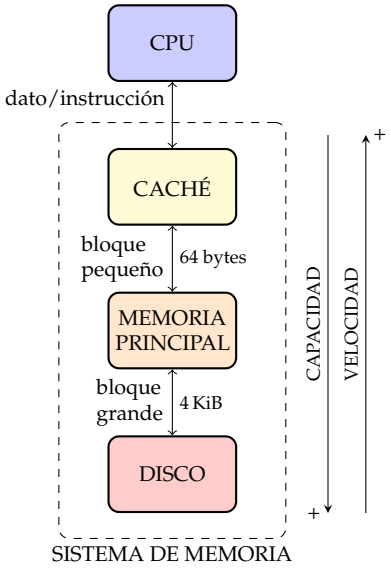


Figura 3.3: Jerarquía de memoria de tres niveles

consecutivas que incluyen la dirección deseada (o direcciones si se accede a más de un byte) desde la memoria principal a la memoria caché. A este conjunto de posiciones de memoria consecutivas se le denomina bloque. Una vez se ha realizado la transferencia, la CPU accede a la caché. En general, cada vez que se produce un fallo en un nivel N del sistema de memoria, se copia del nivel N+1 al nivel N el bloque que contiene la posición deseada y a continuación se realiza el acceso.

Debe notarse que la unidad de transferencia de información entre dos niveles contiguos de la jerarquía es el bloque. Además, el tamaño del bloque no tiene por qué ser el mismo en todos los puntos de la jerarquía. Por ejemplo, el tamaño del bloque en las transferencias del disco a la memoria principal es mucho mayor que el tamaño del bloque entre la memoria principal y la memoria caché.

Es posible que se encadenen fallos en diferentes niveles. Por ejemplo, si la CPU desea leer un dato que no está en la memoria caché, se intenta copiar el bloque de memoria principal que lo contiene a la caché. Si este bloque tampoco está en la memoria principal entonces se produce un fallo de la memoria principal, por lo que el bloque (aún mayor) de disco que lo contiene se copia a la memoria principal, y a continuación el bloque de memoria principal que contiene la posición deseada se copia a la memoria caché.

A continuación se muestra un ejemplo de cómo la jerarquía de memoria es capaz de proporcionar simultáneamente bajo coste, alta capacidad y alta velocidad. Para ello, se construye un sistema de memoria de tamaño 6 GiB empleando tres tecnologías, con los tamaños indicados:

- 4 MiB de memoria SRAM empleados como memoria caché, con un tiempo de acceso $t_c = 1$ nanosegundo.

- 3 GiB de memoria DRAM empleados como memoria principal, con un tiempo de acceso $t_p = 5$ nanosegundos por byte.
- 6 GiB de disco con un tiempo de acceso $t_c = 11$ milisegundos, independientemente del tamaño del bloque de disco[3].

El tamaño del bloque entre la caché y la memoria principal, denotado B_{cp}, es de 64 bytes.

La tasa de aciertos en el acceso de la CPU a la memoria caché, denotada por A_c, es de 0.999, o lo que es lo mismo, 99.9 %. La tasa de aciertos en el acceso a la memoria principal, denotada por A_p, es de 0.9999999, o lo que es lo mismo, 99.99999 %.

El coste de la memoria SRAM es de 28 € por MiB, el de la memoria DRAM 0.0092 € por MiB y el del disco 0.00005 € por MB.

1. ¿Cuál es el coste del sistema de memoria presentado?
2. ¿Cuál es el tiempo medio de lectura, denotado por tr_{cpd}?

El coste se obtiene fácilmente teniendo en cuenta la cantidad de memoria de cada tipo:

$$\text{coste} = 4 \times 28 + 0.0092 \times 3072 + 0.00005 \times 6144 \times \frac{2^{20}}{10^6} = 140.58\,€$$

En cuanto al tiempo medio de lectura del sistema de memoria, este se obtiene teniendo en cuenta que hay tres posibilidades: el dato está en la memoria caché, el dato está en memoria principal y no en caché, o el dato está solo en el disco. No se consideran los casos en los que no existe el dato o es inaccesible.

En el primer caso, si el dato está en la memoria caché se producirá un acierto y el tiempo de lectura será igual al tiempo medio de acceso de la memoria caché.

En el segundo caso, el dato está en la memoria principal pero no en la caché, por lo que se produce un fallo de caché. En consecuencia, se trae un bloque completo desde la memoria principal a la caché. Si bien el tiempo de transferencia del bloque desde memoria principal a la caché depende del ancho del canal de comunicaciones entre ambas memorias, entre otras cosas, por simplicidad se estimará como proporcional al número de bytes de memoria a transferir.

En el último caso, el dato se encontraría únicamente en el disco, por lo que se produciría un fallo en la caché y otro en memoria principal. Tras esto, se copiaría un bloque desde el disco hacia memoria principal, y a continuación se copiaría un bloque desde memoria principal a la caché. Por simplicidad, se supone que el movimiento de datos desde el disco a memoria principal puede realizarse de forma simultánea con el movimiento del bloque de memoria principal a la caché. De igual forma, también se supone que el coste de un fallo de caché es únicamente el tiempo necesario para copiar el bloque desde memoria principal, sin incluir la posterior lectura del dato en la caché que provocaría, esta vez sí, un acierto de caché.

[3] El tamaño de bloque en la práctica es del orden de 4 KiB. Resulta más costoso acceder al primer byte que a los demás. De hecho, este tiempo es el que mayormente proporciona el tiempo de acceso del bloque.

A continuación se muestran las expresiones matemáticas relacionadas con cada uno de los anteriores casos:

$$tr_{cpd} = \begin{cases} \text{acierto de caché} & \Rightarrow A_c \cdot t_c \\ \text{fallo de caché y acierto de M.P.} & \Rightarrow (1 - A_c) \times A_p \times (t_p \cdot B_{cp}) \\ \text{fallo de caché y fallo de M.P.} & \Rightarrow (1 - A_c) \times (1 - A_p) \times t_d \end{cases}$$

O lo que es lo mismo:

$$tr_{cpd} = A_c \cdot t_c + (1 - A_c) \times [A_p \cdot t_p \cdot B_{cp} + (1 - A_p) \cdot t_d]$$

donde:

tr_{cpd} es el tiempo medio de lectura del sistema de memoria.

A_c es la tasa de aciertos de caché.

t_c es el tiempo de acceso a la caché.

A_p es la tasa de aciertos de memoria principal una vez se ha producido un fallo de caché.

t_p es el tiempo necesario para acceder a un byte de la memoria principal.

B_{cp} es el número de bytes a transferir desde memoria principal hacia la caché cuando se produce un fallo de caché.

$t_p \times B_{cp}$ es el tiempo necesario para transferir un bloque desde memoria principal hacia memoria caché.

t_d es el tiempo necesario para llevar un bloque del disco a la memoria principal.

Sustituyendo el valor de cada una de las variables en la expresión anterior resulta $tr_{cpd} = 1.32\,\text{ns}$.

Como se puede observar, se ha obtenido un sistema de memoria con un tiempo medio de lectura del orden de magnitud del de la memoria caché, con una capacidad de 6 GiB y con un coste razonable.

La clave para la obtención de un tiempo medio de lectura cercano al de la memoria caché reside en el principio de localidad, pues es el que asegura unos porcentajes de acierto altos en el acceso a la caché y a la memoria principal. En el caso de que la tasa de aciertos de la caché fuera 0.9 y la tasa de aciertos de memoria principal fuera 0.999, el tiempo medio de lectura del sistema de memoria pasa a ser 1.13 microsegundos, mucho mayor incluso que el tiempo de acceso a la memoria principal.

Es importante destacar que las expresiones utilizadas son únicamente válidas para accesos de lectura a la jerarquía. Para las escrituras es necesario tener en cuenta la estrategia de escritura que se aplica en cada nivel. Concretamente, las estrategias de escritura de la caché se abordan en la sección 3.3.4.

3.3. La memoria caché

En esta sección se estudia el funcionamiento de la memoria caché y su comunicación con la memoria principal. Además, se presenta un ejemplo de organización de la memoria caché en la arquitectura x86.

3.3.1. Conceptos preliminares

La memoria caché es la memoria más rápida del sistema de memoria. En la actualidad se implementa dentro del chip del procesador utilizando memoria SRAM, lo cual redunda en un alto coste.

La memoria caché está organizada en líneas que contienen bloques, todos ellos conteniendo el mismo número de bytes. Estos bytes se ubican en posiciones de memoria consecutivas. Cada vez que la CPU intenta acceder a una dirección de memoria que no está en ninguna línea de la caché se produce un fallo de caché. La consecuencia del fallo es que se suspende momentáneamente el acceso, se copia el bloque que contiene la dirección (o direcciones si el acceso es a varios bytes) desde la memoria principal a la memoria caché y a continuación se reanuda el acceso. Esto supone que la instrucción que está realizando el acceso es detenida temporalmente esperando por la caché. Por tanto, la existencia de fallos de caché implica un incremento en el CPI de las instrucciones, con la consiguiente reducción del rendimiento.

Desde el punto de vista de la interacción con la memoria caché, la memoria principal (más concretamente, el espacio de direcciones de memoria) se puede ver como un gran almacén de bloques del mismo tamaño que los que pueden almacenar las líneas de la memoria caché. Esto no significa que el espacio de direcciones de memoria esté organizada en bloques.

Por ejemplo, una memoria caché de 32 bytes organizada en líneas de 4 bytes podrá almacenar 8 bloques. Si el espacio de direcciones de memoria es de 256 bytes, la memoria caché percibe este espacio de direcciones como 64 bloques de 4 bytes cada uno, tal como se muestra en la figura 3.4.

La dirección de memoria emitida por la CPU se divide en dos campos:

- El número de bloque.
- El desplazamiento dentro del bloque.

El bloque de memoria en el cual está incluida una dirección emitida por la CPU se obtiene teniendo en cuenta que los bits menos significativos de la dirección se emplean para definir el desplazamiento del byte dentro del bloque y, por lo tanto, los demás bits proporcionan el número de bloque de memoria. Esto se muestra también en la figura 3.4.

La memoria caché lleva asociado un controlador que se encarga de gestionar las lecturas y escrituras en la caché y comprobar, por ejemplo, si una dirección está cacheada, es decir, ubicada dentro de la caché, o por el contrario se ha producido un fallo de caché en el acceso.

En general, el funcionamiento de la memoria caché viene definido por las siguientes características:

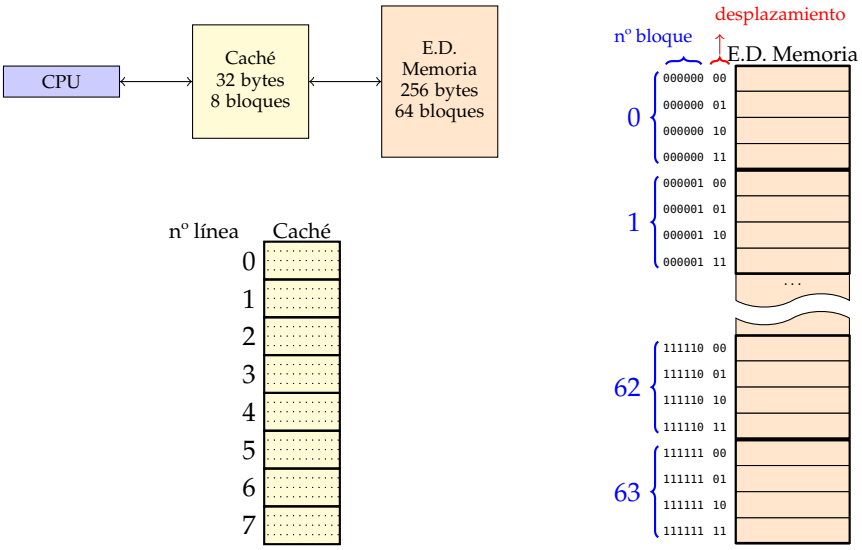


Figura 3.4: Espacio de direcciones de memoria y organización en líneas de la memoria caché

- Estrategia de correspondencia. Establece a qué línea o líneas de caché se puede copiar cada bloque de memoria.
- Estrategia de reemplazo. Establece qué línea de la memoria caché debe reemplazarse para albergar un bloque de memoria cuando se produce un fallo de caché. Esto sólo tiene sentido cuando el bloque de memoria puede ubicarse en más de una línea de caché.
- Estrategia de escritura. Establece la relación entre las escrituras en la memoria caché y el siguiente nivel de la jerarquía de memoria. El objetivo fundamental de esta estrategia es garantizar la coherencia entre la información de ambos niveles de memoria, afectando lo menos posible al rendimiento del sistema.

3.3.2. Estrategias de correspondencia

Básicamente, existen tres estrategias de correspondencia:

- Correspondencia directa.
- Correspondencia (totalmente) asociativa.
- Correspondencia asociativa por conjuntos.

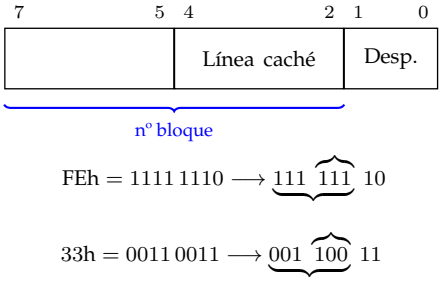


Figura 3.5: Ejemplo de obtención del número de bloque de memoria y línea de caché con correspondencia directa

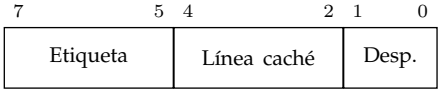


Figura 3.6: Campos de una dirección de memoria cuando se utiliza correspondencia directa

Correspondencia directa

La correspondencia directa es la estrategia de correspondencia más simple. Cada bloque de memoria se asigna siempre a la misma línea de memoria caché según la expresión siguiente:

Línea caché = (Bloque Mem.) MOD (Número de líneas en la caché)

En la práctica el número de líneas de la caché es potencia de dos (2^x), por lo que la línea de memoria caché se obtiene tomando los x bits menos significativos del número binario que representa el bloque de memoria. La figura 3.5 muestra ejemplos de obtención del número de bloque de memoria y el número de línea de caché para la organización mostrada en la figura 3.4.

Cuando se copia un bloque de memoria a una línea de la memoria caché, se escribe conjuntamente con el bloque un conjunto de bits que identifican de forma unívoca el bloque de memoria. Este conjunto de bits recibe el nombre de etiqueta. La dirección de memoria se subdivide entonces en los campos indicados en la figura 3.6 para el caso de emplear correspondencia directa en el sistema de memoria de ejemplo.

Cada línea de la memoria caché incluye también un bit de validez (v) que indica si el bloque almacenado en la línea es válido. Por ejemplo, durante el proceso de inicialización del computador, todas las líneas de la memoria caché son marcadas como inválidas (v=0).

La figura 3.7 muestra la estructura interna de una caché con correspondencia directa. Se puede ver cómo la dirección de memoria se divide en campos para obtener inicialmente el número de línea. A continuación, se compara el campo etiqueta de la dirección con la etiqueta almacenada en la línea. Si coinciden, el campo desplazamiento de la dirección indica el byte de la línea a acceder. Las figuras 3.8 a 3.16

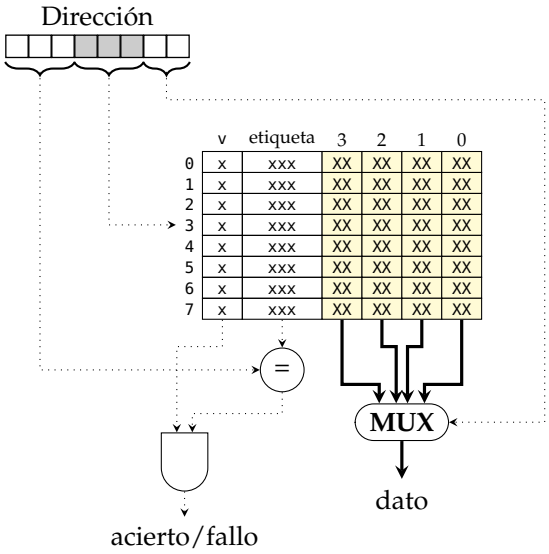


Figura 3.7: Estructura de una caché con correspondencia directa

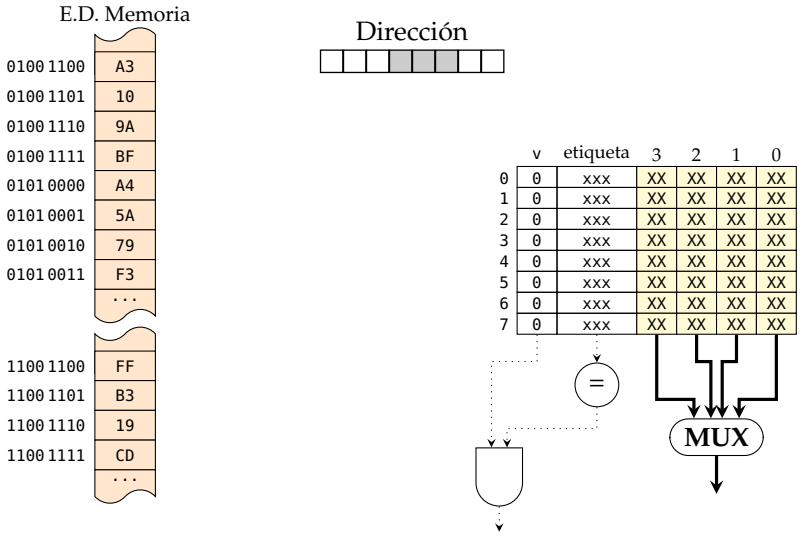


Figura 3.8: Correspondencia directa: situación inicial con la caché vacía

muestran el funcionamiento de este tipo de caché cuando se producen varias lecturas de un byte por parte de la CPU.

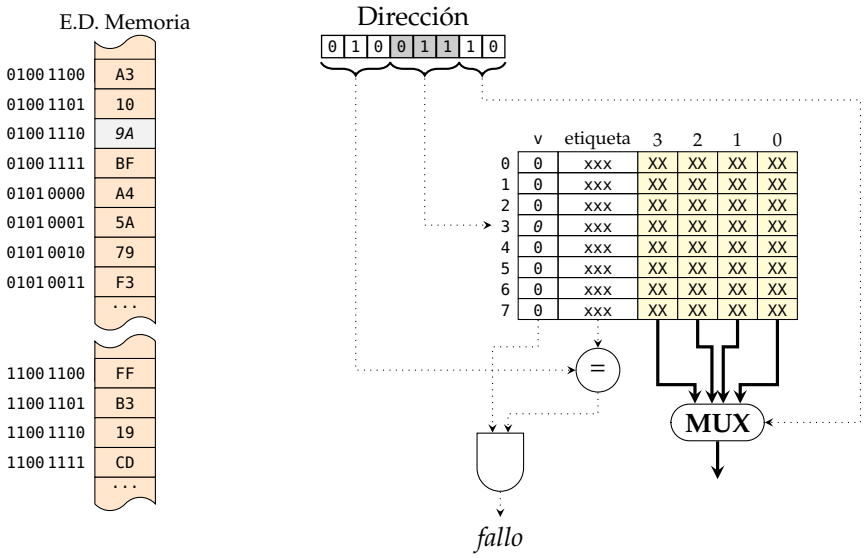


Figura 3.9: Correspondencia directa: petición de lectura de un byte sobre la dirección 4Eh, que produce un fallo de caché

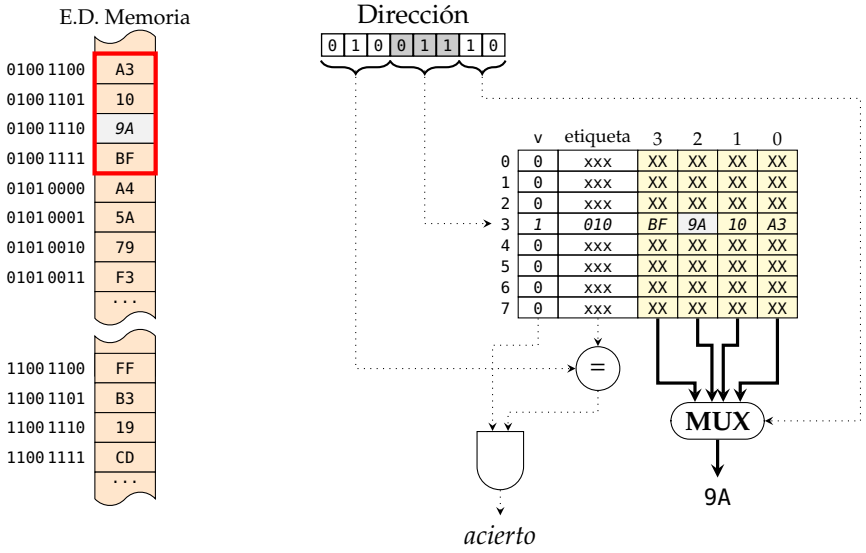


Figura 3.10: Correspondencia directa: se carga la línea 3 y la caché sirve el byte solicitado

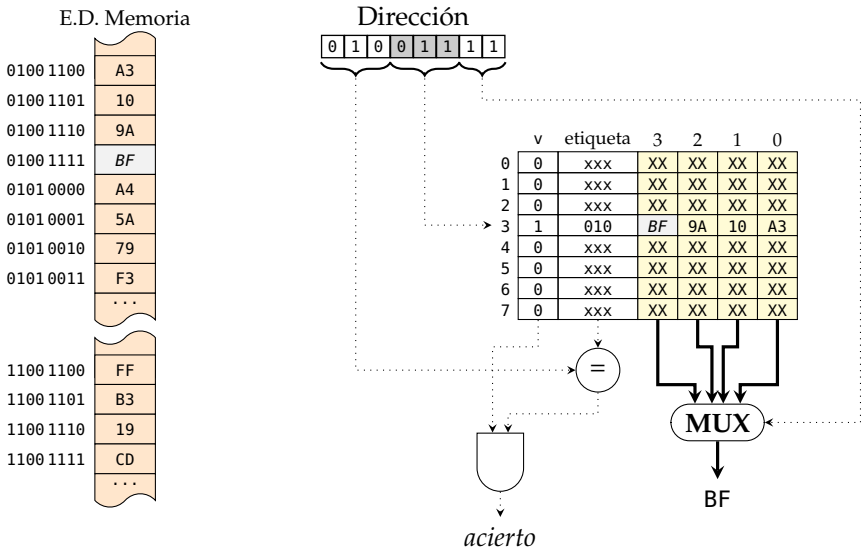


Figura 3.11: Correspondencia directa: petición de lectura de un byte sobre la dirección 4Fh, que produce un acierto de caché

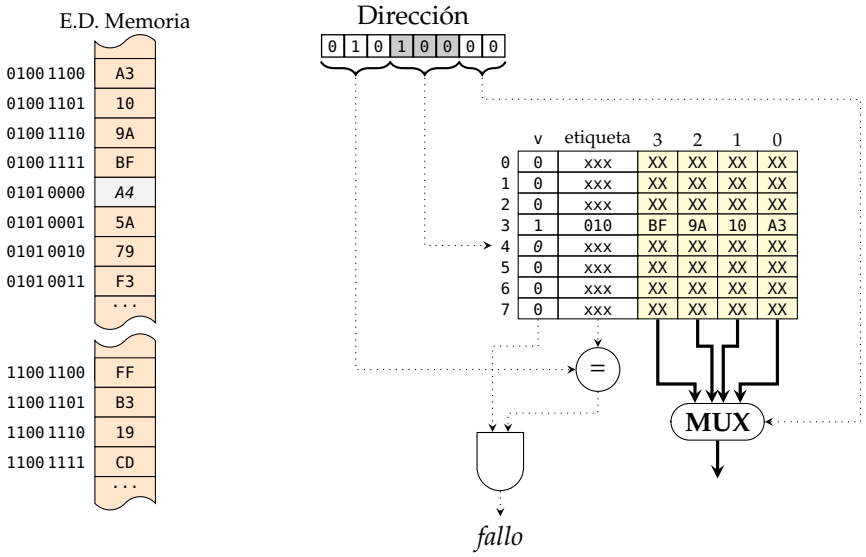


Figura 3.12: Correspondencia directa: petición de lectura de un byte sobre la dirección 50h, que produce un fallo de caché

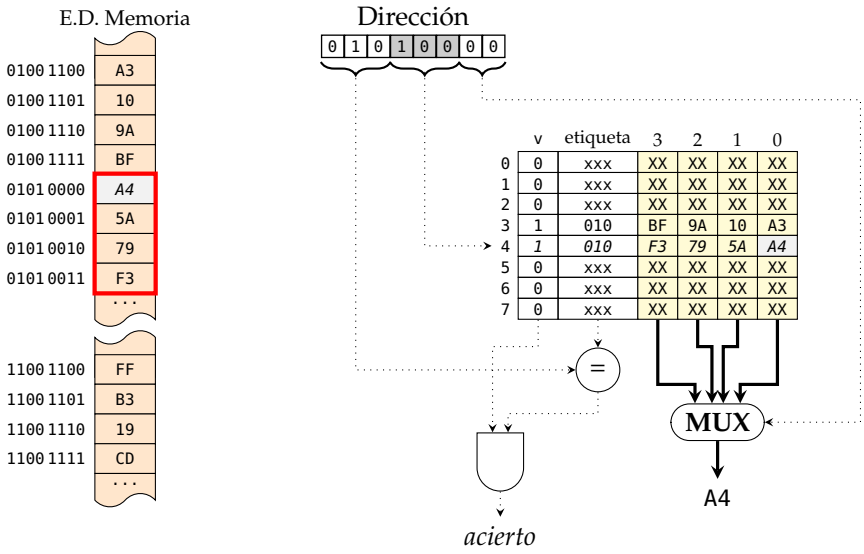


Figura 3.13: Correspondencia directa: se carga la línea 4 y la caché sirve el byte solicitado

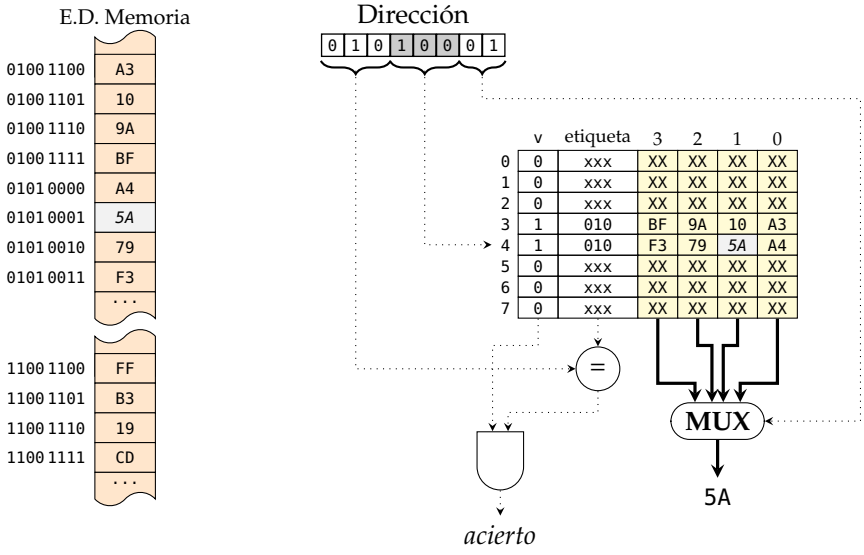


Figura 3.14: Correspondencia directa: petición de lectura de un byte sobre la dirección 51h, que produce un acierto de caché

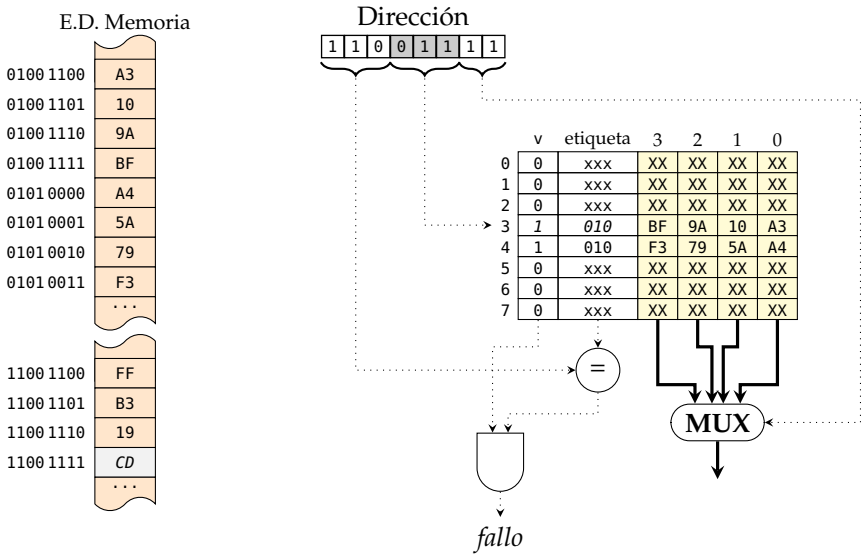


Figura 3.15: Correspondencia directa: petición de lectura de un byte sobre la dirección CFh, que produce un fallo de caché

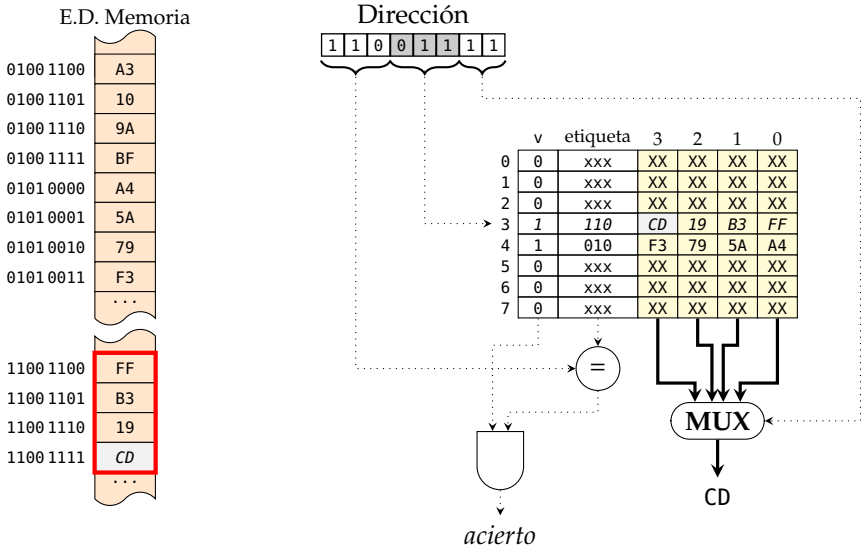


Figura 3.16: Correspondencia directa: reemplazo del bloque en la línea 3 y la caché sirve el byte solicitado

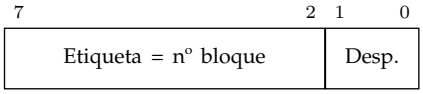


Figura 3.17: Campos de una dirección de memoria cuando se utiliza correspondencia totalmente asociativa

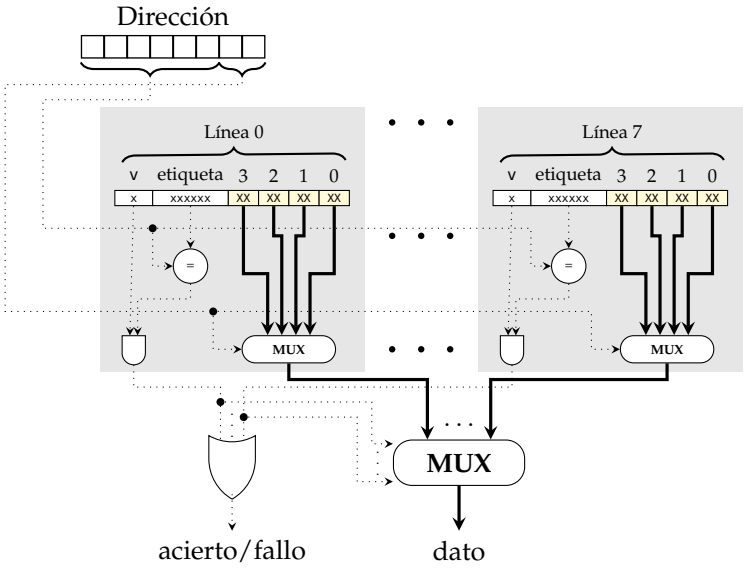


Figura 3.18: Estructura de una caché con correspondencia totalmente asociativa

Correspondencia (totalmente) asociativa

La estrategia de correspondencia directa, explicada anteriormente, es muy sencilla de implementar. Su mayor inconveniente proviene de la rigidez a la hora de asignar bloques de memoria a líneas de caché, pues no hay ninguna libertad. Cada bloque de memoria puede alojarse únicamente en una línea de caché. Si la CPU accede con mucha frecuencia a dos bloques de memoria asignados a la misma línea de caché aparecerán con gran frecuencia fallos de caché, lo cual repercute en un CPI alto y, en consecuencia, un bajo rendimiento.

Para paliar este problema, la estrategia de correspondencia asociativa da total libertad. Un bloque de memoria puede asignarse a cualquier línea de caché. En este caso, la etiqueta que incluye la línea de caché coincide con el número de bloque de memoria. La figura 3.17 ilustra la división en campos de una dirección de memoria bajo esta estrategia de correspondencia.

La figura 3.18 muestra la estructura interna de una caché con correspondencia totalmente asociativa.

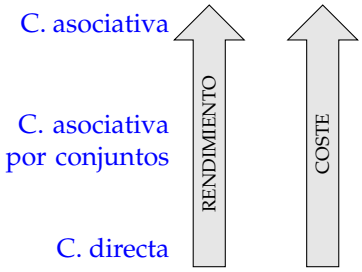


Figura 3.19: Comparación entre las estrategias de correspondencia

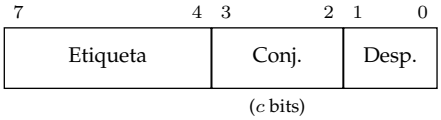


Figura 3.20: Campos de una dirección de memoria cuando se utiliza correspondencia asociativa por conjuntos

Correspondencia asociativa por conjuntos

La estrategia totalmente asociativa proporciona un rendimiento óptimo, derivado de la libertad total en cuanto a la asignación de bloques de memoria a líneas de caché. Sin embargo, presenta un grave inconveniente: el coste de implementación es inasumible para cachés grandes, debido básicamente al elevado número de comparadores que utiliza. Se necesita un comparador por línea. Debe recordarse, que en la correspondencia directa bastaba con un solo comparador para toda la caché. Como solución de compromiso se plantea la correspondencia asociativa por conjuntos, tal como muestra la figura 3.19.

La estrategia de correspondencia asociativa por conjuntos se basa en dividir la memoria caché en conjuntos, cada uno de lo cuales contiene un número fijo de líneas. Cada bloque de memoria puede cachearse en cualquiera de las líneas de un único conjunto. Al número de líneas que forman un conjunto se le denomina número de vías. El número de conjuntos de la memoria caché se elige como una potencia de dos, 2^c, donde c es el número de bits empleados para representar el conjunto. Con este tipo de correspondencia, los c bits menos significativos del número de bloque de memoria proporcionan el conjunto y los restantes la etiqueta, tal como se ilustra en la figura 3.20.

De esta forma, las correspondencias directa y totalmente asociativa vistas anteriormente son casos particulares de la correspondencia asociativa por conjuntos. La correspondencia directa es una correspondencia asociativa por conjuntos de una vía, mientras que la correspondencia totalmente asociativa es una correspondencia asociativa por conjuntos donde hay un único conjunto en la caché que contiene todas sus líneas. En este último caso, el número de vías coincide con el número de líneas de la caché, es decir, el grado de asociatividad es máximo.

La figura 3.21 muestra la asignación de bloques en una memoria caché asociativa por conjuntos.

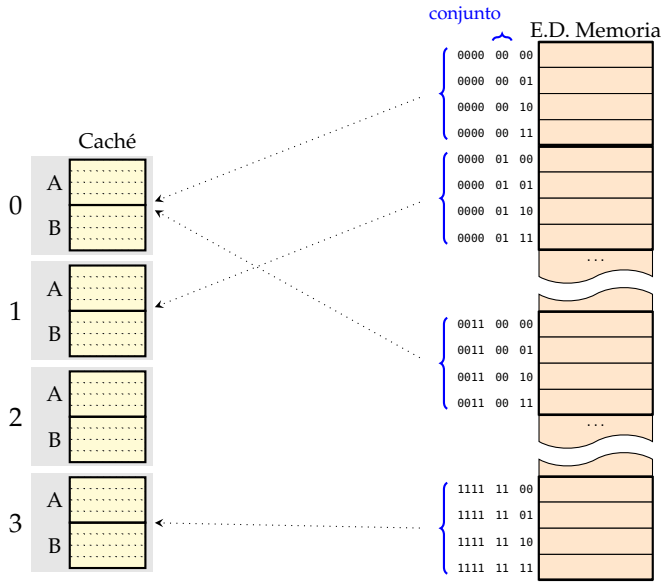


Figura 3.21: Asignación de bloques dentro de una memoria caché asociativa por conjuntos

La figura 3.22 muestra la organización de una memoria caché asociativa por conjuntos. Se aprecia en la figura cómo la dirección de memoria se divide en campos para obtener inicialmente el número de conjunto. A continuación, se compara el campo etiqueta de la dirección con las etiquetas almacenadas en todas las vías del conjunto. Si se produce alguna coincidencia, el campo desplazamiento de la dirección indica el byte de la línea a acceder.

Las figuras 3.23 a 3.25 muestran el funcionamiento de este tipo de caché cuando se producen varias lecturas de un byte por parte de la CPU.

3.3.3. Estrategias de reemplazo

Cada vez que se produce un fallo en la memoria caché se copia un bloque desde el espacio de direcciones de memoria a una línea de la memoria caché, posiblemente reemplazando a un bloque previamente cacheado.

En el caso de seguir una correspondencia directa, está claro el bloque que hay que reemplazar. Sin embargo, en el caso de seguir una correspondencia asociativa o asociativa por conjuntos puede elegirse entre varios bloques candidatos. La estrategia seguida para reemplazar un bloque u otro dentro de los posibles es a lo que se denomina estrategia de reemplazo. Dos son las estrategias que se consideran:

- *Least Recently Used* (LRU). Cada línea de caché almacena unos bits adicionales que indican cuál lleva más tiempo sin haber sido accedida. El bloque de esta línea es entonces reemplazado.
- Reeemplazo aleatorio. Se elige de forma aleatoria entre uno cualquiera de los bloques candidatos.

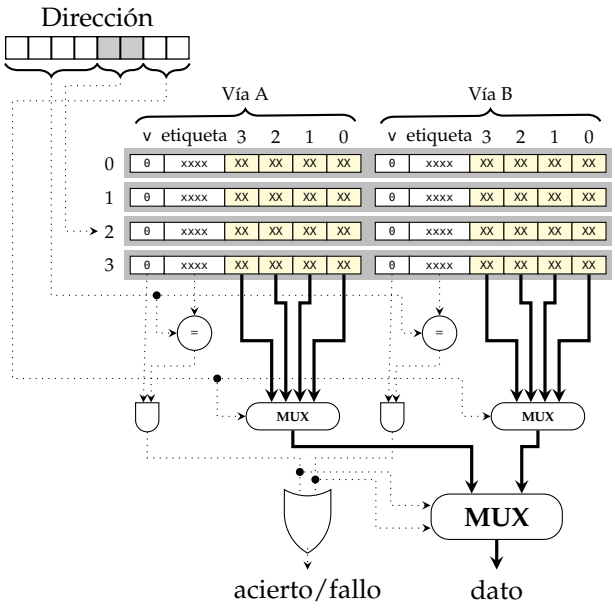


Figura 3.22: Estructura de una caché asociativa por conjuntos

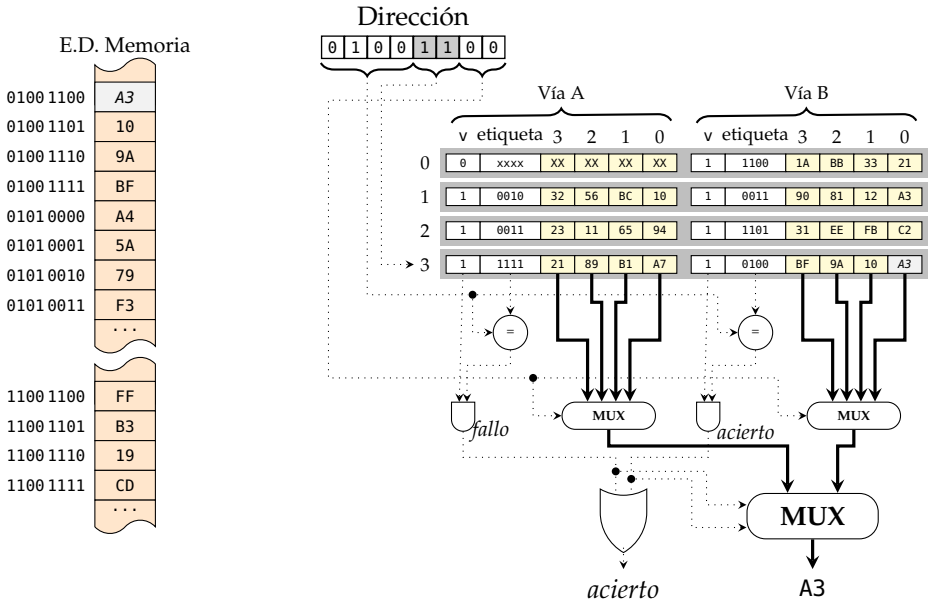


Figura 3.23: Correspondencia asociativa por conjuntos: petición de lectura de un byte sobre la dirección 4Ch, que produce un acierto de caché

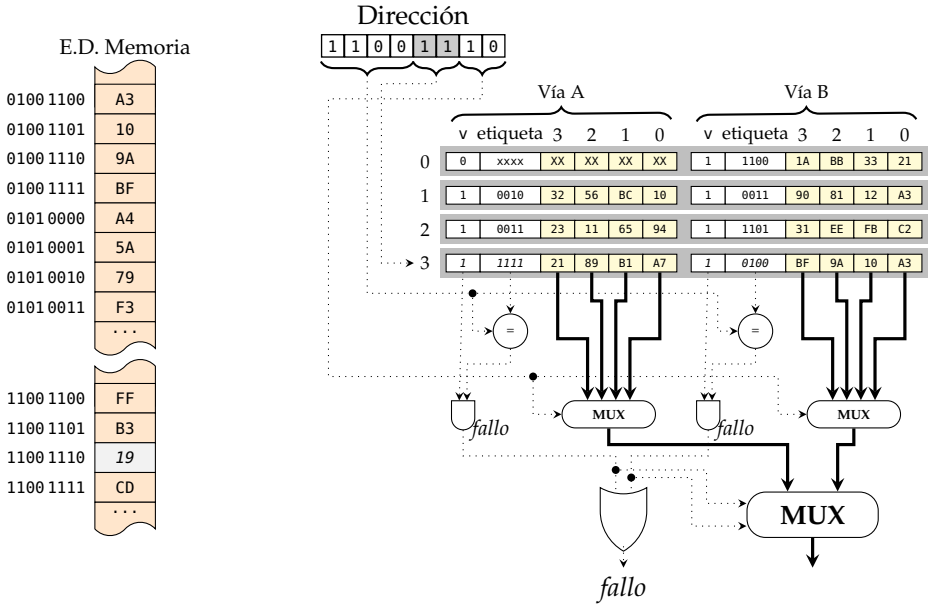


Figura 3.24: Correspondencia asociativa por conjuntos: petición de lectura de un byte sobre la dirección CEh, que produce un fallo de caché

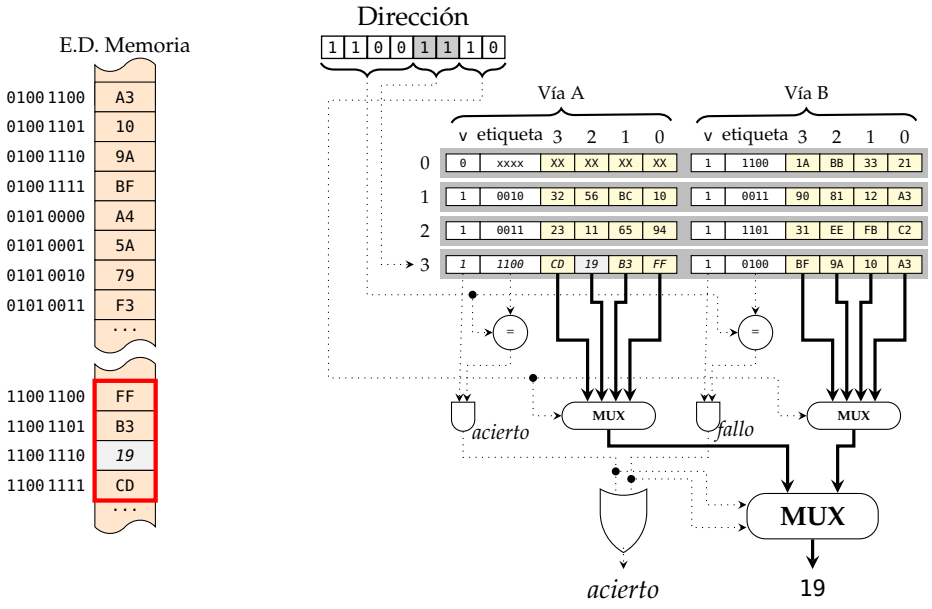


Figura 3.25: Correspondencia asociativa por conjuntos: reemplazo del bloque en la vía A del conjunto 3 y la caché sirve el byte solicitado

3.3.4. Estrategias de escritura

Dos son las estrategias habituales de escritura:

- Escritura a través (*write-through*).
- Escritura diferida (*write-back*).

Con la escritura a través, toda operación de escritura se lleva a cabo simultáneamente sobre la memoria caché y el espacio de direcciones de memoria.

Por el contrario, con la escritura diferida, los datos inicialmente sólo se escriben en la caché. El dato escrito aparece reflejado en el espacio de direcciones de memoria sólo cuando el bloque que lo contiene es reemplazado en la caché. Con la escritura diferida aparecen problemas de coherencia, cuya solución se abordará posteriormente.

Los accesos de escritura, como ocurre con los de lectura, pueden dar lugar a aciertos o a fallos de caché. En el primer caso, se procede tal como dicta la estrategia de escritura seguida: se escribe únicamente en memoria caché, para el caso de escritura diferida, o se escribe en memoria caché y en el espacio de direcciones de memoria en el caso de escritura a través.

El problema aparece cuando ocurre un fallo de caché en la escritura. En ese caso hay dos estrategias que pueden seguirse: *write allocate* o *no write allocate*. Bajo una estrategia *write allocate* el comportamiento para las escrituras es idéntico que para las lecturas: el bloque se copia en memoria caché antes de realizar la escritura. Con una estrategia *no write allocate* el bloque no se lleva a la caché, sino que sólo se escribe en el espacio de direcciones de memoria, evitando el coste que supone copiar todo el bloque a la memoria caché previamente.

Aunque pueden darse varias combinaciones, las más comunes son *write-back* junto con *write allocate* y *write-through* junto con *no write allocate*.

Comparativamente, la estrategia *write-back* proporciona en general un mayor rendimiento que la estrategia *write-through*, a costa de una mayor complejidad hardware para el mantenimiento de la coherencia.

Las figuras 3.26 a 3.30 muestran el funcionamiento de la estrategia *write-back*. Conviene resaltar que se asocia un bit más a cada línea de caché, el bit *dirty* (`d`), que se pone a uno cada vez que la CPU escribe en la línea.

Las operaciones de escritura pueden generar problemas de coherencia entre las diferentes copias de la información que se almacenan en varios niveles de la jerarquía de memoria, tal y como se analizará en la sección 3.3.6.

3.3.5. Organización de la memoria caché

Hay dos factores determinantes en la organización de una memoria caché:

- El número de niveles de caché.
- La separación en cachés de datos y código.

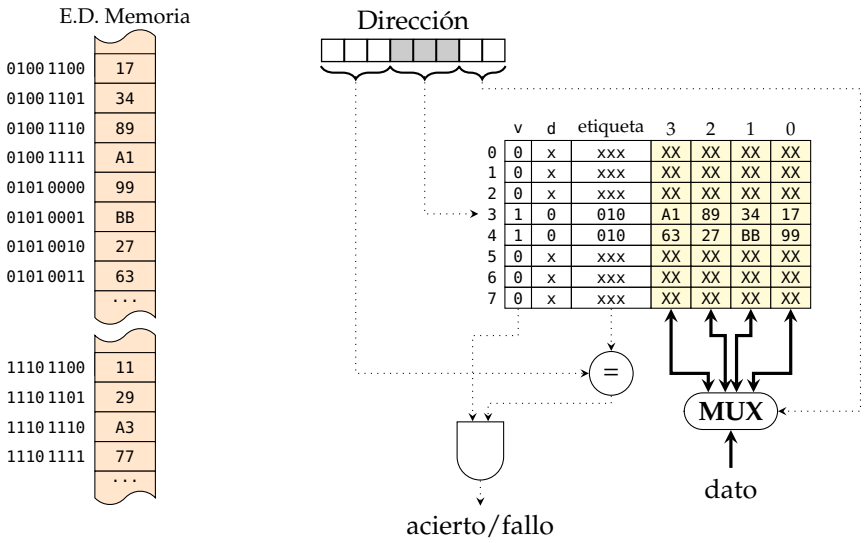


Figura 3.26: Funcionamiento de la estrategia de escritura *write-back*: situación inicial de la caché y del espacio de direcciones de memoria

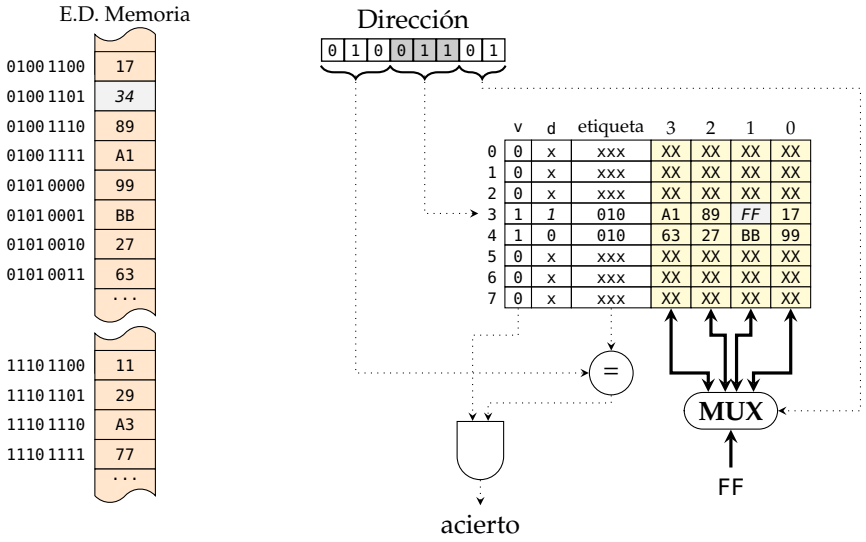


Figura 3.27: Funcionamiento de la estrategia de escritura *write-back*: petición de escritura de un byte sobre la dirección 4Dh; acierto, el bloque se marca como sucio

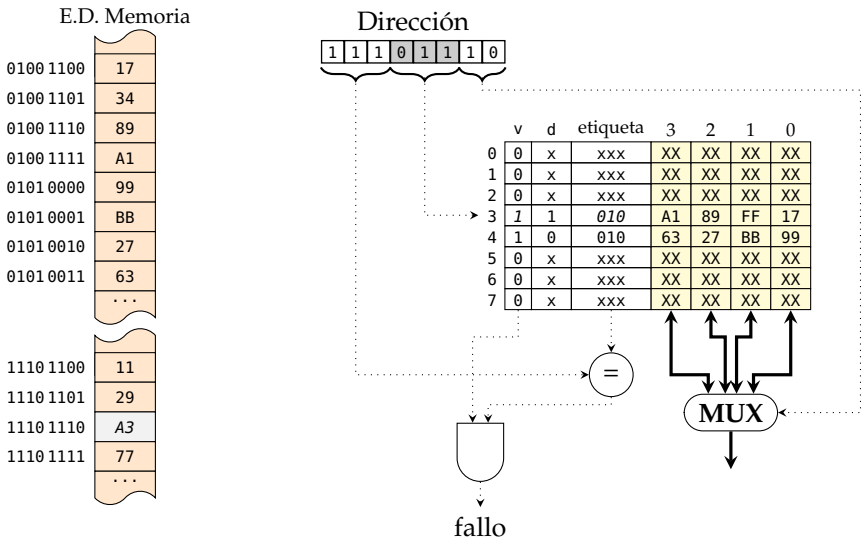


Figura 3.28: Funcionamiento de la estrategia de escritura *write-back*: petición de lectura de un byte sobre la dirección EEh, que origina un fallo de caché

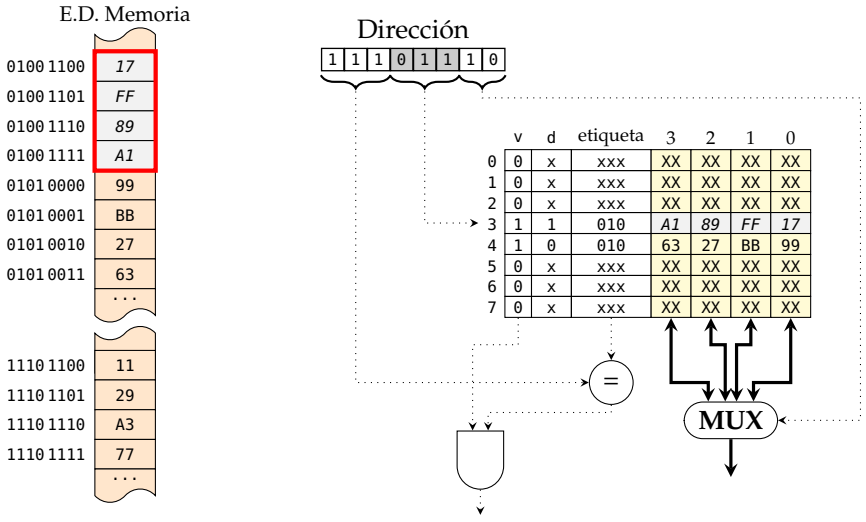


Figura 3.29: Funcionamiento de la estrategia de escritura *write-back*: actualización del bloque sucio en el espacio de direcciones de memoria

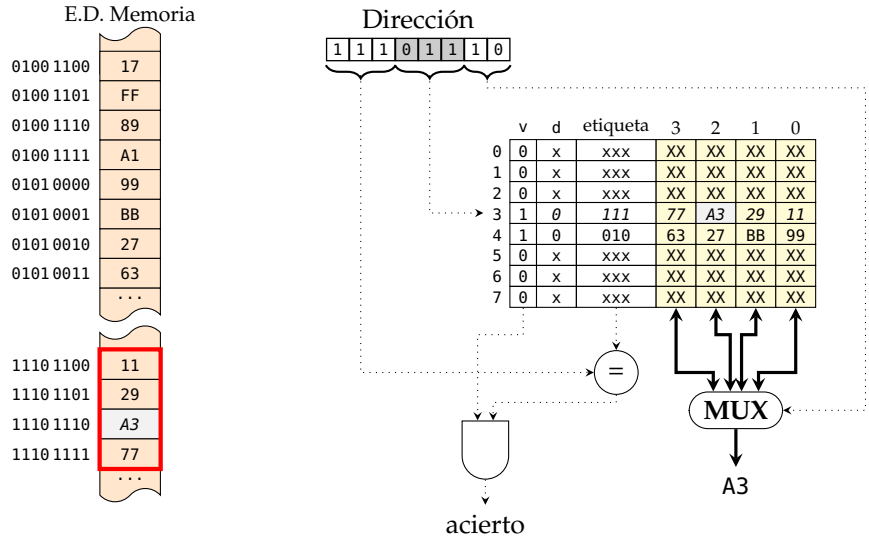


Figura 3.30: Funcionamiento de la estrategia de escritura *write-back*: reemplazo del bloque en la línea 3 y la caché sirve el byte solicitado

Inicialmente cuando se introdujeron las cachés, el sistema típico tenía una única caché. Sin embargo, lo habitual hoy en día es disponer de dos o tres niveles de caché con el objetivo de incrementar la tasa de aciertos de caché a la vez que se minimiza el tiempo de acceso. Estos niveles se denominan L1, L2 y así sucesivamente hasta LLC (*Last Level Cache*), donde el nivel L1 es el más cercano a la CPU y el de menor tamaño, y LLC el más alejado, generalmente compartido entre núcleos. La figura 3.31 muestra una jerarquía de memoria con varios niveles de caché. En el ejemplo, LLC es la caché L2.

El objetivo de la caché L2, y sucesivas, es amortiguar la penalización que supone un fallo de caché en el nivel L1. La diferencia en los tiempos de acceso entre la caché L1 y la memoria principal son demasiado grandes. La caché L2 es de mayor capacidad que la caché L1, por lo que también es más lenta, pero todavía bastante más rápida que la memoria principal. En teoría, cuantos más niveles de caché haya entre la CPU y la memoria principal mejor rendimiento proporciona el sistema de memoria, pero con un coste mayor.

Como se vio en el capítulo dedicado a la CPU, la microarquitectura MIPS64 incorpora memorias independientes para código y datos. Esto se traduce en que una CPU MIPS64 incluye dos cachés L1: una para código y otra para datos, en contraposición con una única caché para datos y código. Podría suceder que la memoria caché de datos tuviese líneas libres, mientras que la memoria caché de código estuviese generando fallos de caché por estar llena. Por su parte, con una caché unificada se reparte automáticamente los bloques cacheados entre datos y código, de tal forma que si son necesarios menos bloques de datos y más de código estos se reparten de forma eficiente y transparente entre las líneas de la caché.

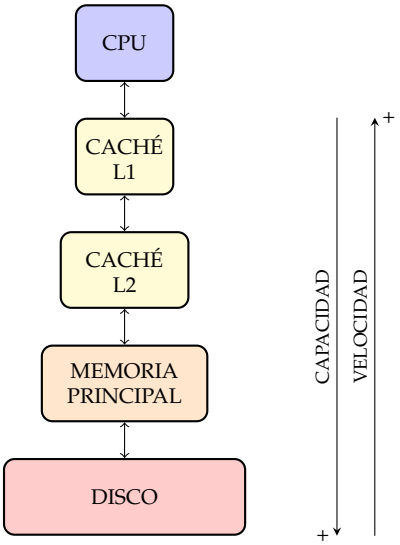


Figura 3.31: Jerarquía de memoria con dos niveles de caché

Las primeras cachés que aparecieron solían ser unificadas debido a la observación anterior, ya que obtienen un mejor rendimiento medido en tasa de aciertos. Sin embargo, las cachés L1 actuales suelen ser divididas por el elevado grado de paralelismo de las CPU. La caché dividida permite un mayor grado de paralelismo, pues una etapa del *pipeline* de la CPU puede escribir un dato en memoria mientras simultáneamente otra etapa puede leer los códigos de las instrucciones.

3.3.6. Coherencia de caché

La CPU trabaja siempre sobre una copia de la información almacenada en el espacio de direcciones de memoria. Esta copia reside en la memoria caché. Si otros elementos del computador pueden acceder al espacio de direcciones de memoria, entonces se generan problemas de coherencia.

El principio de coherencia de la memoria establece lo siguiente: la lectura realizada por una CPU de una dirección de memoria tiene que coincidir con el último valor escrito en esa dirección de memoria por cualquier otro elemento del computador.

Coherencia de caché de un único nivel en sistemas con una única CPU

El problema de la coherencia de caché en sistemas con una única CPU aparece cuando se tienen interfaces de periféricos ubicadas en el espacio de direcciones de memoria. Por ejemplo, si la interfaz de un teclado está ubicada en el espacio de direcciones de memoria y el usuario pulsa una tecla, el código de la tecla pasa a estar almacenado en una de las posiciones del espacio de direcciones de memoria. El problema es que si el bloque de memoria que contiene la dirección de la interfaz está cacheado, la caché no es consciente de esta escritura y, por lo tanto, no contiene

información actualizada. Si la CPU accede a la interfaz del teclado para obtener el código de la tecla pulsada, recibe este dato de la caché y es incorrecto.

La solución en este caso es simple: las áreas de memoria asociadas a interfaces de periféricos se marcan como no cacheables. La caché hace que el acceso a cualquiera de estas posiciones se transmita directamente al espacio de direcciones de memoria.

Otro problema de coherencia más complejo aparece cuando existen interfaces de periféricos con capacidad de DMA (Direct Memory Access)[4]. Por ejemplo, la interfaz del disco escribe en memoria principal un sector de datos solicitado cuando ya dispone de él. Este sector requiere escribir del orden de 1 KiB en posiciones contiguas de memoria. La técnica de marcar ese área como no cacheable soluciona el problema de coherencia pero ocasiona otro problema: el rendimiento en los accesos de la CPU al sector se reduce considerablemente. Este acceso se hace al espacio de direcciones de memoria en lugar de a la memoria caché. Si se tratase de acceder a varios bytes como en el caso de la interfaz de un teclado no habría problema, pero el acceso a un conjunto grande de bytes supone una penalización de rendimiento excesiva.

En el caso de interfaces de periféricos con capacidad de DMA aparecen los siguientes problemas de coherencia:

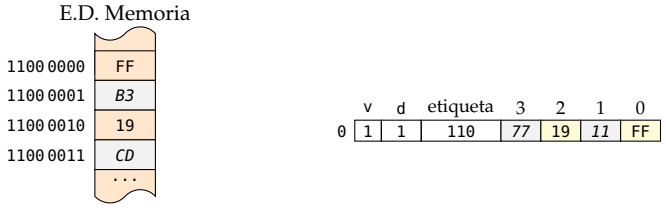
- Un bloque de la memoria que ha sido previamente cacheado, ha sido modificado más tarde por un periférico. La lectura del bloque por parte de la CPU proporcionará datos no actualizados, pues los lee de la memoria caché.
- La CPU modifica una línea de memoria caché empleando la estrategia *write-back*. Si antes de ser reemplazado el bloque en la memoria caché un periférico lee este bloque de la memoria principal empleando DMA los datos que obtiene no serán válidos, pues no están actualizados. Esta incoherencia no aparece cuando se emplea la estrategia de escritura *write-through*.

Los problemas de coherencia anteriores suelen resolverse empleando la técnica de espionaje o *snooping*. Con esta técnica, el controlador de la memoria caché observa continuamente las líneas de control de lectura y escritura del espacio de direcciones de memoria, así como sus líneas de direcciones. En el primer caso, cuando detecta que un periférico está escribiendo un bloque de memoria que está cacheado invalida la línea de caché donde se encuentra el bloque desactivando su bit de validez. En el segundo caso, el controlador de memoria caché detiene temporalmente la lectura de la memoria por parte del periférico, actualiza el bloque de memoria que estaba cacheado y permite a continuación que prosiga la lectura de la memoria por parte del periférico.

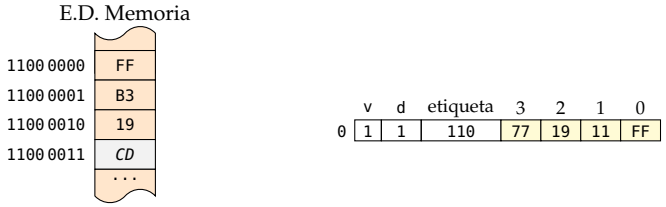
Las figuras 3.32 a 3.34 muestran los tres casos en los que aparecen problemas de coherencia entre la memoria caché y el espacio de direcciones de memoria.

El problema de coherencia mostrado en la figura 3.32 para la estrategia *write-back* también aparece con la estrategia *write-through* (figura 3.33). No obstante, con la primera aparece un problema de coherencia que no aparece con la estrategia *write-through* cuando la interfaz de un periférico con capacidad de DMA escribe sobre un bloque de memoria cacheado y marcado como sucio. La solución se muestra en la figura 3.34.

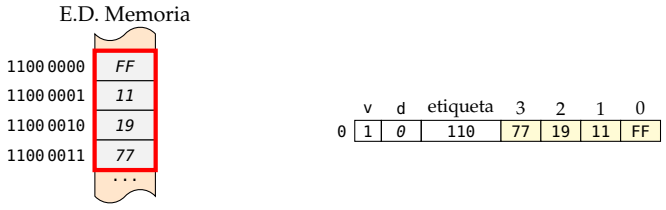
[4]La técnica de DMA se estudia en el capítulo dedicado a la entrada y salida.



(a) Situación inicial: línea de caché modificada, incoherente con el espacio de direcciones de memoria

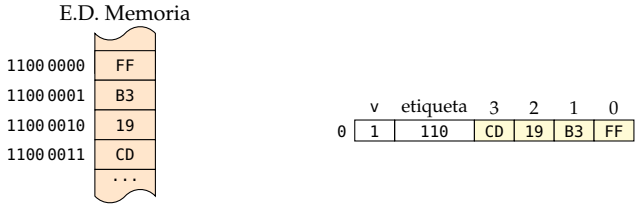


(b) Periférico solicita lectura de la dirección C3h. El controlador de caché detecta que esta dirección pertenece a una línea modificada y detiene la lectura

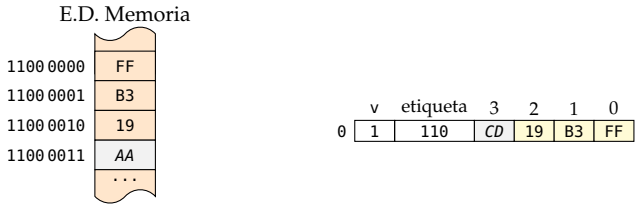


(c) El controlador de caché actualiza el bloque en el espacio de direcciones de memoria (el bit d se pone a 0), después la lectura puede llevarse a cabo

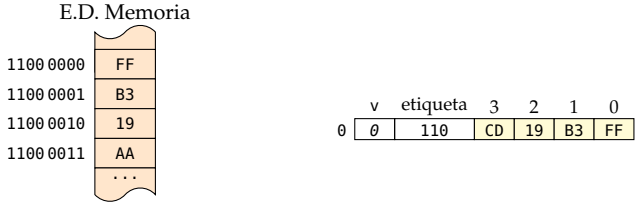
Figura 3.32: Coherencia de caché en la lectura por parte de la interfaz con escritura *write-back*



(a) Situación inicial

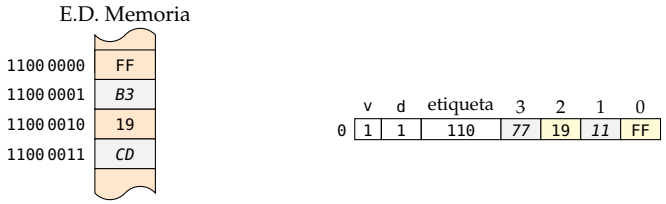


(b) Periférico escribe el dato AAh en la dirección C3h

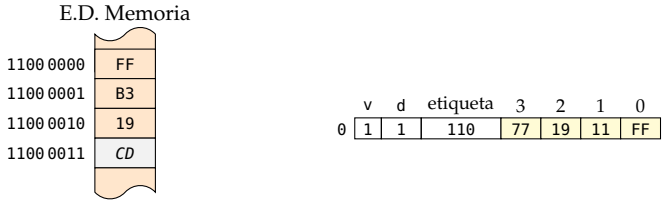


(c) El controlador de caché detecta que esta dirección está cacheada e invalida la línea

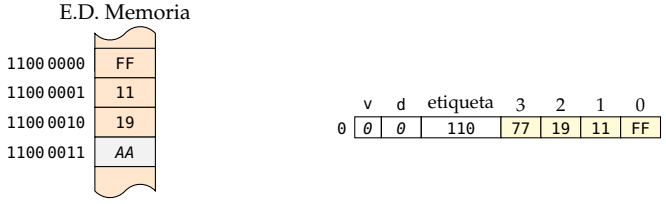
Figura 3.33: Coherencia de caché en la escritura por parte de la interfaz con escritura *write-through*



(a) Situación inicial: línea de caché modificada, incoherente con el espacio de direcciones de memoria



(b) Periférico solicita escritura del dato AAh en la dirección C3h. El controlador de caché detecta que esta dirección pertenece a una línea modificada y detiene la escritura



(c) El controlador de caché actualiza el bloque de memoria (bit d se pone a 0), invalida la línea (bit v se pone a 0) y permite la escritura

Figura 3.34: Coherencia de caché en la escritura por parte de la interfaz con escritura *write-back*

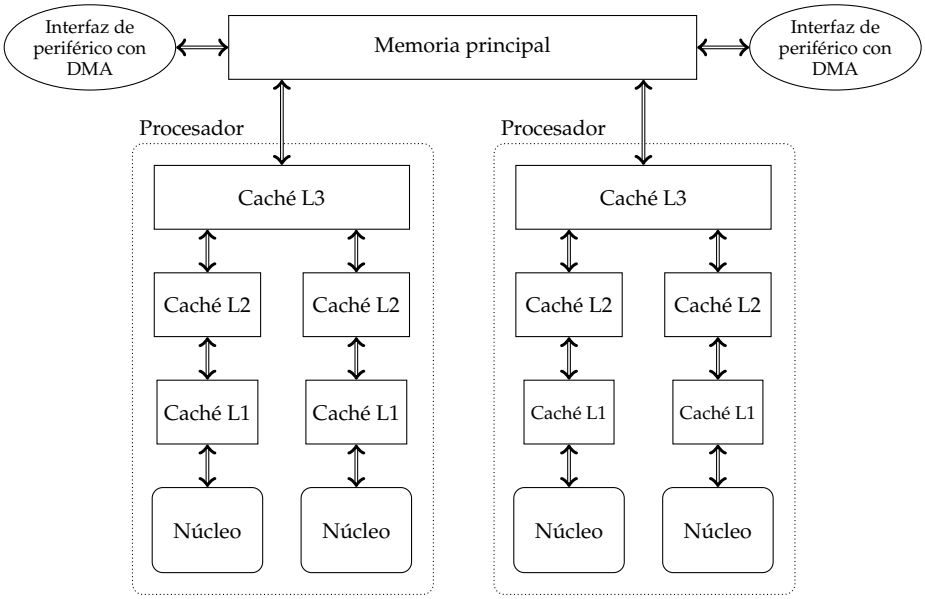


Figura 3.35: Ejemplo de multiprocesador con múltiples núcleos por procesador y varios niveles de caché

Coherencia de caché en sistemas con varias CPU

En la descripción anterior, la memoria caché tiene un único nivel, el sistema una única CPU y la fuente de incoherencia proviene de interfaces de periféricos con capacidad de DMA. En muchos casos los computadores disponen de varios procesadores, cada uno de los cuales incorpora varios núcleos y cada núcleo dispone de varios niveles de caché. La figura 3.35 muestra un ejemplo de esta situación.

Para simplificar, no se hará referencia a interfaces con capacidad de DMA, pues desde el punto de vista de la coherencia de caché pueden asimilarse a una CPU sin caché.

En este escenario, el sistema de memoria en su conjunto es accedido por múltiples CPU para realizar lecturas y escrituras. En la práctica, por cuestiones de rendimiento se emplea la escritura en caché diferida (*write-back*), por lo que la escritura que realiza un procesador en una dirección de memoria aparece en su caché L1 y no migra hacia los niveles de caché más alejados u otras memorias del sistema. El dato escrito no es directamente visible por otras CPU, incluso aunque compartan el mismo procesador. La situación puede ser incluso peor si la dirección de memoria en la que desea escribir ya ha sido previamente escrita por otra CPU.

Estos problemas aparecen cuando los programas que se ejecutan comparten memoria, lo cual es frecuente en los sistemas con varias CPU. Ante toda esta complejidad, hacen falta mecanismos de coherencia más sofisticados que los vistos en el caso de una única CPU.

El mantenimiento de la coherencia influye en el rendimiento del sistema, pues el acceso a memoria es crítico en la ejecución de los programas. La estrategia a seguir es

evitar que estos mecanismos interfieran cuando se accede a direcciones de memoria no compartidas y reducir su impacto cuando se accede a direcciones de memoria compartidas con otras CPU.

El problema fundamental a resolver es la gestión de escrituras, ya que si las CPU solo leyesen de memoria no habría ningún problema de coherencia. Cuando una CPU escribe en una dirección que se encuentra en su caché L1, puede ocurrir que esa dirección también se encuentre cacheada en los niveles de caché L2 y L3 de la misma CPU, o en cualquiera de los niveles de caché de otras CPU. Hay dos alternativas:

- **Actualizar en escritura** (*write-update*). El dato escrito en la caché L1 de una CPU es actualizado en todas las cachés del sistema en las cuales se encuentre cacheada la dirección, incluyendo otras cachés de la misma CPU y las del resto.
- **Invalidar en escritura** (*write-invalidate*). El dato se escribe en la caché L1 de la CPU y se marcan como inválidas todas las líneas en el resto de cachés del sistema que tengan la dirección cacheada. De esta forma, cuando se intente acceder a esa dirección en otras cachés se producirá un fallo de caché y recibirán un bloque actualizado con la escritura realizada.

En la práctica se emplea la estrategia de invalidar en escritura, pues es mucho más rápido invalidar líneas de caché que actualizarlas. Además, frecuentemente se realizan varias actualizaciones consecutivas antes de una lectura, por lo que el coste temporal del mantenimiento de la coherencia es aún menor.

Sea cual sea la alternativa elegida, actualizar o invalidar, cualquier operación de lectura o escritura de memoria por parte de un procesador puede suponer un cambio en múltiples cachés del sistema y en la memoria principal. Por esta razón debe transmitir información para el mantenimiento de la coherencia al resto de cachés del sistema, ya sean del mismo procesador o de otros procesadores, para que estas actualicen sus estados. Hay dos formas básicas de transmitir información de coherencia:

- Espionaje. Cualquier operación de lectura o escritura en una caché es visible a todas las cachés del sistema, e incluso a la memoria principal. Así por ejemplo, si una CPU escribe en su caché L1, todas las cachés del sistema observan la dirección de escritura y si está incluida en alguna de sus líneas marcan la línea como inválida. Si se tiene en cuenta el elevado número total de cachés que pueden coexistir en un sistema multiprocesador y multinúcleo, a priori este enfoque sería ineficiente. Sin embargo, son posibles múltiples optimizaciones que lo hacen viable. Por ejemplo, si se sabe que una línea de caché se encuentra únicamente en la caché L1 de una CPU, se puede leer o escribir sin transmitir información de coherencia a otras cachés. Este es, además, el caso más común, pues la inmensa mayoría de los accesos a memoria se realizan sobre direcciones no compartidas.
- Directorio. Existe un dispositivo hardware que gestiona la coherencia del sistema de memoria y almacena información de coherencia para todas las cachés. Todas aquellas operaciones de lectura o escritura sobre las cachés o la memoria principal que puedan estar afectadas, o afecten a la coherencia, requieren previamente una consulta del directorio y/o una modificación del estado del mismo.

En los sistemas con pocos procesadores es común el empleo de la técnica de espionaje, pero en sistemas multiprocesadores a gran escala se emplea la técnica de directorio.

El protocolo MESI

El protocolo de coherencia MESI es la base de muchos protocolos de coherencia actuales[5]. El objetivo de este apartado es ilustrar la filosofía de trabajo de este mecanismo de coherencia en sistemas con varias CPU sin proporcionar todos los detalles del protocolo.

Se trata de un protocolo de invalidación en escritura comúnmente empleado con *snooping*, en el que cada línea en una caché cualquiera puede encontrarse en uno de estos cuatro estados de coherencia:

- Modificada (M). La línea de caché es válida, ha sido modificada y, por lo tanto, no es coherente con la memoria principal. El bloque contenido en la línea no está cacheado en otras cachés.
- Exclusiva (E). La línea es válida, no ha sido modificada y, por lo tanto, es coherente con la memoria principal. El bloque contenido en la línea no está cacheado en otras cachés.
- Compartida (S). La línea es válida, no ha sido modificada y, por lo, tanto es coherente con la memoria principal. El bloque contenido en la línea puede encontrarse también en otras cachés.
- Inválida (I). La línea de caché es inválida; no contiene ningún bloque.

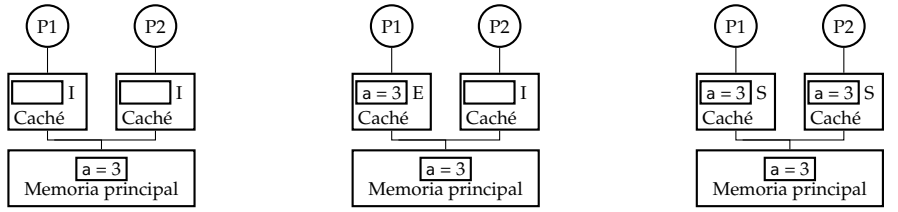
Cada vez que una CPU realiza una lectura o escritura de su caché puede cambiar el estado MESI de una línea de su caché L1 y de cualquier otra caché del sistema.

La figura 3.36 ilustra los cambios de estado de las líneas de caché donde se ubica una variable `a` compartida en un sistema con dos CPU, P1 y P2, cada una con un nivel de caché, cuando se accede a la misma. Inicialmente, el bloque que contiene la variable no está cacheado (figura 3.36a).

Cuando P1 lee por primera vez la variable se produce un fallo de caché y se carga el bloque de memoria que la contiene en una de sus líneas (figura 3.36b). Dado que el bloque de memoria no se encuentra en la caché de la otra CPU, la línea se marca como exclusiva (E). La siguiente ocasión que P1 lee la variable se produce un acierto de caché y no es necesario actualizar el estado de coherencia de ninguna caché del sistema.

Una posterior lectura de la variable por parte de P2 supondrá un fallo de caché, pero en este caso en la caché asociada a esta CPU. Sin embargo, a diferencia del caso anterior, el bloque se encuentra en la caché de P1 además de en memoria principal. El bloque se carga en la caché de P2 desde memoria principal y se marcan las líneas de ambas cachés como compartidas (S), tal como ilustra la figura 3.36c.

[5]Por ejemplo, Intel emplea una variación denominada MESIF, mientras que AMD utiliza otra variación denominada MOESI.



(a) Situación inicial (b) Tras lectura de a por P1 (c) Tras lectura de a por P2

(d) Tras escritura de a por P1 (e) Tras lectura de a por P2

Figura 3.36: Ejemplo de funcionamiento del protocolo MESI

Si a continuación P1 escribe en la variable, se modifica la línea de su caché pero no se actualiza en memoria principal (dada la política de escritura *write-back*). Además, se marcan como inválida (I) la línea de caché correspondiente en P2 y la línea de la caché en P1 como modificada (M). El estado en que queda el sistema es el mostrado en la figura 3.36d.

Por último, si P2 vuelve a leer la variable se producirá de nuevo un fallo de caché. Dado que el bloque se encuentra modificado en la caché de P1, se actualiza el bloque en memoria principal para a continuación cargarlo en la caché de P2. Finalmente, ambas líneas se marcan como compartidas (figura 3.36e).

La mayor parte de los accesos a memoria por parte de una CPU ocurren a direcciones de memoria no compartidas que se encuentran en líneas de su caché L1, en el estado modificado (M) o exclusivo (E). Por lo tanto, cualquier lectura o escritura de esas direcciones no requiere transmitir información de coherencia a otras cachés del sistema, cumpliendo con el objetivo de evitar que los mecanismos de coherencia interfieran en el acceso a direcciones de memoria no compartidas. De esta forma se minimizan los efectos sobre el rendimiento provocados por el mantenimiento de la coherencia.

3.3.7. Memoria caché en el PC

Las características de la caché varían en gran medida de unos procesadores a otros. Actualmente los procesadores tienen dos o tres niveles de caché: L1, L2 y L3. El nivel L1 es una caché dividida para código y para datos, y el nivel L2 una caché unificada que puede estar compartida entre varios núcleos o ser específica de cada uno como el nivel L1. La caché L3 es unificada y común a todo el procesador.

Todos los niveles de caché siguen la estrategia de correspondencia asociativa por conjuntos.

En cuanto a la ubicación física, todos niveles de caché se encuentran dentro del procesador.

Por ejemplo, en el procesador Intel Core i5-6600K, con cuatro núcleos, la memoria caché tiene las siguientes características:

- Caché L1 de datos específica de cada núcleo (4 cachés).
  - Tamaño: 32 KiB.
  - Tamaño de la línea: 64 bytes.
  - Número de vías: 8.
  - Número de conjuntos: 32 KiB / (8 $\times$ 64) bytes = 64.
- Caché L1 de código específica de cada núcleo (4 cachés).
  - Tamaño: 32 KiB.
  - Tamaño de la línea: 64 bytes.
  - Número de vías: 8.
  - Número de conjuntos: 32 KiB / (8 $\times$ 64) bytes = 64.
- Caché L2 unificada específica de cada núcleo (4 cachés).
  - Tamaño: 256 KiB.
  - Tamaño de la línea: 64 bytes.
  - Número de vías: 4.
  - Número de conjuntos: 256 KiB / (4 $\times$ 64) bytes = 1024.
- Caché L3 unificada común a todo el procesador
  - Tamaño: 6 MiB.
  - Tamaño de la línea: 64 bytes.
  - Número de vías: 12.
  - Número de conjuntos: 6 MiB / (12 $\times$ 64) bytes = 8192.

3.4. La memoria virtual

La memoria virtual es un mecanismo de gestión de la memoria que permite utilizar el disco (o de forma general el almacenamiento secundario) como un nivel de la jerarquía de memoria, pero proporciona además muchas otras características que se irán detallando a lo largo de esta sección.

Una vez introducidos los conceptos generales sobre la memoria virtual, se estudia el mecanismo de paginación, que permite implementar la memoria virtual de una forma muy simple. De hecho, éste es el mecanismo empleado hoy en día de forma predominante.

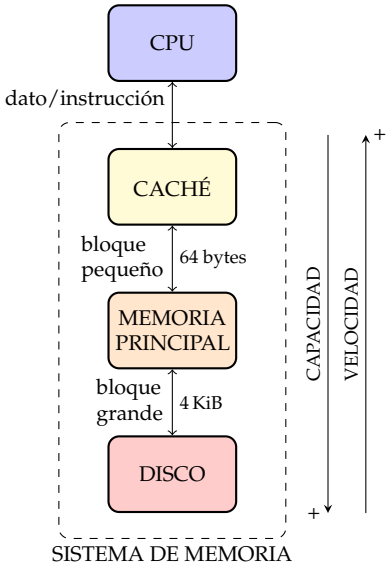


Figura 3.37: Jerarquía de memoria de tres niveles

3.4.1. Introducción

La memoria virtual permite emplear la memoria principal como una caché de los dispositivos de almacenamiento secundario, habitualmente los discos duros (y también las unidades de estado sólido). Estos ocupan el nivel más lejano a la CPU en la jerarquía de memoria, tal como muestra la figura 3.37.

Como se puede observar, los dispositivos de almacenamiento secundario proporcionan el nivel de la jerarquía de memoria de mayor capacidad, pero de menor velocidad. Con este nuevo nivel se consigue incrementar notablemente la capacidad del sistema de memoria vista por la CPU, sin sacrificar la velocidad en la mayor parte de accesos a memoria. Esto último se consigue gracias al principio de localidad visto en la sección 3.2. Sin embargo, la memoria virtual es mucho más que una técnica que incrementa la capacidad de memoria. Proporciona además una rica funcionalidad basada en el concepto de dirección virtual.

Los programas que se ejecutan sobre las CPU que dan soporte a los sistemas operativos multitarea trabajan con direcciones virtuales. Estas direcciones virtuales son traducidas a direcciones físicas por la unidad de gestión de la memoria o MMU (*Memory Management Unit*), y son las que llegan finalmente a las líneas de direcciones del sistema de memoria.

Los dos tipos de direcciones, virtual y física, dan lugar a los espacios de direcciones virtuales y físicas respectivamente. La figura 3.38 representa de forma simplificada el proceso de traducción de las direcciones virtuales a direcciones físicas.

El espacio de direcciones virtuales representa todo el conjunto de direcciones virtuales que se pueden formar. Cada tarea del sistema tiene su propio espacio de direcciones virtuales, independiente de los espacios de direcciones virtuales de otras

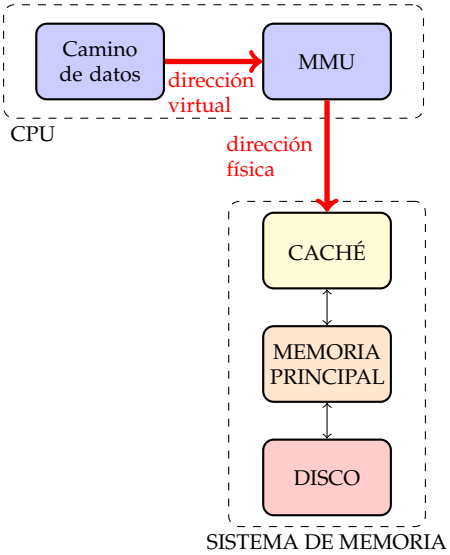


Figura 3.38: Traducción de direcciones virtuales a direcciones físicas

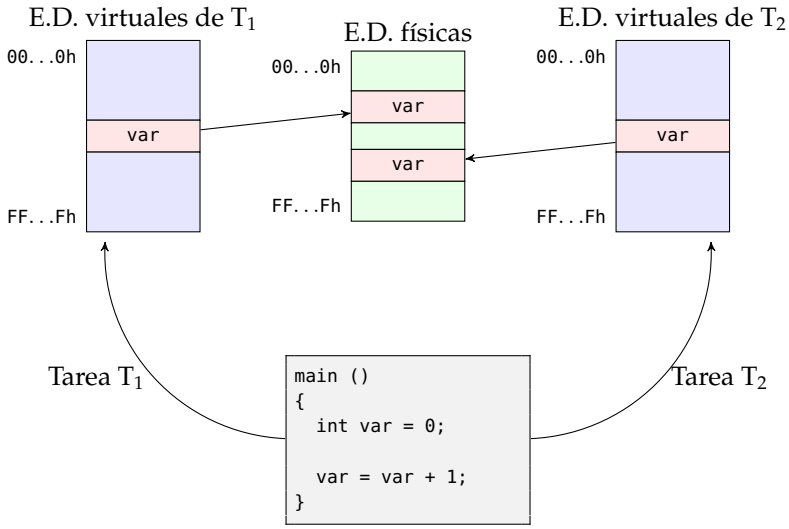


Figura 3.39: Ejemplo que muestra la independencia de diferentes espacios de direcciones virtuales

tareas. Esto significa que una tarea puede leer y escribir en cualquiera de sus direcciones virtuales sin afectar a ninguna otra tarea. La figura 3.39 muestra de forma esquemática los espacios de direcciones correspondientes a dos tareas, T_1 y T_2, que se ejecutan concurrentemente y son instancias del mismo programa.

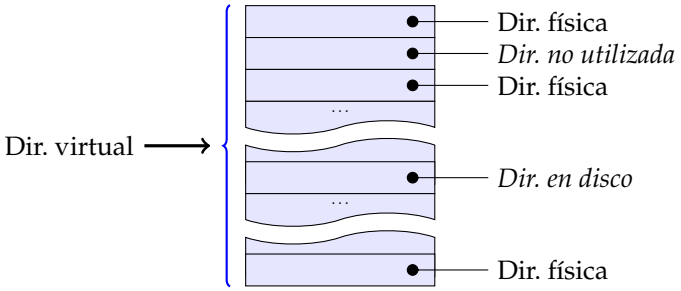


Figura 3.40: Tabla de traducción de una tarea

Las direcciones virtuales de las variables `var` en las tareas T_1 y T_2 son idénticas. Sin embargo, la ejecución de la operación `var = var + 1;` en la tarea T_1 afecta al valor de su variable `var`, pero no a la variable de la tarea T_2. La razón está en que, a pesar de tener las dos variables la misma dirección virtual, tienen diferente dirección física, es decir, diferente dirección de almacenamiento dentro de los dispositivos de memoria. Las direcciones asociadas a los programas por las herramientas de desarrollo (compiladores, enlazadores, etc.) son direcciones virtuales. Sirva como ejemplo la instrucción `ld r3, 80(r7)` de la arquitectura MIPS64, que lleva al registro `r3` la doble palabra ubicada a partir de la dirección virtual `80 + r7`.

El espacio de direcciones físicas representa todo el conjunto de direcciones físicas que se pueden emitir, y es único para todo el sistema. En la práctica, está parcialmente ocupado por dispositivos de memoria. La MMU se encarga de traducir las direcciones virtuales en direcciones físicas apropiadas. Para llevar a cabo la traducción, la MMU dispone de una tabla de traducción de direcciones virtuales a físicas para cada tarea. La figura 3.40 muestra de forma simplificada la tabla de traducción de una tarea. Esta tabla permite no sólo asociar una dirección virtual a una dirección física (dentro de la memoria principal), sino también a una ubicación en el disco, lo que permite ampliar la capacidad de almacenamiento del sistema de memoria por encima de la capacidad de almacenamiento de la memoria principal, tal como se mostrará en el apartado 3.4.2.

3.4.2. La memoria virtual paginada

La asignación de direcciones virtuales a direcciones físicas no se hace dirección a dirección, pues sería muy costoso y poco eficiente (la tabla de traducción sería enorme). En lugar de ello, se divide el espacio de direcciones virtuales en bloques de direcciones virtuales contiguas que se traducen en bloques de direcciones físicas contiguas, tal como se muestra en la figura 3.41.

La técnica de memoria virtual más empleada es la memoria virtual paginada o paginación. Los bloques, denominados páginas, son de tamaño fijo y definido por el hardware.

Usando paginación, una dirección virtual se descompone en dos campos:

- Un número de página virtual.

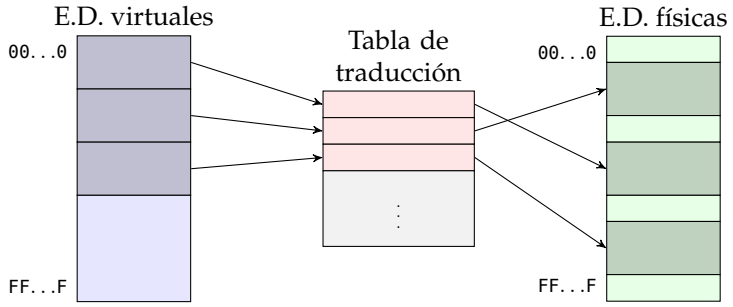


Figura 3.41: Traducción por bloques de las direcciones virtuales a direcciones físicas

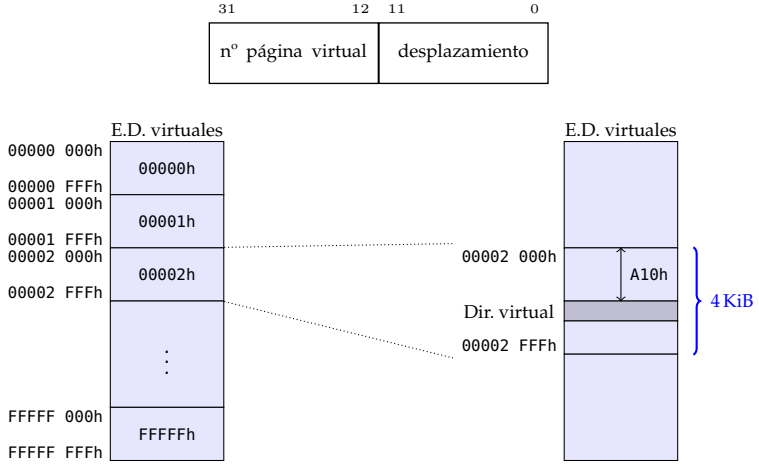


Figura 3.42: Organización del espacio de direcciones virtuales y ubicación de una dirección virtual basada en los campos página y desplazamiento

- Un desplazamiento.

La figura 3.42 muestra a modo de ejemplo una dirección virtual de 32 bits, con 12 bits para el desplazamiento (3 dígitos hexadecimales) y 20 bits para identificar la página virtual (5 dígitos hexadecimales).

El número de bits dedicados al desplazamiento define el tamaño de página. En el caso del ejemplo, el tamaño de la página es de $2^{12} = 4$ KiB, suponiendo un direccionamiento al byte.

El campo página identifica una página de memoria virtual de entre todas las páginas virtuales en que se descompone el espacio de direcciones virtuales. Esto puede verse en la parte izquierda de la figura 3.42. En total hay 2^{20} páginas que van desde la 00000h a la FFFFFh.

El desplazamiento identifica una dirección virtual dentro de una página. De hecho, el desplazamiento sumado a la dirección virtual de comienzo de la página proporciona la dirección virtual.

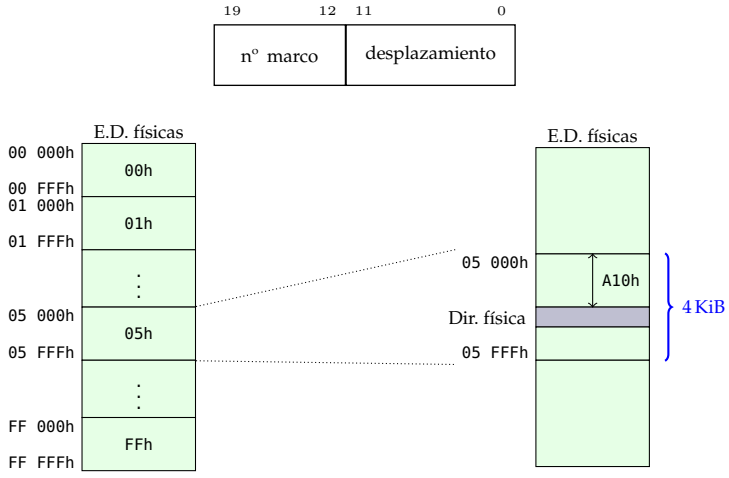


Figura 3.43: Organización del espacio de direcciones físicas y ubicación de una dirección física basada en los campos marco de página y desplazamiento

A modo de ejemplo, la parte derecha de la figura 3.42 muestra la localización de la dirección virtual 0000 2A10h dentro del espacio de direcciones virtuales basada en sus campos número de página virtual y desplazamiento.

Por su parte, las direcciones físicas se descomponen en dos campos:

- Un número de marco de página (o número de página física).
- Un desplazamiento.

La figura 3.43 muestra una dirección física de 20 bits, con 12 bits para el desplazamiento (3 dígitos hexadecimales) y 8 bits para identificar el marco de página (2 dígitos hexadecimales).

El número de bits dedicados al desplazamiento en la dirección física coincide con el número de bits dedicados al desplazamiento en la dirección virtual. Esto no es casual, pues el marco de página es el bloque de memoria física, que debe tener el mismo tamaño que el bloque de memoria virtual (la página).

El campo número de marco de página identifica un marco de página de entre todos los marcos de página en que se descompone el espacio de direcciones físicas. Esto puede verse en la parte izquierda de la figura 3.43. En total hay 2^8 marcos de página que van desde el 00h al FFh.

El desplazamiento identifica una dirección física dentro de un marco de página. De hecho, el desplazamiento sumado a la dirección física de comienzo del marco de página proporciona la dirección física.

La localización de una dirección física dentro del espacio de direcciones físicas basada en los campos número de marco de página y desplazamiento es análoga a la

vista en el caso de una dirección virtual. La parte derecha de la figura 3.43 muestra un ejemplo: dirección 05A10h.

En los apartados siguientes se estudian diferentes aspectos relacionados con la paginación. En primer lugar se muestra la estructura típica de la tabla de traducción: la tabla de páginas. Una vez se conocen los detalles de la tabla de páginas se estudia la configuración de las tablas de páginas para ubicar las tareas y el sistema operativo en los espacios de direcciones virtuales y físicas. Finalmente, se tratan las principales funcionalidades que proporciona la paginación: la protección del sistema de memoria, la compartición de memoria y la ampliación de la capacidad del sistema de memoria.

La tabla de páginas

La tabla de páginas permite convertir las direcciones virtuales en direcciones físicas, asociando páginas virtuales a marcos de página o áreas de disco. Es una estructura de datos asociada a cada tarea, de tal forma que cada tarea tiene su propia tabla de páginas, gestionada por el sistema operativo. La tabla de páginas de una tarea en ejecución está ubicada en memoria física a partir de una posición especificada por un registro de control de la CPU, comúnmente denominado *registro de tabla de páginas*.

La tabla de páginas está formada por tantas entradas como páginas virtuales constituyen el espacio de direcciones virtuales. Por ejemplo, en el caso de las direcciones virtuales presentadas anteriormente, que empleaban 20 bits para el número de página y 12 para el desplazamiento, cada tabla de páginas estará formada por 2^{20} entradas (algo más de 1 millón de entradas). Cada entrada de la tabla de páginas contiene típicamente los siguientes campos:

- Bit de presencia. Indica si la página está en memoria. Cuando este bit está inactivo significa que la página, o bien está en el disco, dentro del archivo de paginación, o bien no tiene asociado almacenamiento. En este último caso la página no puede ser accedida.
- Marco de página/ubicación en disco. Cuando el bit de presencia está activo indica el marco de página asociado a la página virtual. Cuando el bit de presencia está inactivo el sistema operativo le puede dar cualquier significado. Habitualmente proporciona la ubicación del área de disco asociada a la página virtual. Los sistemas operativos disponen de áreas en disco, o en general en dispositivos de almacenamiento secundario, reservadas para implementar la paginación. Por simplicidad, se llamará a estas áreas archivo de paginación[6]. Puede usarse una ubicación en disco no válida para indicar una página virtual que no tiene asociado almacenamiento.
- Bits de protección. Establecen sobre la página virtual restricciones de lectura, escritura, privilegio de acceso, etc.

[6]Windows dispone de uno o más archivos de paginación, mientras que los sistemas operativos Unix disponen habitualmente de una o más particiones para la paginación, conocidas como particiones de *swapping*.

- Bits de estado. Dentro de los bits de estado se encuentra el bit de página escrita o *dirty*, análogo al bit *dirty* de la caché, que se emplea durante el reemplazo de una página virtual en la memoria física. Si la página virtual no ha sido escrita, puede ser eliminada de la memoria sin antes actualizar su contenido dentro del archivo de paginación. Otro de los bits de estado típicos es el de página accedida. Sirve para llevar una contabilidad muy simple de la frecuencia de accesos a las páginas virtuales. Esta información le resulta útil al sistema operativo en el momento de implementar estrategias de reemplazo como la estrategia LRU.

La figura 3.44 muestra un ejemplo de tabla de páginas de una tarea para los espacios de direcciones virtuales y físicas presentados anteriormente. Es decir, páginas de tamaño 4 KiB, 20 bits para identificar la página y 8 bits para identificar el marco de página. Junto con la tabla de páginas se muestra el espacio de direcciones virtuales de la tarea, el espacio de direcciones físicas del sistema y el archivo de paginación. Como puede observarse, se han empleado diferentes patrones de relleno para identificar diferentes tipos de páginas virtuales. Hay páginas virtuales como la 00000h, representadas con un rallado descendente hacia la derecha, que no tienen asociado almacenamiento físico y por lo tanto no pueden leerse ni escribirse. Esto puede comprobarse teniendo en cuenta que el bit de presencia está desactivado y tienen asociada una ubicación en el disco no válida. Otras páginas, como es el caso de la página 00100h, representadas con un relleno sólido, tienen el bit de presencia activado y por lo tanto tienen asociado un marco de página. En este caso se trata del marco 04h. Finalmente, hay páginas representadas con un rallado ascendente hacia la derecha que tienen asociado un bit de presencia desactivado, pero cuyo campo marco de página/ubicación en disco almacena una ubicación dentro del archivo de paginación. Éste es el caso de la página 00101h, que está almacenada en el archivo de paginación en la ubicación `Y`. Para completar la descripción de la tabla de páginas, la figura 3.45 muestra un ejemplo de conversión de una dirección virtual a la dirección física asociada empleando la tabla de páginas anterior.

Ubicación en memoria de las tareas y el SO

Cualquier programa fuente, sea cual sea el lenguaje de programación, estará formado por código y datos. Este programa constituye la entrada para las herramientas de desarrollo como son los compiladores y enlazadores, generándose el código máquina asociado. Este código máquina contiene básicamente la codificación de las instrucciones y datos asociados al programa, así como información adicional que debe gestionar el sistema operativo, como son las inicializaciones de ciertos registros de la CPU. La idea fundamental a tener en cuenta es que las herramientas de desarrollo ubican las instrucciones y los datos sobre el espacio de direcciones virtuales, asociando por tanto páginas virtuales a las instrucciones y datos.

Durante la operación de carga de una tarea para ejecutarla, el sistema operativo asigna almacenamiento físico a las páginas virtuales asociadas a los datos y al código de la tarea. Algunas de las páginas pueden estar en memoria física y otras en el archivo de paginación. Para poder hacer esto, el sistema operativo debe llevar una contabilidad exhaustiva de los huecos disponibles en memoria y en el archivo de paginación. A continuación, el sistema operativo modifica de forma adecuada la tabla

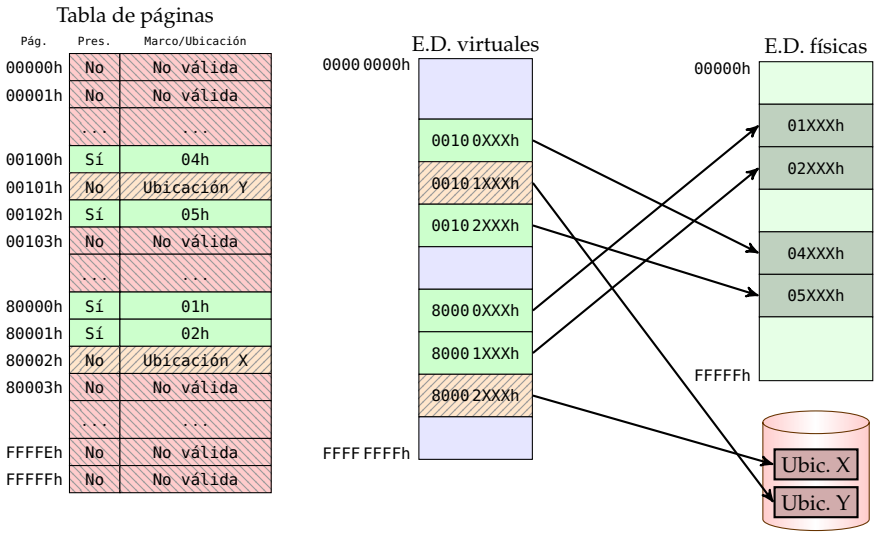


Figura 3.44: La tabla de páginas de una tarea

Dir. virtual		Dir. física
80001 5ABh	$\xrightarrow{\text{traducción}}$	*02 5ABh*

Figura 3.45: Ejemplo de conversión de dirección virtual a dirección física

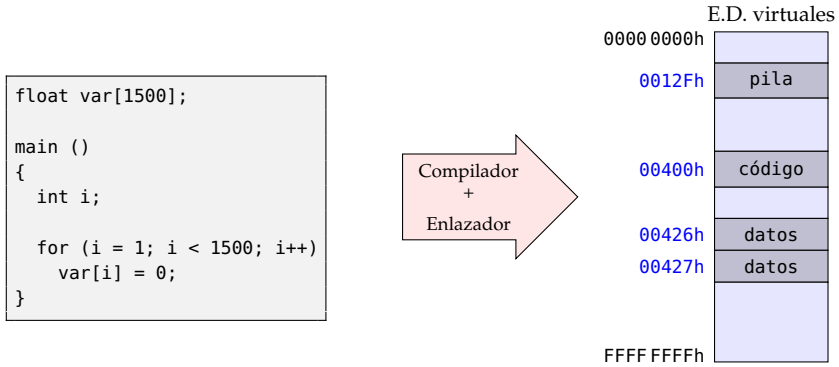


Figura 3.46: Ejemplo de ubicación del código y datos de un programa dentro del espacio de direcciones virtuales de una tarea

de páginas de la tarea reflejando la ubicación de las páginas virtuales de esta última. Debe observarse que este mecanismo permite que las tareas se ejecuten correctamente independientemente de dónde se carguen en memoria, simplificando así la carga de programas por parte del sistema operativo.

Algunos sistemas operativos incluyen mecanismos de seguridad que impiden que las direcciones virtuales sean fijadas en tiempo de compilación y enlazado. En ese caso, se generan direcciones reubicables a las que se les asignan direcciones virtuales y físicas por parte del sistema operativo durante el proceso de carga. De esta forma, las direcciones virtuales y físicas cambian de una ejecución a otra del mismo programa.

El empleo de la paginación simplifica enormemente la labor de las herramientas de desarrollo. La razón es simple, gracias a que cada tarea tiene su propia tabla de traducción, dispone para sí misma de un espacio de direcciones virtuales completo. De esta forma, las herramientas de desarrollo como compiladores y enlazadores pueden generar libremente el código correspondiente a cualquier programa dentro de su espacio de direcciones virtuales, ignorando cualquier otro programa que se ejecute al mismo tiempo.

La figura 3.46 muestra un ejemplo de programa muy simple escrito en C y la asignación de páginas virtuales llevada a cabo por las herramientas de desarrollo dentro de un sistema real.

El programa contiene una variable global de nombre `var`. Esta variable está formada por 1500 números reales de precisión simple, es decir, de 32 bits cada uno. Por lo tanto, son necesarios 6000 bytes para almacenarla. La máquina sobre la que se compiló y enlazó el programa anterior emplea direcciones virtuales de 32 bits y páginas de tamaño 4 KiB. Por lo tanto, son necesarias 2 páginas virtuales para ubicar la variable global. Estas páginas son las páginas número 00426h y 00427h. Por otra parte, dentro de la función `main()` hay una variable local de nombre `i` que está ubicada en la pila. La pila tiene asociada la página virtual 0012Fh, tal como se muestra en la figura. El fragmento de código dentro de la función `main()` tiene asociada la página 00400h. Por último, resaltar que la asignación de páginas virtuales llevada a

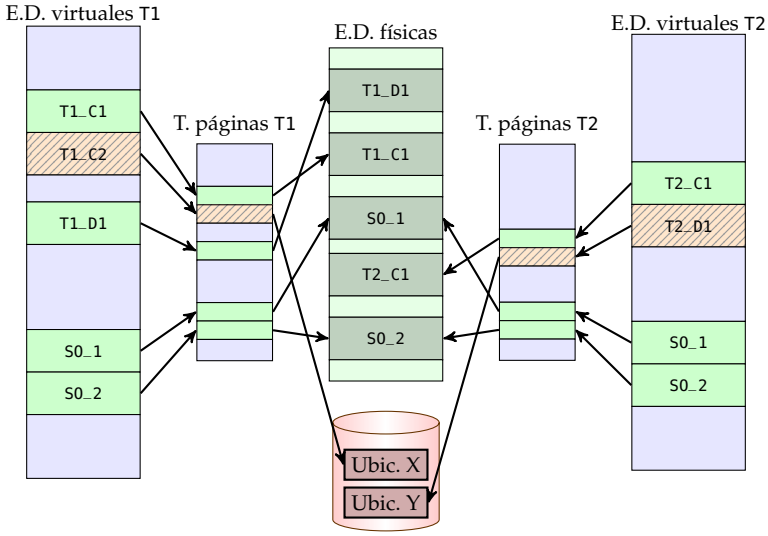


Figura 3.47: Ubicación del sistema operativo dentro de los espacios de direcciones virtuales de las tareas

cabo por las herramientas de desarrollo es un poco más compleja de lo que se muestra. Por ejemplo, la llamada y retorno de la función `main()` requiere un código que no ha sido tenido en cuenta.

Por otra parte, el sistema operativo típicamente se ubica dentro del espacio de direcciones virtuales de todas las tareas. La razón estriba en que cuando se produce una excepción, interrupción o llamada al sistema, es necesario ejecutar código del sistema operativo usando para ello la tabla de páginas actual, que puede ser la de cualquier tarea. La figura 3.47 muestra una situación en la que hay dos tareas `T1` y `T2`, cada una con su tabla de páginas, donde figuran unas páginas virtuales ocupadas por el sistema operativo.

La tarea `T1` tiene asociadas dos páginas virtuales de código (`T1_C1` y `T1_C2`) y una página virtual de datos (`T1_D1`), mientras que la tarea `T2` tiene una página de código (`T2_C1`) y una de datos (`T2_D1`). En la figura puede verse también la asignación de marcos de página y espacio en el archivo de paginación a las páginas virtuales de las tareas. En particular, todas las páginas virtuales tienen asociados marcos de página, excepto las páginas `T1_C2` y `T2_D1`, que están localizadas en el archivo de paginación, en las ubicaciones `X` e `Y` respectivamente.

La figura 3.47 muestra asimismo la ubicación del sistema operativo dentro del espacio de direcciones virtuales de todas las tareas. Puesto que la transferencia de control entre el sistema operativo y una tarea no supone un cambio en el registro de tabla de páginas, la tabla de páginas que usa el sistema operativo es la de la tarea que le cedió el control. Esto implica que la ubicación del sistema operativo en el espacio de direcciones virtuales es la misma para ambas tareas. Como también se puede observar, los marcos de página asociados al sistema operativo están siendo compartidos por las tareas, pues no tiene sentido tener tantas copias del sistema operativo como tareas haya en el sistema.

Protección de memoria

La técnica de memoria virtual paginada permite conseguir de forma muy sencilla la protección de memoria, para lo cual implementa típicamente tres mecanismos:

- Independencia de los espacios de direcciones virtuales de las tareas.
- Tipo de acceso.
- Nivel de privilegio.

El primero de estos mecanismos es la independencia de los espacios de direcciones de las tareas. La figura 3.47 anterior muestra la ocupación de sus espacios de direcciones virtuales por parte de dos tareas `T1` y `T2` y por parte del sistema operativo, así como la ubicación de las páginas virtuales dentro del espacio de direcciones físicas y el archivo de paginación. El sistema operativo programa adecuadamente las tablas de páginas de `T1` y `T2` de tal forma que no haya solapamiento entre los marcos de página asociados a sus páginas virtuales. De igual forma, la ubicación en disco de las páginas virtuales de `T1` y `T2` es diferente, evitando cualquier solapamiento. Por lo tanto, lo que la tarea `T1` lea o escriba sobre sus páginas virtuales `T1_C1`, `T1_C2` y `T1_D1` no afecta a la tarea `T2`. Asimismo, lo que la tarea `T2` lea o escriba sobre sus páginas virtuales `T2_C1` y `T2_D1` no afecta a la tarea `T1`. La propiedad de independencia de los espacios de direcciones virtuales protege a unas tarea de otras y simplifica las herramientas de desarrollo y carga, tal como se había comentado anteriormente.

Otro mecanismo de protección que proporciona la memoria virtual paginada es definir tipos de acceso a las páginas virtuales. Por ejemplo, la entrada de la tabla de páginas asociada a una página virtual de una tarea contiene un campo que especifica si la página virtual puede ser leída, leída y escrita, sólo ejecutada, etc. Esto permite proteger determinadas páginas virtuales frente a accesos inapropiados.

El último mecanismo de protección proporcionado por la paginación consiste en asociar un nivel de privilegio mínimo a cada página virtual. Este nivel de privilegio representa el privilegio mínimo con que debe ejecutarse una instrucción para poder acceder a la página virtual. La mayor parte de las CPU asocian a las páginas virtuales dos posibles niveles de privilegio: un nivel de privilegio bajo para aquellas páginas que pueden ser accedidas por las tareas y el sistema operativo, y un nivel de privilegio alto para aquellas páginas que solo pueden ser accedidas por el sistema operativo.

La figura 3.48 muestra los bits de protección asociados a las páginas virtuales de la tarea `T1` y del sistema operativo.

Como se puede observar, todas las páginas de código están marcadas como de solo lectura, mientras que las de datos están marcadas como de lectura y escritura. Además, las páginas virtuales asociadas al sistema operativo sólo pueden ser accedidas cuando la CPU se encuentra ejecutando con el nivel de privilegio de supervisor, es decir, mientras ejecuta el sistema operativo.

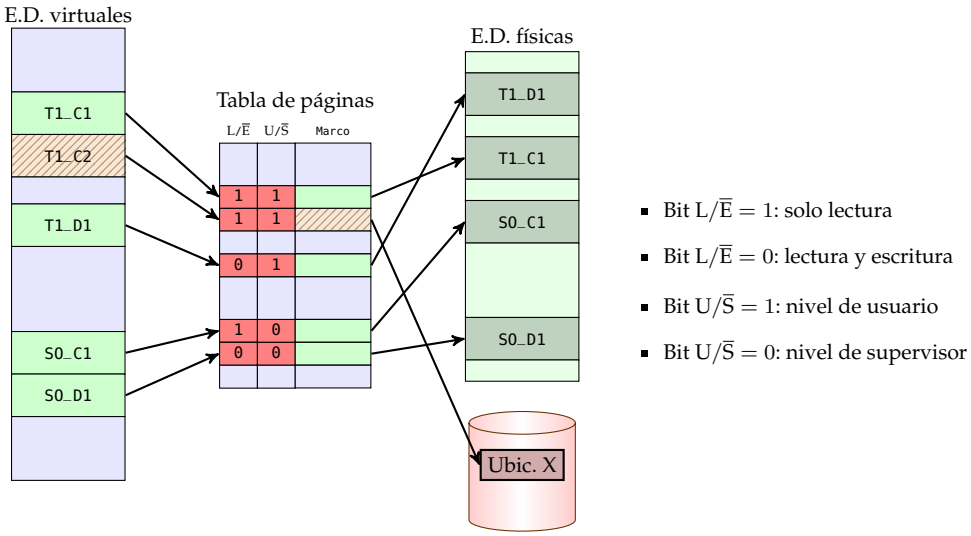


Figura 3.48: Bits de protección asociados a las páginas de una tarea y del sistema operativo

Compartición de memoria

Otro aspecto importante de la memoria virtual paginada es la posibilidad de compartir memoria de forma muy simple.

El sistema operativo posibilita la compartición de memoria entre tareas programando adecuadamente las tablas de páginas. La compartición de memoria permite comunicar a gran velocidad dos tareas o evitar duplicidades en memoria física. Por ejemplo, tal como se observa en la figura 3.47 anterior, todas las tareas comparten la memoria física asociada al sistema operativo, evitando que tenga que haber tantas copias del sistema operativo como tareas haya en el sistema. Otro ejemplo en el que se evita la duplicidad es mediante el uso de bibliotecas de enlace dinámico. Si dos o más tareas utilizan la misma biblioteca, no es necesario que cada una tenga su copia en la memoria física, sino que es suficiente con una instancia de dicha biblioteca en memoria física compartida por todas las tareas.

La figura 3.49 muestra cómo se pueden crear áreas de memoria compartida durante la ejecución de los programas. En un momento dado, la tarea `T1` solicita al sistema operativo[7] un marco de página asociado a su página virtual `T1_D2` para ser compartido. Suponiendo que el sistema operativo concede la petición, éste programa adecuadamente la entrada asociada de la tabla de páginas de `T1` para que apunte al marco de página concedido. A partir de ese momento, la tarea `T1` puede leer y escribir en su página virtual `T1_D2`, pues tiene asociado almacenamiento físico. Más tarde, la tarea `T2` solicita al sistema operativo la compartición del marco de página anterior y que sea asociado a su página virtual `T2_D2`. Una vez el sistema operativo

[7] Esto se hace mediante la llamada a servicios específicos del sistema operativo.

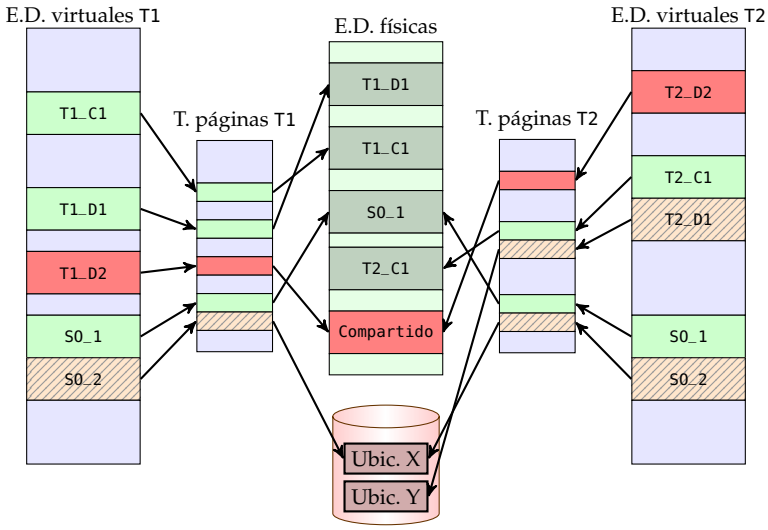


Figura 3.49: Compartición de una página de memoria entre dos tareas

conceda esta petición, el marco de página pasa a estar compartido por `T1` y `T2`. Esto significa que lo que `T1` escriba en su página virtual `T1_D2` puede ser leído por `T2` a través de su página virtual `T2_D2` y viceversa.

Ampliación de la capacidad de memoria

El objetivo es disponer de un sistema de memoria de gran capacidad para:

- Poder ejecutar programas de mayor tamaño que la memoria principal instalada.
- Poder ejecutar muchos programas al mismo tiempo que ocuparían más memoria que la memoria principal instalada.

Esta ampliación de la capacidad de memoria se consigue empleando la memoria principal como una caché del disco. Cuando la CPU emite una dirección virtual, el campo número de página virtual se emplea como índice dentro de la tabla de páginas de la tarea. Si dicha página se corresponde con un marco de memoria física, la MMU genera la dirección física correspondiente a la dirección virtual y se accede directamente al dato en la memoria principal. Por el contrario, si la tabla de páginas indica que la página tiene asociada una ubicación dentro del disco, se produce una excepción, denominada fallo de página. Como consecuencia, se ejecuta el manejador de la excepción, que forma parte del sistema operativo, y se mueve la página entre el disco y la memoria principal. Finalmente, el sistema operativo reprograma la entrada apropiada de la tabla de páginas para reflejar la nueva situación. Ahora, la página está accesible en la memoria principal y se repite el acceso, pero en este caso ya no se produce la excepción.

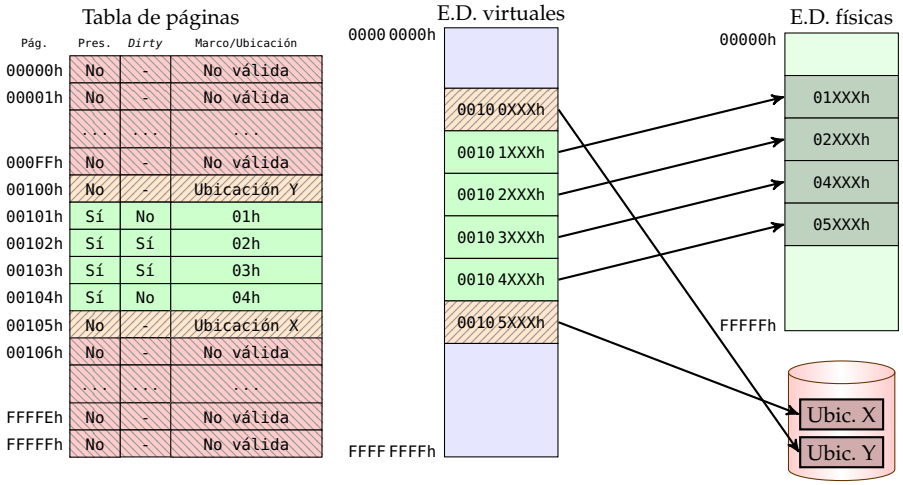


Figura 3.50: Situación inicial antes del fallo de página

Aunque la paginación permite ampliar la capacidad del sistema de memoria por encima de la capacidad de la memoria principal, esa capacidad extra se obtiene a costa de reducir el rendimiento del sistema. Durante los fallos de página es necesario mover bloques de disco a memoria principal, lo que requiere tiempos muy elevados con respecto a los tiempos de acceso a la memoria principal. A continuación se describe con mayor detalle la secuencia de operaciones que ocurren durante un fallo de página.

El fallo de página ocurre cuando la CPU emite una dirección virtual perteneciente a una página no ubicada en memoria física. Esta situación es detectada fácilmente por el hardware de la CPU, pues en ese caso el bit de presencia está desactivado. Cuando ocurre esto, la CPU genera una excepción de fallo de página, que es gestionada por el manejador de fallo de página que forma parte del sistema operativo.

Para ilustrar los pasos seguidos por el manejador de la excepción de fallo de página se asume la situación inicial definida en la figura 3.50. A continuación se enumeran los pasos que sigue el manejador.

A) Se analiza la causa del fallo de página. Hay dos posibles causas: dirección en página virtual sin almacenamiento asignado o dirección en página almacenada en disco.

B) Una dirección en una página virtual sin almacenamiento asignado implica que no se encuentra ni en memoria física ni en el archivo de paginación. En este caso, se produce una excepción de violación de acceso a memoria. Por ejemplo, al intentar acceder a la dirección 000F F05Eh (página 000FFh) usando la tabla de páginas de la figura 3.50.

C) Si la dirección está almacenada en disco, como por ejemplo la dirección 0010 51B3h (página 00105h), se llevan a cabo las siguientes operaciones:

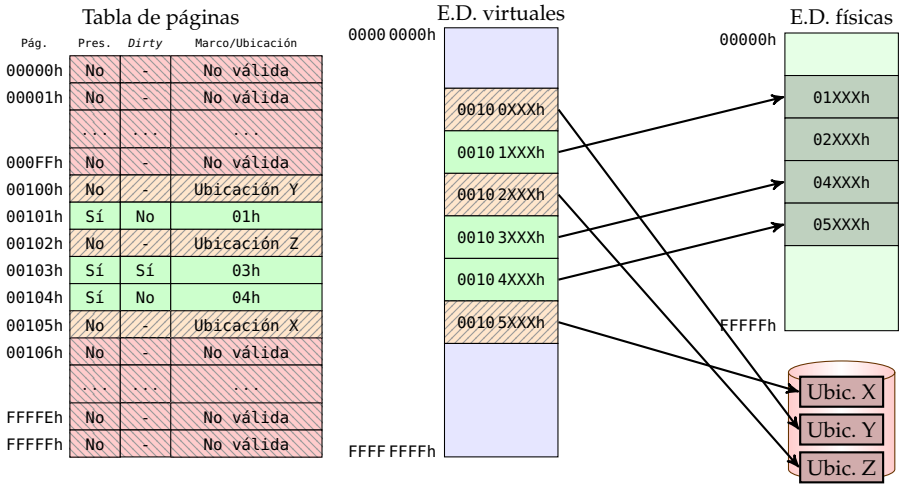


Figura 3.51: Movimiento de la página 00102h modificada al disco antes de ser reemplazada

C1) El sistema operativo determina en qué marco de página se va a cargar la página de disco. Para ello se emplea un algoritmo de reemplazo (por ejemplo del tipo LRU), de forma análoga a como ocurría con la memoria caché. El algoritmo puede elegir entre uno de los cuatro marcos de página asignados a la tarea o bien asignarle un marco que anteriormente no le pertenecía. Al conjunto de marcos de página asignados a una tarea se le denomina *working set*. Se elige el marco de página 02h en el ejemplo.

C2) Si la página a reemplazar refleja cambio (bit *dirty* activado), como ocurre en el ejemplo (página 00102h), se guarda en el disco antes de ser reemplazada y se actualiza la entrada de la tabla de páginas asociada. La nueva situación se muestra en la figura 3.51.

C3) Se carga desde disco la página solicitada y se actualiza la entrada correspondiente de la tabla de páginas. La nueva situación se muestra en la figura 3.52.

C4) El manejador de fallo de página retorna a la instrucción causante de la excepción una vez que la dirección a la que se desea acceder está en memoria física. Se ejecuta de nuevo esta instrucción, pero esta vez sin causar fallo de página.

3.4.3. El TLB

Hasta ahora, se han tratado aspectos relacionados con la funcionalidad de la memoria virtual, y en particular de la memoria virtual paginada. No obstante, el uso de esta técnica tiene importantes implicaciones en el rendimiento de la máquina que es necesario tener en cuenta.

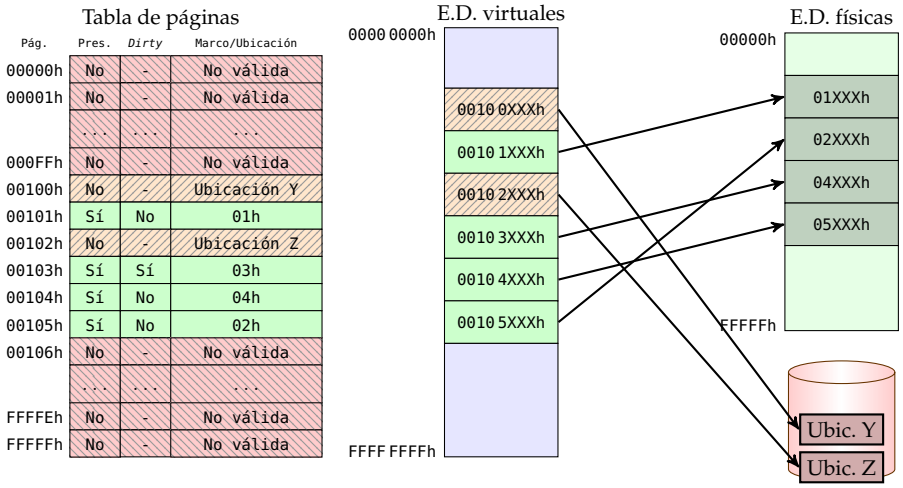


Figura 3.52: Carga desde el disco de la página 00105h

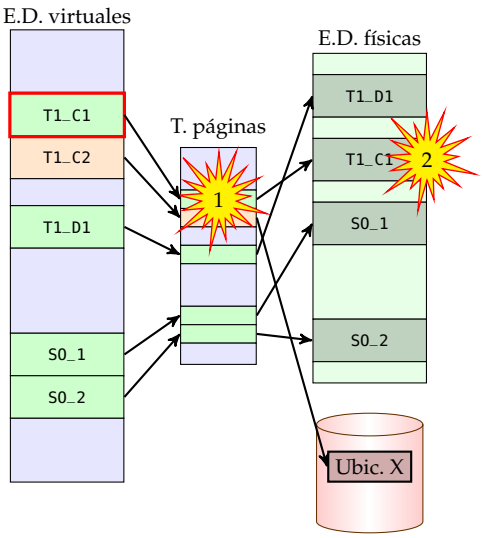


Figura 3.53: Acceso a memoria empleando paginación sin TLB

Con el empleo de la memoria virtual paginada, cada acceso a una dirección virtual por parte de la CPU supone en realidad dos accesos a la memoria: uno a la tabla de páginas y otro a la dirección física asociada. La figura 3.53 ilustra el problema.

La tarea `T1` accede a un byte de memoria dentro de la página virtual `T1_C1`, lo que requiere:

1. Acceder a la memoria física para leer la entrada de la tabla de páginas asociada a la página virtual `T1_C1`.
2. Acceder a la dirección física resultado de traducir la dirección virtual usando la entrada en la tabla de páginas.

Lógicamente esto reduce de forma considerable el rendimiento del sistema. Si se supone de momento que no existe memoria caché, cada acceso a una dirección virtual supondría dos accesos a la memoria principal.

Debido al principio de localidad, una tarea accede habitualmente a unas pocas páginas virtuales, cuyas entradas pueden almacenarse en una pequeña memoria caché específica dentro de la CPU. Esta caché suele ser totalmente asociativa y se denomina *Translation Lookaside Buffer* o TLB. De esta forma, el acceso a una dirección virtual se reduce a un acceso al TLB y a continuación un acceso a la memoria principal. Puesto que el tiempo de acceso al TLB es sensiblemente menor que el de acceso a la memoria principal, éste podría despreciarse. Por consiguiente, el uso de la paginación, en principio, no afectaría apenas al rendimiento del sistema cuando se usa TLB.

La gestión de los fallos de TLB suele hacerse por hardware al igual que ocurre con las cachés de propósito general. No obstante las diferentes arquitecturas suelen proporcionar instrucciones privilegiadas relacionadas con el TLB, por ejemplo para invalidar todas las entradas del TLB o alguna entrada en particular.

Cada vez que se produce un cambio de contexto, al pasar de ejecutar una tarea a ejecutar otra, cambia el contenido del registro de tabla de páginas y cambia en consecuencia la tabla de páginas. Por lo tanto, el sistema operativo debería marcar como inválidas todas las entradas del TLB. Resulta entonces que con cada cambio de contexto se producen varios fallos de TLB consecutivos (al menos uno para el área de datos, otro para el de código y otro para el de pila), lo que ralentiza la reanudación de los programas tras dicho cambio de contexto. La solución comúnmente empleada es añadir un conjunto de bits a cada una de las entradas del TLB que identifican la tarea a la que pertenece la entrada. Puede reservarse además una secuencia especial de estos bits para marcar ciertas entradas como globales, es decir, válidas para todas las tareas. Estas entradas globales podrían usarse para traducir las direcciones virtuales asociadas al sistema operativo, pues sus direcciones de memoria no cambian al cambiar de tarea. Por supuesto, al usar un único TLB para todas las tareas, se requiere que éste tenga una capacidad mucho mayor.

Al introducir el concepto de TLB se ha ignorado la presencia de la memoria caché entre la CPU y la memoria principal de forma deliberada para simplificar la exposición. Sin embargo, la coexistencia del TLB y la caché de propósito general tiene grandes implicaciones desde el punto de vista del rendimiento. El tiempo de acceso al TLB deja de ser despreciable frente al tiempo de acceso a memoria, pues el tiempo

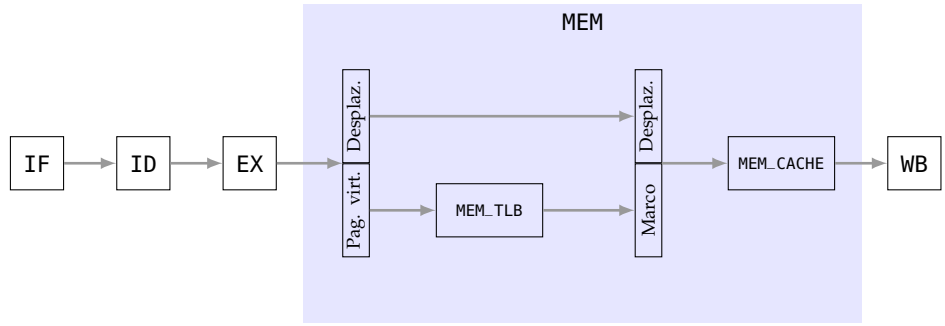


Figura 3.54: Segmentación en el acceso al TLB y a la caché

de acceso a memoria es ahora, en muchas ocasiones, el tiempo de acceso a caché y es similar al tiempo de acceso al TLB. Por tanto, un acceso a memoria pasa de un acceso a caché a un acceso al TLB más otro a la caché, es decir, equivalente a dos accesos a caché, lo que supone una importante penalización en el rendimiento. Puesto que esta penalización en el rendimiento es inasumible, se proponen a continuación diferentes soluciones.

La figura 3.54 ilustra una posible solución al problema de redimiento anterior a partir de una CPU segmentada análoga a la vista en el capítulo dedicado a la CPU. La etapa de memoria `MEM` original se segmenta en dos etapas, `MEM_TLB` y `MEM_CACHE`, cada una de las cuales requiere un ciclo de reloj. La primera se encarga de obtener el marco de página a partir de la página virtual y la segunda se encarga de acceder a la caché usando la dirección física obtenida.

En condiciones ideales, gracias a la segmentación de los accesos a memoria, las etapas `MEM_TLB` y `MEM_CACHE` pueden trabajar en paralelo, por lo que el tiempo de acceso a memoria será de un ciclo de reloj. Sin embargo, la efectividad de esta solución es limitada, ya que para ser efectiva requeriría muchas instrucciones de acceso a memoria consecutivas. Por esta razón, la solución empleada en la práctica son las cachés virtuales, que se introducen a continuación.

Cachés virtuales

Las cachés virtuales son cachés que emplean parcial o totalmente direcciones virtuales para su funcionamiento. La idea fundamental es la siguiente: si la caché trabajase con direcciones virtuales, se podría acceder a la misma sin tener que esperar a que el TLB proporcione el marco de página asociado a la página virtual.

Antes de seguir profundizando en el estudio de las cachés virtuales conviene recordar que cualquier bloque de memoria que se encuentre cacheado es identificado por dos elementos:

- Conjunto (o índice). Las líneas de caché se organizan en conjuntos, de tal forma que un bloque de memoria puede ubicarse en una línea cualquiera dentro del conjunto que le corresponde. Al número de líneas o vías del conjunto se le denomina comúnmente grado de asociatividad. Las correspondencias directa

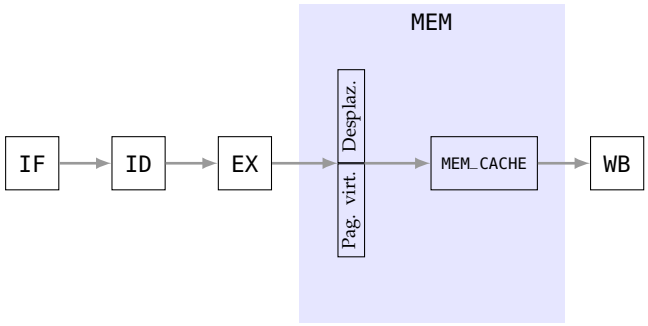


Figura 3.55: Caché indexada y etiquetada virtualmente (V/V)

y totalmente asociativa son casos particulares de la correpondencia asociativa por conjuntos.

- Etiqueta. Permite identificar un bloque de memoria de entre todos los que pueden cachearse dentro de un conjunto dado.

Las cachés con las que se ha trabajado hasta ahora se denominan cachés indexadas y etiquetadas físicamente (*Physically indexed/Physically tagged*), y se representan como P/P. Tanto el conjunto de caché como la etiqueta provenían de la dirección física, por esta razón era necesario acceder al TLB antes de acceder a la caché. Tal como se mostró anteriormente, este tipo de cachés acarrean una pérdida de rendimiento, incluso segmentando el acceso al TLB y a la caché. No obstante, existen otras alternativas. Estas alternativas se aplican siempre a la caché L1, pues al ser el nivel de caché más cercano a la CPU es el que presenta el problema de rendimiento por la traducción de direcciones. Los niveles de caché L2, L3 y sucesivos no presentan este problema, pues trabajan siempre con direcciones físicas, de ahí que sean ignorados al tratar la traducción de direcciones.

Las cachés V/V son cachés indexadas y etiquetadas virtualmente (*Virtually indexed/Virtually tagged*). Estas cachés trabajan exclusivamente con direcciones virtuales, por lo que para acceder a la caché no es necesario esperar por el TLB. La figura 3.55 muestra el esquema de funcionamiento de una caché L1 indexada y etiquetada virtualmente.

Cuando la dirección virtual a la que se desea acceder está cacheada, la caché L1 sirve directamente el dato, necesitando un ciclo de reloj. Por lo tanto, en caso de acierto de caché el tiempo de acceso a memoria es de un ciclo de reloj y no dos, solucionando así el problema de rendimento. Cuando se produce un fallo de caché es necesario acceder al TLB para traducir la dirección virtual en física para acceder a la caché L2, que trabaja con direcciones físicas, y así satisfacer el fallo de caché L1.

Aunque el empleo de las cachés L1 de tipo V/V soluciona el problema de rendimiento introduce varios problemas adicionales.

- Para que la caché pueda cachear datos de diferentes tareas es necesario que cada línea incorpore un identificador de tarea, de forma análoga al TLB con las cachés físicas. En caso contrario, debería invalidarse la caché con cada cambio

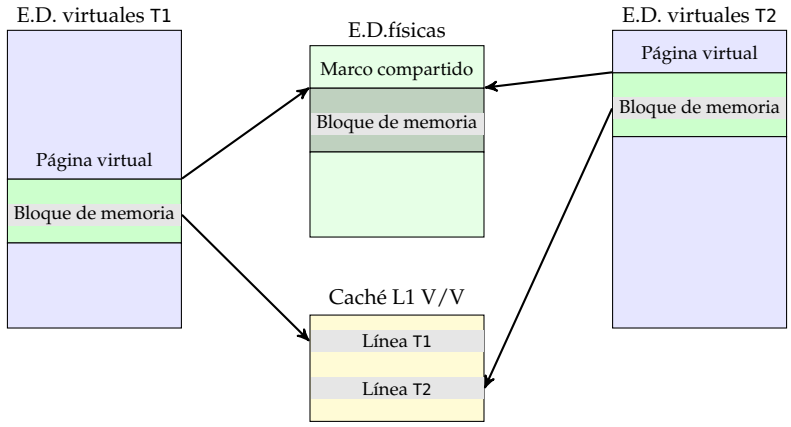


Figura 3.56: El problema de los sinónimos en las cachés virtuales

de contexto, dado que una misma dirección virtual puede apuntar a diferentes direcciones físicas en dos tareas distintas.

- La caché debe gestionar la protección en el acceso a memoria. Cuando se produce un acierto de caché no se atraviesa el TLB, por lo que la caché debe hacer esta gestión. Para ello, cada línea debe incluir bits para indicar el tipo de acceso ($L/\overline{E}$) y el nivel requerido ($U/\overline{S}$) coherentes con los bits asociados a la página virtual en la tabla de páginas.

- Aparece el problema de los sinónimos. Este problema requiere especial atención y se ilustra en la figura 3.56.

  Las tareas T1 y T2 comparten un marco de memoria a través de páginas virtuales diferentes. Puesto que las direcciones virtuales del área compartida son diferentes para cada tarea, un mismo bloque de memoria dentro de ese área puede aparecer en dos conjuntos diferentes a la vez, uno con el identificador de tarea T1 y otro con el identificador de tarea T2. Esto es posible dado que las direcciones virtuales son diferentes. Además del desperdicio de caché, pues un mismo bloque de memoria aparece cacheado dos veces, puede aparecer un problema de coherencia. Si la tarea T1 escribe en el bloque de memoria compartido, escribirá en la línea de caché con el identificador T1. Si a continuación T2 lee el contenido del bloque compartido, leerá la línea de caché con el identificador T2, por lo que no leerá el valor actualizado y de ahí el problema de coherencia.

  Una solución para este problema es invalidar la caché L1 (actualizando previamente los bloques modificados en la caché L2) con cada cambio de contexto. Otra solución es buscar sinónimos en toda la caché L1 con cada escritura e invalidar las líneas con sinónimos. En ambos casos el coste en términos de rendimiento es inaceptable.

Como se ha visto hasta ahora, tanto las cachés físicas (P/P) como las virtuales (V/V) presentan problemas de rendimiento. En la práctica la solución empleada con-

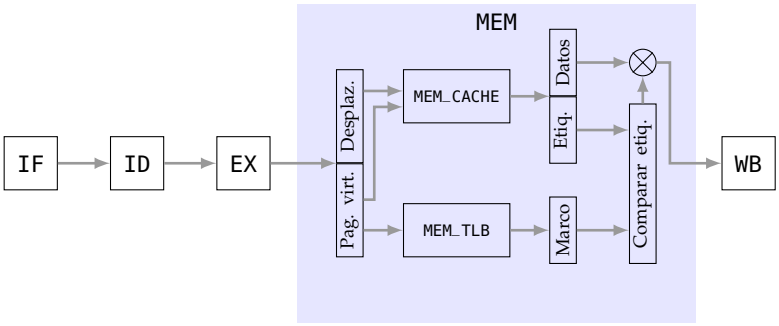


Figura 3.57: Caché indexada virtualmente y etiquetada físicamente (V/P)

siste en mezclar ambas estrategias empleando una caché L1 indexada virtualmente y etiquetada físicamente, denominada V/P (*Virtually indexed/Physically tagged*). La figura 3.57 muestra su principio de funcionamiento.

La caché se indexa virtualmente, lo que significa que se emplea la dirección virtual para seleccionar el conjunto de caché en el que la dirección virtual puede estar cacheada. La novedad respecto a la caché V/V es que ahora el campo etiqueta de una línea de caché no proviene de la dirección virtual, sino de la dirección física. Por lo tanto, para comprobar si una dirección virtual está cacheada debe compararse la etiqueta almacenada en cada vía del conjunto con la etiqueta obtenida a partir de la dirección física. Para obtener la etiqueta de la dirección física se emplea el TLB, al que se accede en paralelo a la caché usando la página virtual como de costumbre, proporcionando el marco de página a partir del que se obtiene la etiqueta de la dirección física. Al trabajar en paralelo el TLB y la caché no se incurre en ninguna penalización de rendimiento; el tiempo de acceso a memoria es de un ciclo de reloj.

Las cachés V/P tienen la ventaja respecto a las cachés V/V de que todos los accesos a memoria requieren el uso del TLB, por lo que no es necesario implementar la gestión de privilegios u otras restricciones de acceso en la caché. No obstante, en general comparten el problema de los sinónimos con las cachés V/V. Puede ocurrir que el mismo bloque de memoria dentro de un marco compartido por dos tareas `T1` y `T2` sea cacheado en conjuntos diferentes para una y otra tarea al estar la caché indexada virtualmente. Sin embargo las soluciones a este problema son mucho más simples y eficientes que para el caso de las cachés V/V.

Aunque existen varias posibles soluciones, la más común es limitar el tamaño de la caché L1 y/o incrementar su asociatividad (número de vías). Si el tamaño de la página multiplicado por el número de vías de la caché es mayor o igual que el tamaño de la caché, el conjunto en el que se cachea la dirección virtual se puede obtener a partir del desplazamiento de la dirección virtual, que coincide con el de la dirección física, y por lo tanto no depende del número de página virtual. Dicho de otra forma, si varias tareas comparten un bloque de memoria, este bloque de memoria se cacheará en el mismo conjunto sea cual sea la tarea, por lo que no pueden existir sinónimos (duplicados). Esta solución es la elegida en muchos procesadores para PC de Intel y AMD. Por ejemplo, cada núcleo del procesador Intel Core i5-6600K tiene una caché L1 de datos de 32 KiB, 8 vías por conjunto y el tamaño mínimo de

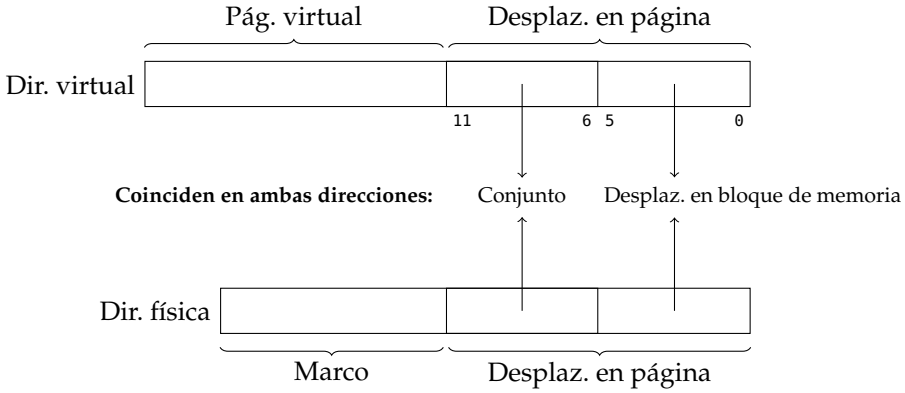


Figura 3.58: Obtención del conjunto de caché a partir del desplazamiento de la dirección física

página es de 4 KiB. La figura 3.58 muestra en este ejemplo cómo se puede obtener el conjunto de caché directamente del desplazamiento de la dirección, por lo que no depende de la página virtual. La situación es análoga a quitar la flecha que va de la página virtual a la etapa `MEM_CACHE` en la figura 3.57.

Multiplicando el tamaño de la página por el número de vías de la caché se obtiene 4 KiB $\times$ 8 = 32 KiB, justo el tamaño de la caché. El problema de esta estrategia es el incremento de coste de la caché L1, que crece rápidamente con su grado de asociatividad. No obstante, este coste es asumible con las tecnologías de fabricación actuales.

3.5. Soporte a la virtualización

La memoria virtual que se acaba de estudiar es un ejemplo de virtualización en el que un recurso físico único, el espacio de direcciones físicas, se muestra como múltiples espacios de direcciones virtuales, uno para cada tarea. Cada tarea percibe la «ilusión» de que dispone de un espacio de direcciones completo para ella sola, cuando en realidad está compartiendo un espacio de direcciones físicas con otras tareas. La MMU con la gestión adecuada por parte del sistema operativo construye esa «ilusión».

Cuando se utilizan máquinas virtuales se requiere un nuevo nivel de virtualización, ya que hay varios sistemas operativos que desean utilizar al mismo tiempo un único espacio de direcciones físicas que forzosamente deben compartir. La compartición del espacio de direcciones físicas entre los sistemas operativos es gestionada por el hipervisor, para lo cual necesita introducir un nivel de virtualización adicional entre la memoria virtual (gestionada por el sistema operativo) y la memoria física (gestionada por el hipervisor). Se definen por tanto tres espacios de direcciones:

- Direcciones virtuales del invitado (*Guest Virtual Addresses* o GVA). Son las que percibe una tarea.

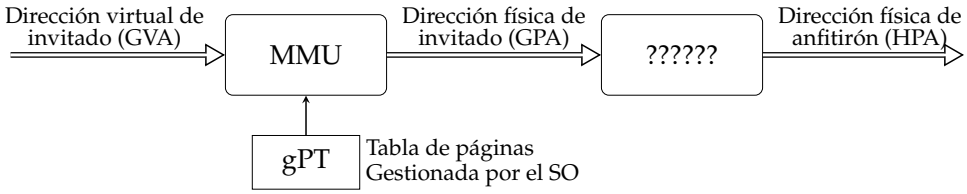


Figura 3.59: Traducción entre los tres espacios de direcciones

- Direcciones físicas del invitado (*Guest Physical Addresses* o GPA). No son direcciones físicas reales. Son gestionadas por el sistema operativo dentro de la memoria asignada a la máquina virtual.
- Direcciones físicas del anfitrión (*Host Physical Addresses* o HPA). Estas son las direcciones físicas reales del sistema, gestionadas por el hipervisor.

Por lo tanto, para traducir la dirección virtual de una tarea a una dirección física del sistema son necesarias dos traducciones, como se muestra en la figura 3.59:

- Dirección virtual de invitado (GVA) → Dirección física de invitado (GPA). Esta traducción la realiza la MMU empleando la tabla de páginas gestionada por el sistema operativo tal como se ilustró en la sección 3.4.2. Esta tabla de páginas se denomina tabla de páginas de invitado (*guest Physical Table* o gPT) en este contexto.
- Dirección física de invitado (GPA) → Dirección física de anfitrión (HPA). Esta traducción plantea un problema, pues la CPU no dispone de ningún dispositivo que pueda realizarla.

A continuación se presentan las soluciones empleadas en la arquitectura x86.

Una primera solución al problema de la traducción de direcciones físicas de invitado a direcciones físicas de anfitrión consiste en el empleo de la técnica de las tablas de páginas en la sombra (*shadow page tables*). Cada tarea tiene asociada una tabla denominada tabla de páginas en la sombra (*shadow Page Table* o sPT) gestionada por el hipervisor que contiene la traducción directa entre una dirección virtual de la tarea (GVA) y la dirección física del sistema (HPA) correspondiente. La MMU por tanto trabaja con la tabla de páginas en la sombra (sPT) y no con la tabla de páginas de invitado (gPT), tal como se muestra en la figura 3.60.

Para que el sistema operativo invitado tenga la ilusión de que está administrando el espacio de direcciones físicas de la máquina, el hipervisor intercepta cualquier escritura del sistema operativo sobre las tablas de páginas de invitado (gPT), por ejemplo marcando todas las entradas como de sólo lectura. De esta forma, en el momento en el que el sistema operativo intente modificar una entrada se producirá una excepción de fallo de página. El manejador de la excepción de fallo de página ejecutado por el hipervisor modificará la entrada correspondiente de la tabla de páginas en la sombra. Asimismo, también es necesario que el hipervisor intercepte todas aquellas instrucciones de gestión del TLB que ejecutan los sistemas operativos, como por ejemplo las instrucciones de invalidación de entradas del TLB.

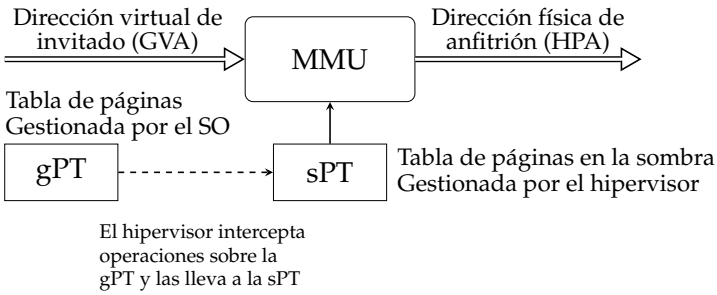


Figura 3.60: Traducción empleando tablas de páginas en la sombra

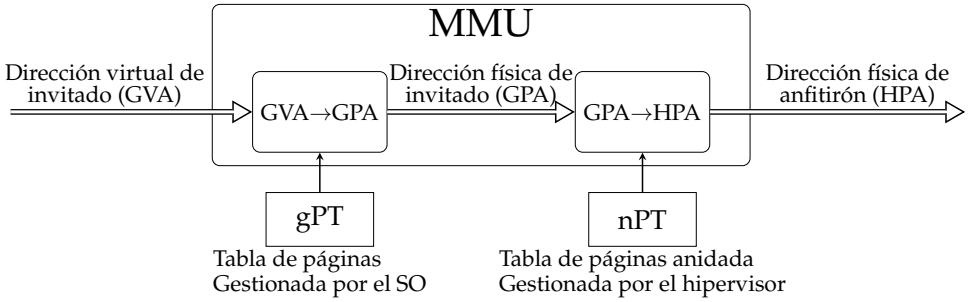


Figura 3.61: Traducción empleando paginación anidada

Uno de los problemas del empleo de tablas de páginas en la sombra es el elevado número de cambios de contexto que ocurren entre los sistemas operativos y el hipervisor para la gestión de la paginación, penalizando el rendimiento. Para paliar esta reducción de rendimiento, los fabricantes de CPU x86, como parte del soporte a la virtualización por hardware, han desarrollado la técnica denominada traducción de segundo nivel (*Second Level Address Translation* o SLAT), también conocida como paginación anidada (*nested paging*).

Con la paginación anidada cuando una tarea accede a una dirección virtual (GVA), la MMU en una primera fase la traduce en una dirección física de invitado (GPA) empleando su tabla gPT, gestionada por el sistema operativo anfitrión o el hipervisor. El sistema operativo anfitrión puede escribir en la tabla y realizar las operaciones de gestión del TLB habituales sin interferencia por parte del hipervisor.

La MMU en una segunda fase traduce la dirección resultante en una dirección física del sistema (HPA) empleando una segunda tabla de traducción, denominada tabla de páginas anidada (*nested Page Table* o nPT). Esta segunda tabla está bajo el control del hipervisor. La figura 3.61 esquematiza la traducción de direcciones de una tarea empleando la paginación anidada.

Empleando paginación anidada el TLB almacena la traducción directa entre las direcciones virtuales de invitado (GVA) y las direcciones físicas del anfitrión (HPA) durante la ejecución del sistema operativo y sus tareas.

La paginación anidada no siempre resulta en un mejor rendimiento respecto al empleo de tablas de páginas en la sombra. El coste de un fallo de TLB es mayor

empleando paginación anidada que tablas de páginas en la sombra pues es necesario leer de la memoria el doble de entradas de tabla de páginas: una entrada de la gPT y otra de la nPT.

Capítulo 4

El sistema de E/S

La definición de la arquitectura del computador se completa con el tercer bloque, conocido como sistema de E/S. Este bloque de la arquitectura es parte fundamental en todo computador, ya que es el encargado de comunicar el computador con su entorno. A diferencia de los bloques de la arquitectura anteriores (CPU y memoria), el sistema de E/S es bastante heterogéneo debido a que la comunicación con el entorno puede implicar distintos tipos de interlocutores (personas u otros sistemas) y además, cada interlocutor puede llevar a cabo comunicaciones de distintas características (sonido, imágenes, texto impreso, etc.).

La comunicación del computador con su entorno normalmente implica a tres elementos:

- Los periféricos. Son los dispositivos que realizan la comunicación directa con el entorno: pantalla, teclado, ratón, dispositivos de comunicaciones, etc.
- Las interfaces. Son los componentes del computador encargados de realizar la adaptación entre la información binaria que maneja el computador y el tipo de señales que utiliza el periférico.
- El sistema de interconexión. Recibe este nombre el conjunto de tecnologías y dispositivos que se utilizan para mover la información. Su uso no es exclusivo de la comunicación con el entorno (la comunicación entre la memoria y la CPU también se hace a través de él), pero sí es el responsable de la diversidad de las tecnologías existentes para adaptarse a los requerimientos de la comunicación con los periféricos.

Los tres elementos anteriores participan en la comunicación del computador con el entorno, si bien en la práctica suelen estudiarse de forma independiente. Se comenzarán estudiando las interfaces de E/S, posteriormente se presentará el sistema de interconexión y por último se hará una breve reseña de los periféricos.

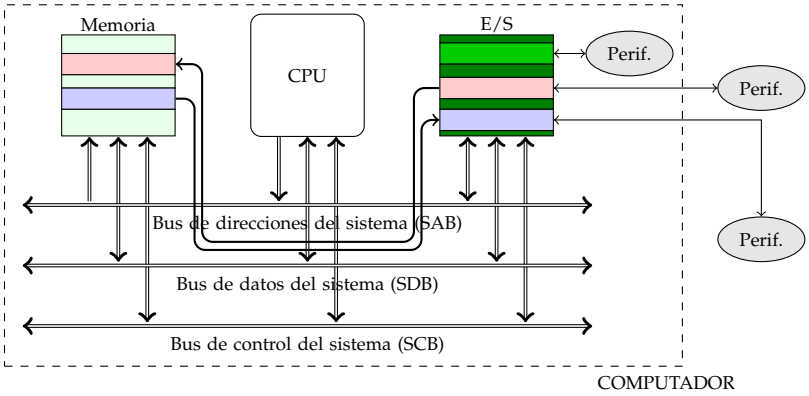


Figura 4.1: Operaciones de entrada y salida dentro del computador von Neumann

4.1. Interfaces de E/S

Forman parte del sistema de E/S las interfaces de todos los periféricos del sistema. Habitualmente, cada periférico tiene una interfaz, aunque puede darse el caso de que una misma interfaz gestione varios periféricos[1].

En general, cualquier operación de E/S consiste en la transferencia de información entre el sistema de memoria y el periférico. La dirección de la comunicación se establece siempre desde el punto de vista del computador. Así, una operación de entrada consiste en la transferencia de información desde el periférico, a través de su interfaz, al sistema de memoria. Una operación de salida consiste en la transferencia de información desde el sistema de memoria a un periférico, a través de su interfaz.

La figura 4.1 ilustra las operaciones de E/S en el computador von Neumann. Dentro de la línea de puntos se encuentran los componentes del computador. Puede observarse cómo algunos periféricos están incluidos en el computador[2].

4.1.1. Ubicación en los espacios de direcciones

Para poder llevar a cabo las operaciones de E/S la CPU necesita acceder a las interfaces para leer o escribir la información. Este acceso se realiza asignando direcciones a los distintos elementos de las interfaces (registros o puertos). Estas direcciones se implementan como posiciones dentro de los espacios de direcciones de la CPU. Ha de recordarse, que el espacio de direcciones es el conjunto de todas las direcciones posibles que puede formar la CPU. La E/S consistirá finalmente en la lectura y escritura en las direcciones en las que se encuentran ubicados los elementos accesibles de las interfaces. El efecto de la lectura o escritura en esas posiciones depende de la interfaz. Por ejemplo, el valor escrito en una cierta posición de la interfaz de

[1] Este es el caso de la interfaz de teclado y ratón PS/2 del PC, la interfaz USB o la interfaz *Firewire*.

[2] Hay periféricos como los discos duros que se emplean para soportar ciertas funciones internas del computador y no para comunicarlo con su entorno. Estos periféricos suelen estar ubicados físicamente dentro del computador.

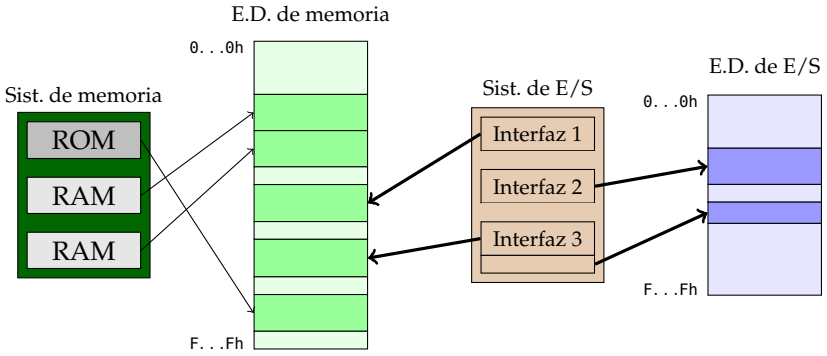


Figura 4.2: Ejemplo de ubicación en los espacios de direcciones

vídeo se convierte en datos que viajan hacia la pantalla, provocando que se visualice una determinada información.

En la práctica, todas las CPU disponen de al menos de un espacio de direcciones, denominado espacio de direcciones de memoria. En este espacio de direcciones se ubica, entre otras cosas, la memoria principal del sistema. Los computadores cuyas CPU disponen de un sólo espacio de direcciones ubican las interfaces de todos los periféricos en el espacio de direcciones de memoria. Está técnica, conocida como E/S mapeada en memoria, consiste en reservar determinadas direcciones para asignarlas a las interfaces de los periféricos. De esta forma, se simplifican los accesos a E/S al convertirlos en simples accesos a memoria que se realizan utilizando las mismas instrucciones, aunque por otro lado, tiene el inconveniente de reducir el número de posiciones disponibles para la memoria principal.

Existen en cambio algunas CPU, como las que implementan la arquitectura x86, que disponen de un espacio de direcciones adicional, denominado espacio de direcciones de E/S. En este último caso, las interfaces pueden ubicarse también en el espacio de direcciones de E/S. Este mecanismo de acceso a E/S recibe el nombre de E/S separada. Tiene la ventaja de que no sacrifica posiciones de memoria en favor de las interfaces, pero como contrapartida, tiene el inconveniente de que necesita instrucciones específicas para acceder al espacio de direcciones de E/S. La figura 4.2 muestra una posible ubicación del sistema de memoria y del sistema de E/S en los espacios de direcciones.

4.1.2. Protección

Desde el punto de vista de la E/S, lo que diferencia una CPU que soporta sistemas operativos multitarea de una que no los soporta es la posibilidad de limitar el acceso a las interfaces de los periféricos. Estas limitaciones impiden a las tareas acceder directamente a las interfaces. Sólo el sistema operativo puede acceder a las interfaces.

La protección de las interfaces ubicadas en el espacio de direcciones de memoria suele llevarse a cabo asignando el nivel de privilegio de supervisor a las páginas de memoria en las que se ubican las interfaces. De esta forma, sólo el sistema operativo puede acceder a dichas interfaces.

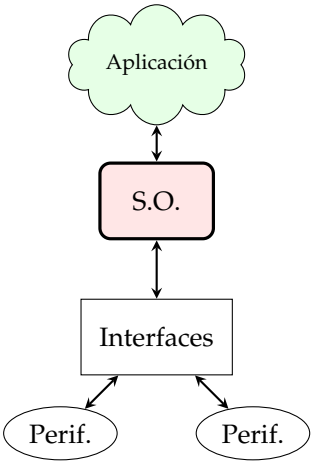


Figura 4.3: Acceso a la E/S en sistemas operativos multitarea

La lectura y escritura del espacio de direcciones de E/S se lleva a cabo empleando instrucciones específicas. Por ejemplo, las instrucciones `in` y `out` en la arquitectura x86. La protección del espacio de direcciones de E/S se consigue permitiendo ejecutar estas instrucciones únicamente en modo supervisor.

La figura 4.3 muestra cómo cualquier aplicación que quiera acceder a las interfaces en un sistema multitarea no podrá hacerlo directamente, sino que debe utilizar los servicios que le proporciona el sistema operativo. Incluso en el caso de sistemas que no tienen soporte multitarea, el acceso a las interfaces se suele realizar empleando los servicios del sistema operativo.

4.1.3. Técnicas de E/S

Tal como se comentó, las operaciones de E/S consisten fundamentalmente en el movimiento de información entre las interfaces de los periféricos y la memoria.

Una cuestión a resolver es cómo se sincroniza la CPU con la operación de E/S. Tanto la CPU como las interfaces de los periféricos pueden iniciar operaciones de E/S. En el primer caso, en un punto de un programa se llega a una instrucción que implica una operación de E/S. Por ejemplo, la llamada a la función de C `fprintf` que escribe en un fichero. Por lo tanto, la CPU es la que inicia la operación de E/S. En el segundo caso, en un momento dado la interfaz de un periférico dispone de datos proporcionados por el periférico, pero la CPU no puede saber a priori cuándo sucede esto. Por ejemplo, la CPU no puede saber a priori cuándo el usuario va a pulsar una tecla o mover el ratón. Se dice entonces que la operación de E/S es iniciada por el periférico o su interfaz.

Una vez iniciada la operación de E/S es necesario mover datos entre la interfaz de un periférico y la memoria. Este movimiento puede ser realizado por la CPU (ejecutando instrucciones de carga y almacenamiento), o pueden utilizarse otros dispositivos para descargar la CPU de este trabajo y que así pueda dedicarse a ejecutar otras tareas.

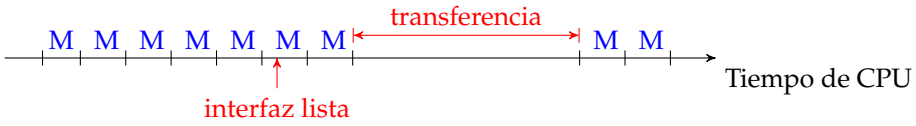


Figura 4.4: Técnica de muestreo continuo

Las técnicas que existen para realizar operaciones de E/S se enumeran a continuación de menos eficiente a más eficiente:

- E/S programada con muestreo.
- E/S con interrupciones.
- Acceso directo a memoria o DMA (*Direct Memory Access*).
- Procesadores de E/S.

E/S programada con muestreo

Es la técnica de entrada/salida más simple, pero también la peor desde el punto de vista del rendimiento del sistema.

Generalmente, las interfaces disponen al menos de un registro de estado y uno de datos. Esta técnica se basa en que la CPU consulta de forma continua el registro de estado de la interfaz, lo que permite conocer cuándo la interfaz está lista, es decir, cuándo tiene datos disponibles en el registro de datos o está preparada para recibir nuevos datos a través de este. Una vez que la interfaz está lista, el movimiento de datos entre la interfaz y la memoria lo lleva a cabo directamente la CPU. Esto se ilustra en la figura 4.4. Como puede observarse, mientras la interfaz no está lista la CPU se encuentra en un bucle en el que lee (muestrea) continuamente el registro de estado. Una vez que la interfaz está lista la CPU sale del bucle.

Esta técnica puede aplicarse simultáneamente a más de una interfaz. Para ello, la CPU debe muestrear de forma secuencial los registros de estado de las diferentes interfaces.

Esta técnica es muy ineficiente, pues mantiene a la CPU ocupada muestreando los registros de estado hasta que haya una interfaz lista. Puede aplicarse a pequeños sistemas monotarea donde el rendimiento no es un factor determinante. No se emplea sin embargo en computadores personales o servidores en los que la multitarea y el rendimiento son factores muy a tener en cuenta. Tampoco se utiliza en dispositivos a batería por el elevado consumo de energía que supone.

E/S con interrupciones

Se trata de una técnica muy empleada con la que se evita el muestreo de las interfaces para conocer cuándo están listas. La idea es muy simple. En lugar de que la CPU muestree el registro de estado de la interfaz, será la propia interfaz quien avise

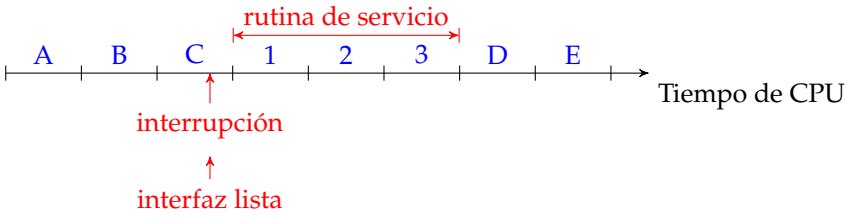


Figura 4.5: Ejecución de un programa con una interrupción

a la CPU cuando esté lista. De esta forma, la CPU no pierde el tiempo en muestreos del registro de estado.

Cuando la interfaz del periférico está lista, por ejemplo cuando dispone de un dato, ésta realiza una petición de interrupción a la CPU. La CPU al final de la ejecución de cada instrucción comprueba si tiene alguna petición de interrupción. Si no hay ninguna petición de interrupción prosigue su ejecución con la siguiente instrucción. En caso contrario, ejecuta un fragmento de programa asociado a la interfaz, en el que típicamente mueve los datos entre la memoria y la interfaz. Una vez finaliza la ejecución de ese fragmento de programa, la CPU prosigue la ejecución del programa justo donde lo había dejado. El fragmento de programa asociado a la interfaz recibe el nombre de rutina de tratamiento de la interrupción o manejador de la interrupción.

La figura 4.5 muestra la ejecución del programa formado por las instrucciones `A`, `B`, `C`, `D` y `E` y de la rutina de tratamiento formada por las instrucciones `1`, `2` y `3` cuando se produce una petición de interrupción durante la ejecución de la instrucción `C`.

El tratamiento de las interrupciones en una CPU segmentada es análogo al de las excepciones, tal como se describió en la sección 2.3.4.

La implementación de las interrupciones en una determinada arquitectura plantea varios problemas a resolver:

- Priorización. Cuando dos o más interfaces solicitan una interrupción durante la ejecución de la misma instrucción debe elegirse cuál se sirve primero. Para ello, deben fijarse prioridades entre las interfaces de los periféricos.
- Inhabilitación de interrupciones. Hay situaciones en las que la CPU no debe ser interrumpida durante la ejecución de una cierta sección de código. Por ejemplo, mientras modifica la tabla de páginas. Si comienza la ejecución de la rutina de interrupción antes de haber modificado completamente la tabla de páginas, puede provocar la caída del sistema. Por supuesto, debe haber también mecanismos para habilitar las interrupciones y así retornar a la situación anterior.
- Identificación de la interfaz. Es muy común que un procesador disponga de un número limitado de líneas de petición de interrupción. Por ejemplo, los procesadores compatibles x86 disponen de una sola línea de petición de interrupción que comparten todas las interfaces. Por lo tanto, una vez recibe una petición de interrupción debe disponer de algún mecanismo para conocer la interfaz (de mayor prioridad) que ha solicitado la interrupción y así saber qué rutina de servicio debe ejecutar.

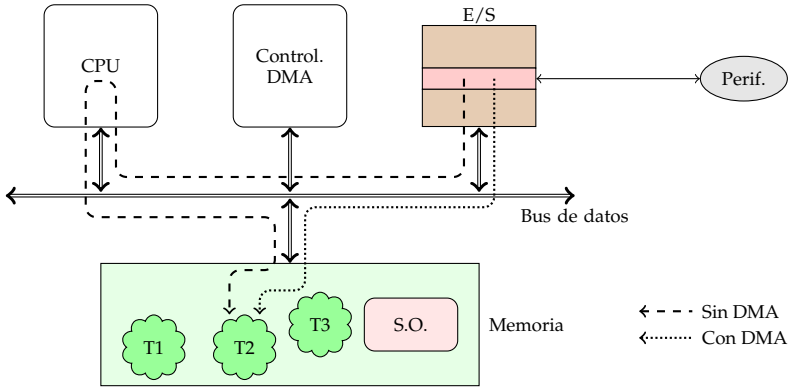


Figura 4.6: Movimiento de datos durante una operación de entrada sin DMA y con DMA en presencia de un sistema operativo multitarea

Acceso directo a memoria o DMA (*Direct Memory Access*)

La E/S con interrupciones tiene un inconveniente, más patente en sistemas operativos multitarea cuando el flujo de datos entre la interfaz y la memoria es elevado. El movimiento de datos entre la memoria y la interfaz lo lleva a cabo la CPU, por lo que ésta no puede ejecutar otros programas mientras se realiza la transferencia de datos entre la interfaz y memoria.

La idea del DMA es liberar a la CPU de este movimiento de datos. Así, éste pasa a ser realizado por un dispositivo denominado controlador de DMA. El controlador de DMA clásico es un elemento que simplemente mueve información entre las interfaces de los periféricos y la memoria.

La figura 4.6 muestra el camino seguido por los datos en una operación de entrada sin DMA y con DMA, en un escenario en el cual se tiene un sistema operativo y 3 tareas de usuario. La tarea `T2` necesita datos de un periférico y se los pide al sistema operativo, quedando suspendida hasta que este se los proporcione.

Empleando la técnica de E/S con interrupciones, el sistema operativo realizaría un cambio de contexto haciendo que la CPU ejecutara otra tarea, por ejemplo la tarea `T1`. Más tarde, una vez que la interfaz está lista generaría una interrupción. La CPU dejaría la tarea `T1` y ejecutaría la rutina de servicio asociada al periférico, dentro de la cual se movería la información de la interfaz a memoria y se despertaría la tarea `T2`, produciéndose un nuevo cambio de contexto. La CPU se encarga de realizar el movimiento de la información en este caso.

Empleando la técnica de DMA también se realiza un cambio de contexto para pasar a ejecutar la tarea `T1`. La diferencia está en que cuando los datos están listos son copiados desde la interfaz a la memoria por el controlador de DMA, mientras la CPU sigue ejecutando la tarea `T1`. Cuando la transferencia de información finaliza se genera una interrupción, pero en este caso el manejador de la misma no tiene que mover los datos, simplemente tiene que despertar la tarea `T2`; se ha mejorado el rendimiento del sistema.

Debe observarse que el DMA aporta ventajas sólo si el sistema operativo empleado es multitarea, o si es monotarea y el controlador de DMA es capaz de mover datos más rápido que la CPU.

Veamos a continuación en detalle cuáles son los pasos habituales de la E/S con DMA.

1. La CPU ejecutando el sistema operativo programa el controlador de DMA indicándole la operación a realizar (lectura o escritura de la interfaz), el número de datos a transferir y las direcciones involucradas en la transferencia.
2. La CPU ejecutando el sistema operativo programa la interfaz del periférico indicándole que debe usar DMA.
3. Cuando la interfaz está lista para que se lleve a cabo la transferencia avisa al controlador de DMA realizando una petición de DMA, tras lo cual este toma el control de los buses y transfiere los datos entre la interfaz y la memoria.
4. Al final de la transferencia, se notifica la conclusión de la operación a la CPU mediante una interrupción.

El controlador de DMA clásico es un dispositivo con las siguientes características:

- Nula inteligencia. Simplemente copia el número de datos especificado de unas direcciones a otras.
- Se trata de un dispositivo compartido. Todas las interfaces con capacidad de DMA comparten el/los controladores de DMA.

Actualmente, el DMA clásico ha evolucionado hacia lo que se conoce como *bus mastering*. Con esta técnica, cada interfaz dispone de un controlador que es capaz de actuar como maestro de los buses. Es decir, es capaz de tomar el control de los buses y emplearlos de igual forma que lo hace la CPU. Este controlador asociado a la interfaz es el encargado de realizar la transferencia de información entre la interfaz y la memoria, pero con mucha más flexibilidad que un controlador de DMA clásico, pues se trata de un dispositivo inteligente. Puede incluso realizar transferencias por iniciativa propia. Al final de la transferencia avisa a la CPU mediante interrupciones.

Procesadores de E/S

Los procesadores de E/S son un paso más en la evolución de la E/S. Se trata de ampliar el controlador de DMA dotándolo de su propia CPU. En este caso la CPU principal solo le indicará las órdenes a realizar, siendo el procesador de E/S el que se encargue de todo el proceso de E/S, lo que habitualmente implicará la ejecución de un programa en el procesador de E/S. Al finalizar la transferencia el procesador de E/S notificará la conclusión de la operación. Se trata de interfaces con capacidad de *bus mastering* que incluyen una o varias CPU con elevada capacidad de procesamiento. En algunos casos, de mayor complejidad que la CPU principal.

Este es el caso de las aceleradoras gráficas actuales. Reciben órdenes de alto nivel por parte de la CPU, como por ejemplo mover la ventana de una aplicación de

una posición de la pantalla a otra. De esta forma, la CPU no tiene que actualizar el estado de los píxeles de las zonas de la pantalla ocupadas inicial y finalmente por la ventana. Este efecto es especialmente significativo en el caso de juegos y programas de animación en tres dimensiones.

Otro ejemplo son las tarjetas de decodificación de imagen y sonido que evitan a la CPU el trabajo de realizar la decodificación, pudiendo dedicarse la CPU a la realización de otras tareas.

La tendencia actual en la E/S es la incorporación de procesadores de E/S, lo que tiende a convertir al computador en un sistema multiprocesador, compuesto de procesadores especializados.

4.2. Sistema de interconexión

En la máquina von Neumann los distintos componentes intercambian información mediante unos canales de comunicación denominados buses de datos, direcciones y control. En general, la realización práctica de los canales de comunicación dentro del computador es lo que se conoce como sistema de interconexión del computador. En la actualidad, el sistema de interconexión es heterogéneo abarcando diferentes tecnologías, en muchos casos motivada por la diversidad de las interfaces y sus características de comunicación.

4.2.1. Topologías

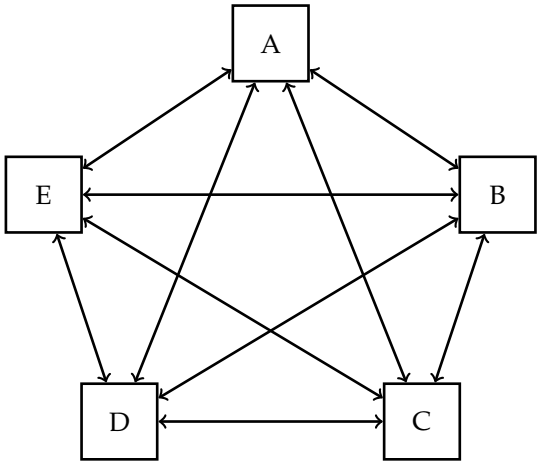
El computador está formado por dispositivos que necesitan comunicarse. Este es el caso de la CPU, el sistema de memoria y las interfaces de los periféricos. El computador incluye además dispositivos de control que ocupan posiciones dentro de los espacios de direcciones, como los temporizadores, controladores de interrupciones, etc. Finalmente, las interfaces de los periféricos necesitan comunicarse con los periféricos.

La forma más simple de conectar dos dispositivos es mediante un canal punto a punto. Con este tipo de conexión, si dos dispositivos `A` y `B` necesitan comunicarse, existe un canal dedicado que conecta única y exclusivamente `A` y `B`. La figura 4.7 muestra los canales punto a punto necesarios para comunicar entre sí los dispositivos `A`, `B`, `C`, `D` y `E`.

Esta técnica de interconexión tiene varias ventajas:

- Facilidad de implementación.
- Elevada velocidad de transferencia.
- Es posible el paralelismo en las transferencias.

Sin embargo, también tiene importantes inconvenientes:



$$\text{Nº de canales} = \frac{\text{Nº de dispos.} \times (\text{Nº de dispos.} - 1)}{2}$$

Figura 4.7: Conexión punto a punto entre los dispositivos del computador

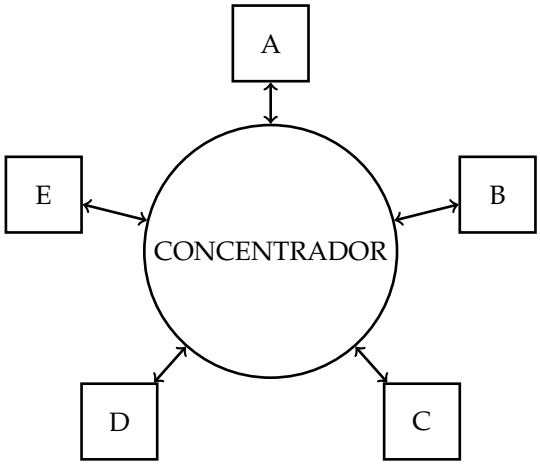
- Como se puede observar en la figura 4.7, son necesarios 10 canales para comunicar los dispositivos entre sí, todos con todos. Si en lugar de cinco dispositivos hubiese por ejemplo veinte, harían falta 190 canales. Cuando es necesario comunicar entre sí muchos dispositivos, el número de canales necesarios crece rápidamente, con el coste económico que esto supone.
- Cuando un dispositivo desea enviar la misma información a más de un dispositivo destino es necesario llevar a cabo tantas comunicaciones como dispositivos destino.

Está claro que el mayor defecto de los canales punto a punto es el elevado número de canales necesarios. Una forma de solucionar este problema consiste en emplear un concentrador (*hub*) y desplegar una topología en estrella. La figura 4.8 muestra la conexión de los dispositivos `A`, `B`, `C`, `D` y `E` empleando un concentrador.

Esta topología requiere un solo canal por dispositivo. Cada vez que un dispositivo desea enviar información a otro envía al concentrador no sólo esta información, sino también información que identifica el dispositivo destino. El concentrador utiliza dicho identificador para retransmitir la información al dispositivo destino.

La topología en estrella tiene las siguientes ventajas con respecto al empleo de canales punto a punto:

- Requiere un menor número de canales. Entre el concentrador y los dispositivos hay tantos canales punto a punto como dispositivos.
- Si el concentrador lo permite, es posible retransmitir a varios dispositivos la información procedente de un dispositivo.



Nº de canales = Nº de dispositivos

Figura 4.8: Topología en estrella de los dispositivos del computador empleando un concentrador

La principal desventaja de esta topología es la necesidad del concentrador. Este concentrador debe ser muy rápido y permitir transferencias simultáneas entre varios dispositivos para evitar ser un cuello de botella. Por ejemplo, sería recomendable que el concentrador permitiese simultáneamente la comunicación entre los dispositivos `A` y `B` y entre los dispositivos `E` y `C` de la figura 4.8.

Otra de las topologías de conexión existentes es la topología de bus. Un bus no es más que un canal de comunicación compartido por varios dispositivos. La figura 4.9 muestra la conexión de los dispositivos `A`, `B`, `C`, `D` y `E` mediante un bus.

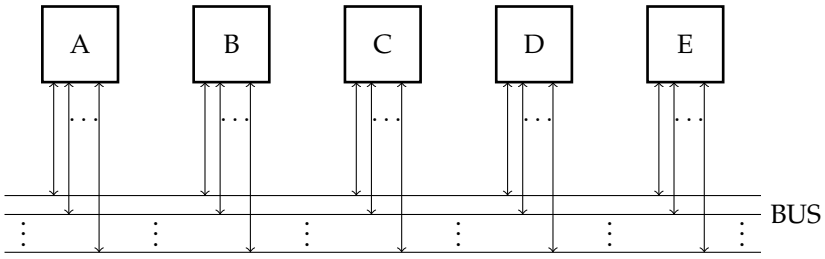
Como puede observarse, el número de conexiones al bus coincide con el número de dispositivos, lo que hace del empleo de buses una técnica de interconexión barata. Dos dispositivos cualesquiera conectados al bus se comunican a través de éste. Cuando un dispositivo desea enviar la misma información a más de un dispositivo destino, puede enviarla sólo una vez en modo *broadcast*.

Sin embargo, no todo son ventajas empleando el bus como técnica de interconexión. La compartición del medio físico de conexión impide transferencias de información simultáneas y puede convertir el bus en un cuello de botella. Por otro lado, la constitución del bus mediante líneas paralelas limita su frecuencia de trabajo debido a problemas de interferencias entre las líneas, y por tanto su velocidad de transmisión.

La conexión de todos los dispositivos de un computador a través de un único mecanismo de interconexión plantea numerosos problemas.

- En el computador hay dispositivos de capacidades y características muy dispares. Por ejemplo, un mecanismo de interconexión apropiado para recibir datos

Nº de conexiones = Nº de dispositivos



Representación simplificada

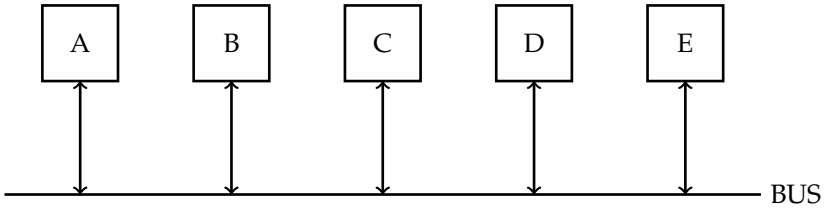


Figura 4.9: Conexión de los dispositivos mediante un bus

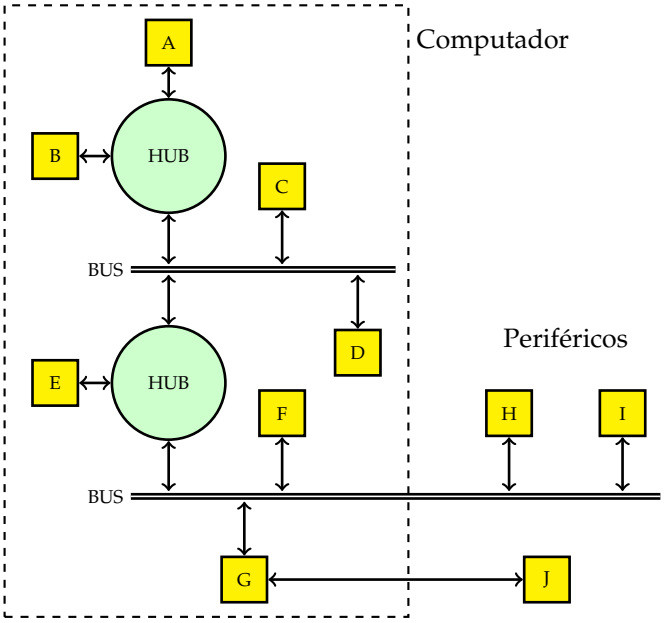


Figura 4.10: Combinación de técnicas de interconexión

de una interfaz de red a velocidades de 1 Gbps no es apropiado para la interfaz de un teclado, con velocidades de varios bytes por segundo.

- Con un elevado número de dispositivos conectados a un bus, la capacidad de transmisión efectiva del mismo se reduce por la sobrecarga de los protocolos de comunicación. Este problema también aparece, aunque en menor medida, con las conexiones a través de concentradores.

Para solventar dichos problemas, en la práctica se combinan las tres topologías. La figura 4.10 muestra un ejemplo. En la misma pueden observarse concentradores capaces de gestionar no sólo conexiones punto a punto, sino también buses.

4.2.2. Características

A continuación se indican una serie de características cualitativas que diferencian unas tecnologías de interconexión de otras.

- Sincronización. Las tecnologías se pueden dividir en síncronas y asíncronas. Las síncronas se caracterizan por disponer de una señal de reloj dentro de sus líneas de control. Esta señal de reloj se utiliza para secuenciar todas las fases dentro de la comunicación. Por ejemplo, en un canal punto a punto el emisor deposita la información sobre unas líneas de datos y genera una señal de reloj para que el receptor conozca dónde empieza y dónde termina cada unidad de datos que transmite.

En las tecnologías asíncronas hay líneas adicionales de control, por ejemplo, una línea que se activa cuando hay datos disponibles para ser leídos. En general, estas líneas de control se van activando por el emisor y el receptor siguiendo un protocolo que permite la realización de la comunicación entre los dispositivos.

En general las conexiones síncronas son más simples, lo que facilita su implementación y proporciona una mayor velocidad de transferencia. Sin embargo, la longitud del canal de comunicación debe ser corta o la frecuencia de reloj baja, ya que la señal de reloj se deforma al viajar por el canal, llegando un momento en que no sería reconocible. Esta degradación se ve aumentada con la distancia y la frecuencia en un efecto que se conoce como *clock skew*.

- Tamaño de los datos. La función primordial es transferir datos. Cada operación de lectura o escritura supone la transferencia de un dato que tiene un cierto tamaño en bits.

- Serie o paralelo. Una conexión paralela dispone de tantas líneas de datos como bits hay en un dato transferido. Por el contrario, una serie dispone de menos líneas, normalmente una, a través de la que se transmiten en secuencia todos los bits que componen un dato. Por ejemplo, si el tamaño de los datos es 32 bits y se dispone de una única línea de datos, es necesario llevar en secuencia (en serie) los 32 bits sobre la misma línea de datos.

- Multiplexado o no multiplexado. Con objeto de reducir el número de líneas suelen emplearse determinadas líneas para llevar a cabo varias funciones en instantes diferentes. Por ejemplo, es habitual multiplexar las líneas de datos y direcciones. De esta forma, durante una escritura se pone una dirección sobre las líneas multiplexadas y a continuación se pone el dato a escribir sobre las mismas líneas. La multiplexación reduce el número de líneas, lo que mejora el coste y la fiabilidad. No obstante, suele implicar una reducción en la velocidad de transferencia respecto a una versión no multiplexada.

- Ancho de banda teórico. Representa la velocidad de transferencia máxima en condiciones ideales. Suele expresarse en Megabits por segundo (Mbps) o Gigabits por segundo (Gbps).

- Longitud máxima. La longitud del medio físico de conexión no puede ser arbitraria. Las señales eléctricas que viajan por los cables no se propagan instantáneamente, sino a una velocidad ligeramente inferior a la de la luz[3]. Este efecto es especialmente crítico en las comunicaciones síncronas por el efecto *clock skew*. Además, debe tenerse en cuenta que los cables o las pistas de circuito impreso funcionan como antenas que emiten y reciben ondas electromagnéticas, lo que puede provocar señales incorrectas en los cables. Los efectos mencionados son tanto mayores cuanto más largo es el cable.

- Plug and Play (PnP). Las tecnologías PnP definen mecanismos para conocer la identidad de los dispositivos conectados, así como los recursos hardware que necesitan (posiciones dentro de los espacios de direcciones, líneas de petición

[3] Requieren unos 3 ns en recorrer un metro.

de interrupción, etc.). La idea es que estos dispositivos se conectan y el sistema automáticamente los configura de una forma totalmente transparente al usuario para evitar conflictos.

- Conexión en caliente. Hay tecnologías de interconexión que permiten la conexión en caliente de dispositivos, es decir, mientras el sistema está funcionando.

4.2.3. PCI Express (PCIe)

Todas las tecnologías empleadas en el computador, incluidas las más exitosas, deben ser sustituidas por otras nuevas en algún momento. La causa fundamental reside en la continua mejora de velocidad a la que se ven sometidos los diferentes elementos del computador.

Este fue el caso de la especificación PCI, precursora de PCI Express. La especificación inicial evolucionó durante muchos años ampliando sus capacidades. Sin embargo, sus limitaciones han hecho que haya prácticamente desaparecido del mercado. El hueco dejado por PCI fue cubierto por otra tecnología de conexión, nacida en el año 2002, denominada PCI Express.

Muchas son las diferencias a nivel físico entre PCI y PCI Express. Sin embargo, inicialmente fueron compatibles a nivel software. La idea fundamental de esta compatibilidad es que todo el software realizado previamente para dispositivos PCI fuese perfectamente válido para dispositivos PCI Express. Esta idea general de compatibilidad hacia atrás es muy importante en el mundo de los computadores cada vez que se desea introducir una nueva tecnología, o mejorar una tecnología existente.

A continuación, como ejemplo de sistema de interconexión, se enumerarán algunas de las características de PCI Express versión 1.0 a nivel físico:

- PCI Express es una conexión punto a punto. Cada conexión PCI Express permite conectar única y exclusivamente dos dispositivos de forma bidireccional.
- PCI Express es una conexión serie. Los bits en serie de cada byte se envían no sobre un hilo, sino sobre dos hilos de forma diferencial[4]. No obstante, cada conexión PCI Express puede tener varios carriles (*lanes*) serie trabajando en paralelo.
- Cada carril PCI Express es capaz de transmitir 2 Gbps en un sentido y otros 2 Gbps en el otro sentido al mismo tiempo, lo que supone una tasa de transferencia teórica de 4 Gbps.
- Cada conexión PCI Express puede tener 1, 2, 4, 8, 12, 16 o 32 carriles, en cuyo caso recibe el sufijo 1x, 2x, 4x, 8x, 12x, 16x o 32x. Por lo tanto, con la especificación PCI Express se puede alcanzar una tasa de transferencia teórica de hasta 128 Gbps. Además, se pueden conectar dispositivos PCI Express con un número de carriles diferente, pues antes de iniciar la comunicación se produce

[4]El bit se transmite como diferencia de tensiones entre los dos hilos. De esta forma la transmisión es inmune a las interferencias, puesto que aunque afecten a los dos hilos, lo hará de forma similar en ambos, manteniéndose la diferencia entre señales y por tanto la información.

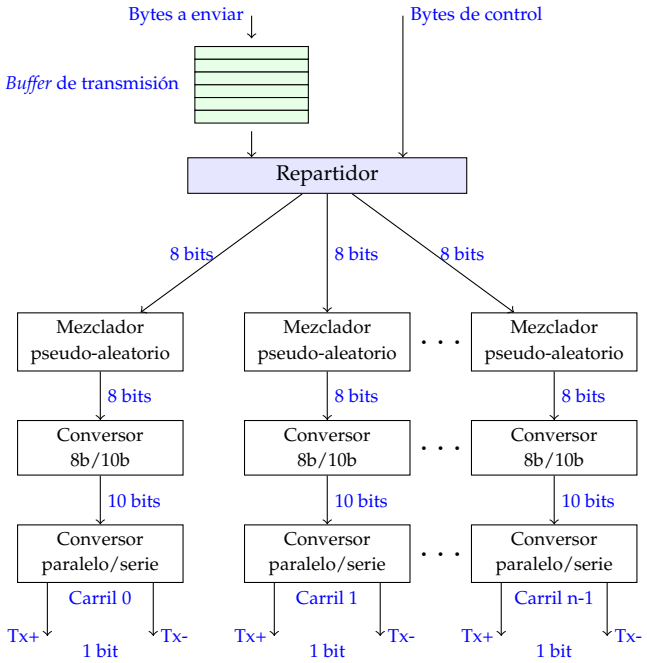


Figura 4.11: Procesamiento del paquete a enviar por un dispositivo PCI Express

una negociación para decidir el número de carriles a utilizar. Por ejemplo, si se conecta un dispositivo con 4 carriles a otro con 12 carriles, la comunicación se llevará a cabo con 4 carriles.

- El número de pines empleado por PCI Express es muy bajo. PCI Express emplea tan sólo 2 líneas por cada carril. Esto supone una gran reducción de costes, así como una mejora en la fiabilidad.

La figura 4.11 muestra cómo el hardware emisor de uno de los dispositivos PCI Express procesa un paquete de datos que se envía a otro dispositivo PCI Express. El procesamiento llevado a cabo por el receptor sería justo el inverso, de ahí que no se muestre.

El paquete a enviar es un conjunto de bytes que se almacenan en el *buffer* de transmisión del emisor. Debe tenerse en cuenta que junto con los datos se envía información de control. Hay un repartidor que se encarga de seleccionar un byte de datos o un byte de control, vaciando en el primer caso de forma progresiva el *buffer* de transmisión.

La secuencia de bytes generada por el repartidor (bytes de datos y control) se reparte byte a byte entre los carriles de comunicación.

Cada carril tiene asociado un mezclador pseudo-aleatorio, un conversor 8b/10b y un conversor paralelo a serie.

Versión	Año	Velocidad teórica	Velocidad efectiva	Codificación
PCIe 1.0	2003	2.5 Gbps	2 Gbps	8b/10b
PCIe 1.1	2005	2.5 Gbps	2 Gbps	8b/10b
PCIe 2.0	2007	5 Gbps	4 Gbps	8b/10b
PCIe 2.1	2009	5 Gbps	4 Gbps	8b/10b
PCIe 3.0	2010	8 Gbps	7.88 Gbps	128b/130b
PCIe 3.1	2014	8 Gbps	7.88 Gbps	128b/130b
PCIe 4.0	2017	16 Gbps	15.75 Gbps	128b/130b
PCIe 5.0	2019	32 Gbps	31.5 Gbps	128b/130b

Tabla 4.1: Versiones y velocidades de PCIe

El mezclador pseudo-aleatorio cambia el orden de los bytes que recibe para conseguir que no haya patrones de bytes repetidos, distribuyendo la radiación electromagnética de las líneas `Tx+` y `Tx-` del carril en un amplio rango de frecuencias[5].

El conversor 8b/10b transforma cada byte en una secuencia de 10 bits. El objetivo de esta transformación es conseguir que la distancia entre las transiciones `0` a `1` y `1` a `0` no sea mayor que un número de bits determinado. La razón es que estas transiciones sincronizan el reloj del receptor con el reloj de 2.5 GHz del emisor. Sin esta codificación, la separación entre dos transiciones consecutivas puede ser arbitrariamente grande, por ejemplo, cuando tenemos muchos bytes seguidos iguales a 00h o muchos bytes seguidos iguales a FFh. Teniendo en cuenta la frecuencia de reloj del emisor y que en cada ciclo de reloj se transfiere un bit, resulta que la velocidad de transferencia teórica (en un sentido) es de 2.5 Gbps. Sin embargo, la velocidad efectiva es de 2 Gbps, debido a los dos bits añadidos en la conversión 8b/10b.

El conversor paralelo a serie recibe palabras de 10 bits y transmite sus bits en serie sobre las líneas `Tx+` y `Tx-`. Cada bit se transmite de forma diferencial. Un uno lógico es una diferencia de tensiones positiva entre `Tx+` y `Tx-`, mientras que un cero lógico es una diferencia de tensiones negativa.

A lo largo de los años la tecnología de PCIe ha ido mejorando en sucesivas versiones hasta llegar a las versiones actuales, que proporcionan mejoras sustanciales. En la tabla 4.1 se resumen las principales versiones de PCIe lanzadas hasta el momento de redacción de este texto junto con sus anchos de banda para un carril en un solo sentido.

4.3. Periféricos

Este apartado está dedicado a los dispositivos que permiten al computador relacionarse con el mundo exterior. Estos dispositivos reciben el nombre de periféricos y se comunican con el computador a través de interfaces cuya misión es adecuar las características de funcionamiento de cada periférico al funcionamiento digital del computador.

[5]De esta forma, la radiación electromagnética que reciben los conductores cercanos se asemeja a un ruido aleatorio (ruido blanco), menos pernicioso que un ruido concentrado en unas pocas frecuencias.

4.3.1. Introducción

Si bien es complicado establecer una clasificación de periféricos debido a la cantidad de criterios que se podrían seguir, la clasificación más general que se puede realizar es atendiendo al sentido en el que fluye la información. La clasificación de los periféricos podría ser la siguiente:

- **Periféricos de entrada**. Su misión principal es introducir información al computador. Los más utilizados son el teclado y el ratón. No obstante, también pertenecen a este grupo: lectores de banda magnética, detectores ópticos (marcas/barras/puntos/caracteres), reconocimiento de voz, lápices ópticos, *joysticks*, escáneres, cámaras de fotos y vídeo digital, etc.
- **Periféricos de salida**. Su misión es presentar la información al usuario. El periférico más común de este tipo es la pantalla, aunque también pertenecen a este grupo: la impresora, el registrador gráfico o *plotter*, diversos tipos de *displays* específicos, dispositivos para realidad aumentada, etc.
- **Periféricos de entrada/salida**. Estos periféricos sirven tanto para introducir información al computador como para extraerla. En este grupo los elementos más representativos son los periféricos de interconexión como las tarjetas de red. En este grupo podrían englobarse también las tarjetas de sonido, las pantallas táctiles, etc.
- **Periféricos de memoria auxiliar o almacenamiento**. Este grupo de periféricos es un caso particular de los periféricos de entrada/salida, pues sirven tanto para introducir como extraer información del computador. Sin embargo, el objetivo del trasiego de información tiene como destino fundamental el almacenamiento, más que la presentación. Pertenecen a este grupo los discos magnéticos, los discos ópticos (CD, DVD y Blu-ray), las cintas magnéticas, los discos magneto ópticos y la tecnología de almacenamiento basada en semiconductores por ejemplo los lápices USB y unidades SSD.

Debido a su importancia, se tratarán en mayor profundidad los periféricos de almacenamiento, representados por el disco duro y la unidad de estado sólido.

4.3.2. Discos duros

Un disco duro, o HDD (*Hard Disk Drive*), es un dispositivo que permite almacenar información de forma no volátil, es decir, la información se mantiene en ausencia de alimentación. Está formado por varios platos unidos por un eje que giran solidarios a velocidades que pueden alcanzar las 15 000 rpm. Las superficies de estos platos están recubiertas de un material ferromagnético que permite almacenar la información. En ambos lados de cada plato hay una cabeza que permite llevar a cabo la lectura y escritura de la información. Las cabezas están unidas mediante un brazo a una estructura mecánica que les permite moverse radialmente sobre el plato (hacia adentro y hacia afuera). Este movimiento radial, junto con el movimiento de giro del disco, permite a las cabezas alcanzar cualquier punto del mismo. En la figura 4.12 se muestra un esquema de los componentes de una unidad de disco.

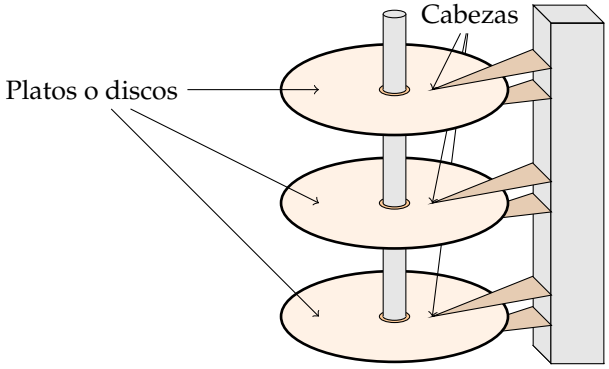
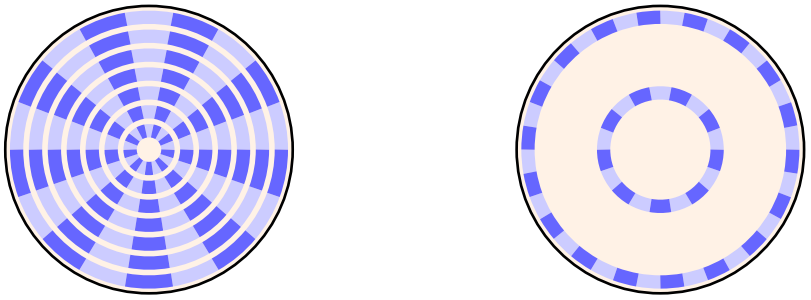


Figura 4.12: Partes de una unidad de disco



(a) Número de sectores constante

(b) Densidad de información constante

Figura 4.13: Distribución de la información sobre un plato

Organización de la información

La organización de la información determina cómo debe ubicarse la información para posteriormente recuperarla. En la figura 4.13 se muestra esta organización. En esta figura se puede indentificar:

- Pistas o *tracks*. Son círculos concéntricos sobre la superficie del plato. Se numeran desde el 0 al (Nº total pistas $-$ 1). Se le asigna el número 0 a la pista situada más externamente.
- Cada una de las pistas se divide en pequeños segmentos denominados sectores. Cada sector contiene un determinado número de bytes, generalmente 512 o 1024 bytes.

  El sector es la unidad mínima que se puede leer o escribir en un disco duro (aunque sólo sea necesario un byte, se lee o escribe un sector completo).
- En un plato se puede almacenar información en una cara o en ambas. Técnicamente se denominan superficies.

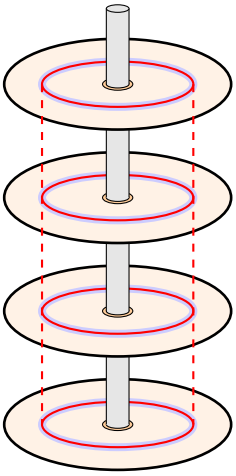


Figura 4.14: Definición del cilindro en un disco duro

- Por último, se define con el término de cilindro al conjunto de pistas con igual número en todas las superficies. Es decir, a todas las pistas que estarían sobre la misma vertical. En la figura 4.14 se muestra la definición gráfica de cilindro.

Dependiendo del sistema operativo se define la unidad conocida como *cluster*. Un *cluster* agrupa dos o más sectores y representa el tamaño mínimo de disco que se puede utilizar para realizar una operación de lectura o escritura en un determinado sistema de ficheros.

Durante el proceso de formateado a bajo nivel, las cabezas escriben la información necesaria para identificar las pistas y sectores sobre cada plato. Esta información servirá para ubicar la información en el disco a la hora de leer o escribir.

Después del proceso de formateado a bajo nivel, le sigue una operación de formateado a alto nivel. Esta operación consiste básicamente en copiar en una zona definida del disco duro información que permita iniciar el computador (*Master Boot Record* y la tabla de particiones), así como una estructura de datos (sistema de ficheros) que permita servir como índice para encontrar los archivos sobre el disco. En esta tabla, para cada nombre de fichero se guarda el número de *cluster* en el que comienza[6]. Los *clusters* están numerados de forma consecutiva y a cada uno de ellos le corresponden unas determinadas coordenadas de superficie, pista y sectores. Estos valores son los que se envían a la controladora de disco para acceder al contenido del *cluster*.

Finalmente, se debe realizar una consideración relativa al número de sectores que existen en un plato. En la figura 4.13a puede verse cómo con la definición realizada, en cada pista existe el mismo número de sectores. Ahora bien, como en cada sector se almacena la misma cantidad de información (512 bytes), en los sectores más externos, que son más grandes, la densidad de información es menor, mientras la

[6]Si el fichero ocupa más de un *cluster*, al final del *cluster* en uso se almacena el número de *cluster* en el que continúa el fichero. El último *cluster* finaliza con una secuencia especial de fin de fichero.

densidad de información aumenta en los sectores próximos al centro. Este hecho supone un desperdicio de material magnético. Otra alternativa consiste en mantener la densidad de información constante. De esta forma en las pistas más externas habrá más sectores que en las pistas más internas, tal como se puede ver en la figura 4.13b. Esta forma de almacenar la información aprovecha al máximo la superficie magnética.

Parámetros básicos

Los parámetros más importantes a considerar en un disco duro son los siguientes:

1. **Capacidad**. Es el número de bytes que puede almacenar la unidad. Se calcula:

$$\text{Capacidad Total} = \frac{\text{Bytes}}{\text{Sector}} \times \frac{\text{Sectores}}{\text{Pista}} \times \frac{\text{Pistas}}{\text{Superficie}} \times \frac{\text{Superficies}}{\text{Unidad}}$$

   La capacidad real es ligeramente menor, pues parte de la capacidad total se dedica a contener información que permite localizar los datos almacenados[7].

2. **Tiempo de acceso**. Es el tiempo necesario para acceder a una información ya sea para lectura o escritura. Esto requiere una serie de acciones con su correspondiente consumo de tiempo:

   - Situar de forma estable, sin oscilaciones, la cabeza de lectura/escritura sobre la pista adecuada. Al tiempo necesario se le denomina tiempo de búsqueda (*track seek time*).
   - Esperar a que el principio del sector buscado pase bajo la cabeza de lectura/escritura. A este tiempo se le denomina latencia rotacional (*rotational delay*) o simplemente latencia. El tiempo de latencia depende de la velocidad de giro de la unidad; estadísticamente se estima como la mitad del tiempo que invierte la unidad en dar una vuelta completa.
   - Transferir (leer o escribir) los bytes según van pasando bajo la cabeza. Al tiempo necesario se le denomina tiempo de transferencia (*transfer time*).

   Considerando las tres etapas necesarias, el tiempo de acceso puede cuantificarse mediante la expresión:

$$\text{Tpo. acceso} = \text{Tpo. búsqueda} + \text{Tpo. latencia} + \text{Tpo. transferencia}$$

3. **Densidad de almacenamiento**. Una vez conocidas las dimensiones físicas del disco duro y la capacidad de almacenamiento, se pueden obtener medidas que representan la cantidad de información almacenada en relación con el espacio físico usado. Estas medidas se pueden ver en la figura 4.15 y se definen como:

   TPI: Pistas por pulgada (*Tracks per inch*). Indica el número de pistas almacenadas por unidad de longitud radial del plato.

   BPI: Bits por pulgada (*Bits per inch*). Indica el número de bits almacenados por unidad de longitud de pista. Si el número de sectores por pista es constante, esta densidad aumenta según las pistas se van aproximando al centro del disco.

[7] Las estructuras de datos del sistema de ficheros.

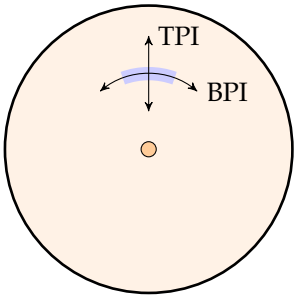


Figura 4.15: Definición de las unidades de densidad de almacenamiento

4. ***Buffer***. Los discos duros disponen de lo que se denomina un *buffer* de disco. Cuando el disco recibe una petición de lectura, el disco accede a la pista donde está almacenada la información y en lugar de leer sólo los sectores solicitados, copia toda la información de la pista al *buffer* de disco y la petición se sirve desde allí. La misión del *buffer* de disco es actuar como una memoria caché del disco. Entra en juego de nuevo el principio de localidad de referencias.

4.3.3. Unidades de estado sólido

Una unidad de estado sólido, o SSD (*Solid State Drive*), es un dispositivo que, al igual que el disco duro, permite almacenar información de forma persistente. A diferencia de este último, el almacenamiento en una unidad SSD está basado en semiconductores, esto es, el mismo principio que la memoria principal del computador: disponen de celdas de memoria basadas en transistores cada una de las cuales puede almacenar un bit.

Las unidades SSD actuales utilizan memoria *flash*, que tiene como celda base puertas NAND. Las celdas NAND flash son más lentas que las celdas de memoria DRAM, utilizadas para la memoria principal del computador, pero al no tener partes mecánicas móviles son aún mucho más rápidas que los discos duros. Dependiendo de la tecnología empleada se puede conseguir que una única celda pueda almacenar, uno, dos o tres bits (el rango de tensión almacenado se dividiría en 2, 4 u 8 niveles de tensión). El incremento de capacidad conseguido con cada tecnología lleva aparejado un coste en velocidad: a mayor número de bits por celda, mucho más lento es el acceso a la información debido a la necesidad de distinguir entre los diferentes niveles posibles.

La tecnología usada para el almacenamiento en las unidades SSD es la misma que la usada en los lápices USB. La diferencia está en su capacidad y la interfaz que permite conectarlos al computador.

Organización de la información

En una unidad SSD las celdas NAND se organizan de forma matricial. Cada fila de la matriz se conoce como página y la matriz completa recibe el nombre de bloque.

Los tamaños de página habituales pueden ser: 2 KiB, 4 KiB, 8 KiB o 16 KiB, y dentro de cada bloque puede haber 128 o 256 páginas, lo que da un tamaño de bloque que oscila entre 256 KiB y 4 MiB.

El funcionamiento de la unidad SSD está relacionado con la organización de la información. Debe distinguirse también entre el proceso de lectura/escritura/sobreescritura.

- La cantidad más pequeña de información que se puede leer en un SSD es una página. La lectura es un proceso muy rápido.
- En el caso de la escritura, también se realiza a nivel de página y siempre que la página esté vacía. Aunque más lento que la lectura, la escritura también es un proceso rápido.
- Un caso diferente es cuando se lleva a cabo una escritura sobre una página que no está vacía (que fue escrita anteriormente), por ejemplo para actualizar el contenido de un archivo. En este caso el proceso es diferente, pues implica un borrado de información previo y el borrado se realiza a nivel de bloque. Una opción es copiar el contenido del bloque a memoria principal, borrar el bloque en el SSD, realizar la modificación y volver a escribir el bloque al SSD. Otra opción sería transferir el contenido del bloque modificado a un bloque vacío y marcar el bloque antiguo como disponible.

Las sobreescrituras y la falta de espacio a medida que se llena la unidad son las responsables de que estos dispositivos se vayan ralentizando con el tiempo. Es labor del controlador de la unidad SSD balancear las escrituras para, en la medida de lo posible, evitar esta merma de rendimiento.

4.3.4. Comparativa entre discos duros y unidades de estado sólido

La principal diferencia entre ambos tipos de dispositivos es que los HDD basan su funcionamiento en el movimiento giratorio del disco y de traslación de las cabezas lectoras. En cambio, en los SSD no hay partes móviles. Este aspecto influye de la siguiente forma:

- Los SSD son más rápidos que los HDD al no tener partes móviles, especialmente para lectura. Este es el motivo por lo que se suelen utilizar SSDs como unidad de soporte al sistema operativo, ya que el arranque es un proceso mayoritariamente de lectura.
- Al no tener partes móviles, los SSD son mucho más silenciosos que los HDD.
- Los SSD también son más fiables ante entornos ambientales hostiles. Al no tener partes móviles no se ven afectados por vibraciones, humedades, etc. De hecho, el primer uso de los SSD fue en entornos industriales donde los HDD no resultaban fiables.
- Los SSD tienen un tamaño mucho menor que un HDD a igualdad de capacidad, al no requerir toda la parte mecánica.

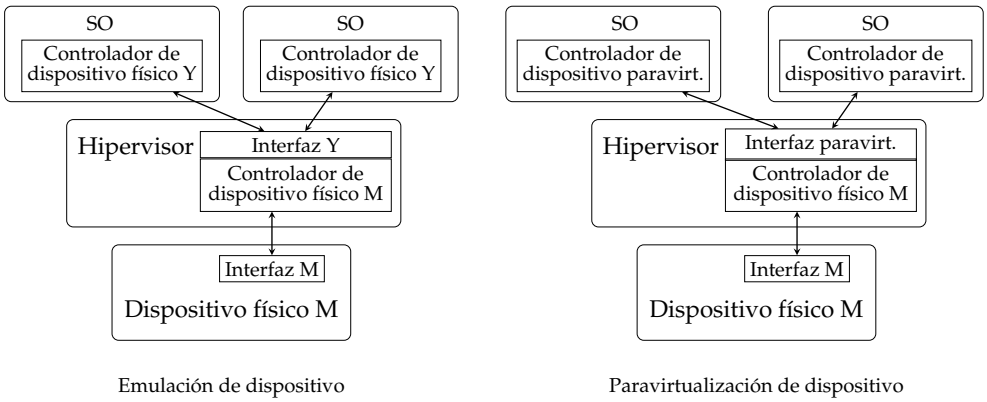


Figura 4.16: Virtualización de dispositivos por software

Los HDD también tienen algunas ventajas respecto a los SSD:

- Los HDD tienen una mayor densidad de almacenamiento, lo que les permite una mayor capacidad que los SSD (a igual coste). En los SSD al reducir el tamaño de las puertas NAND se ve afectada la persistencia de la información.
- El coste por GiB en los HDD es mucho menor que en los SSD, por lo que resulta más barato el almacenamiento en HDD.
- Por último, comentar que en los SSD los procesos de escritura y sobreescritura afectan a la vida útil de los bloques, si bien el número de operaciones a realizar para producir el fallo es cada vez más elevado debido a las mejoras en los controladores de los SSD.

4.4. Virtualización de la E/S

En un sistema virtualizado las máquinas virtuales tienen que compartir el acceso a los dispositivos de E/S físicos, idealmente sin causar problemas de seguridad o rendimiento. El sistema operativo que se ejecuta en cada máquina virtual debe percibir la «ilusion» de dispositivos sobre los que tiene un control absoluto a través de sus interfaces, tal como ocurre en los entornos no virtualizados.

Por ejemplo, en un servidor con una tarjeta de red física y varias máquinas virtuales ejecutándose, se desea que cada una de las máquinas disponga de una tarjeta de red virtual por la que enviar sus mensajes sin interferir con las tarjetas de red virtuales de otras máquinas. Las operaciones sobre las tarjetas de red virtuales tendrán que ser servidas por la tarjeta de red física, con lo que se está virtualizando esta última.

La virtualización de la E/S puede conseguirse por software, de forma análoga a la virtualización de la CPU, empleando las técnicas que se enumeran a continuación y se ilustran en la figura 4.16.

- Emulación de dispositivos. En esta alternativa, los dispositivos de E/S virtuales se emulan por software y, además, el controlador de dispositivo (*driver*) utilizado por el sistema operativo es un controlador asociado a una interfaz física. Por ejemplo, si se virtualiza la interfaz de red para que la interfaz virtual sea compatible con la interfaz real 82540EM de Intel, el sistema operativo instalado en la máquina virtual empleará el mismo controlador que emplearía sobre una máquina real que incorporase la interfaz real 82540EM.

  Para conseguir la emulación, el hipervisor utiliza la técnica *trap-and-emulate*. Cualquier operación de lectura o escritura sobre la interfaz `Y` llevada a cabo por el controlador `Y` de la figura (ejecutado por el sistema operativo) es capturada por el hipervisor y traducida en operaciones sobre la interfaz física `X`. Asimismo, cualquier operación que la interfaz física `X` deba notificar, debe ser trasladada al controlador `Y` pasando por el hipervisor. Aparte de la sobrecarga de la traducción entre dos interfaces diferentes `X` e `Y`, estas operaciones suponen varios cambios de contexto entre el sistema operativo y el hipervisor, haciendo que esta técnica de virtualización tenga un rendimiento muy bajo.

- Paravirtualización de dispositivos. Es una variante de la emulación de dispositivos, pero en este caso la máquina virtual presenta al sistema operativo una interfaz paravirtualizada. Se trata de una interfaz que no coincide con ninguna interfaz física y está adaptada para coordinarse de manera más eficiente con el hipervisor. Esta alternativa mejora el rendimiento respecto a la opción anterior y al contrario de lo que ocurría con la paravirtualización explicada en el tema de CPU, no requiere modificar el sistema operativo. No obstante, tiene la desventaja de requerir un controlador de dispositivo específico que debe proporcionarse al sistema operativo. Además, el rendimiento, aunque mejor que el de la opción anterior, sigue siendo pobre.

Para reducir los problemas de rendimiento anteriores se han ideado técnicas de virtualización de E/S asistida por hardware, cuyo objetivo fundamental es comunicar las máquinas virtuales y los dispositivos de E/S con la mínima intervención del hipervisor. Estas técnicas se describen a continuación y se ilustran en la figura 4.17.

- Asignación de dispositivos o *passthrough*. El hipervisor asigna el dispositivo físico a la máquina virtual, por lo que la máquina virtual accede directamente al dispositivo físico sin intervención del hipervisor, lo que supone una gran mejora de rendimiento. Un hipervisor de tipo 1 empleando esta técnica no requiere disponer del controlador de dispositivo, lo que amplia el rango de hardware compatible. De forma análoga, el sistema operativo anfitrión con virtualización de tipo 2 tampoco requiere disponer del controlador de dispositivo, solo lo requiere el sistema operativo invitado. El único inconveniente de esta técnica es que un dispositivo físico sólo se puede asignar a una máquina virtual. Por ejemplo, no puede aplicarse si se dispone de una única interfaz de red que deben compartir dos máquinas virtuales.

- Compartición de dispositivos. Se trata de una variante de la asignación de dispositivos que combina el acceso directo al dispositivo y la compartición por

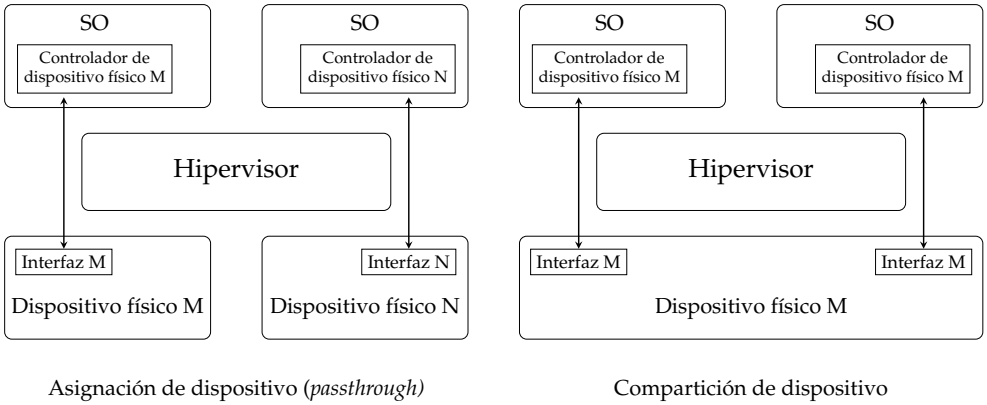
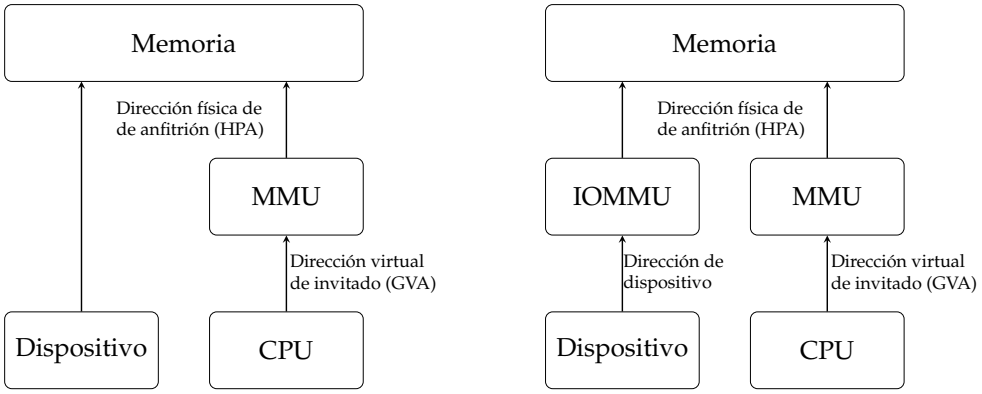


Figura 4.17: Virtualización de dispositivos por hardware

parte de varias máquinas virtuales. Requiere dispositivos que presenten múltiples interfaces, cada una de las cuales puede asociarse directamente a una máquina virtual diferente. Este es el caso de dispositivos PCI Express que incorporan la extensión *Single Root I/O Virtualization* (SR-IOV), que presentan multiples interfaces PCI Express independientes para un mismo dispositivo. Por ejemplo, una tarjeta de red PCI Express con cuatro interfaces podría asociarse a cuatro máquinas virtuales, cada una de las cuales la percibiría como una tarjeta de red independiente.

La implementación de la asignación de dispositivos, o su compartición, plantea una serie de problemas que deben resolverse por hardware. Para entenderlos se utilizará el ejemplo de una interfaz de red, la cual trabaja con DMA, representada en la figura 4.18a. En un momento dado, el sistema operativo que se ejecuta en una máquina virtual programa la interfaz de red para enviar una trama que se encuentra almacenada a partir de una dirección física. En el contexto de un entorno virtualizado esta dirección física es la dirección física de invitado (*Guest Physical Address* o GPA). El hipervisor no interceptará esta operación de configuración, por lo que la interfaz de red se configurará con esa dirección y leerá la trama a partir de esa dirección. Como la interfaz de red trabaja con direcciones físicas (reales), en realidad accederá a una dirección física de anfitrión (*Host Physical Address* o HPA) idéntica a la dirección física de invitado que programó el sistema operativo, provocando un funcionamiento incorrecto del sistema.

Para solucionar entre otros el problema anterior, los computadores modernos basados en la arquitectura x86 incluyen una unidad de gestión de memoria de E/S (*I/O Memory Management Unit* o IOMMU), análoga a la MMU para la memoria virtual, ubicada entre las interfaces de E/S y la memoria. La IOMMU traduce las direcciones que genera un dispositivo DMA en direcciones físicas de anfitrión (HPA) antes de que lleguen a la memoria. A esta técnica se la conoce también como remapeo de DMA (*DMA remapping*). La figura 4.18b muestra esta nueva situación. Cada dispositivo tiene asociada una tabla, análoga a la tabla de páginas, que emplea la IOMMU



(a) Sin remapeo de direcciones de dispositivos

(b) Con remapeo de direcciones de dispositivos

Figura 4.18: Estrategias de remapeo de dispositivos DMA

para traducir las direcciones GPA a HPA. Asimismo, para acelerar la traducción entre las direcciones GPA y HPA los dispositivos disponen de cachés de traducción denominadas IOTLB, análogos al TLB empleado en la paginación.

La inclusión de la IOMMU permite mejorar en gran medida el rendimiento en entornos virtualizados, pero al contrario de lo que ocurría con otras técnicas de virtualización asistidas por hardware, requiere soporte por parte del sistema operativo. No obstante, no suele ser un problema, pues la IOMMU se emplea también en entornos no virtualizados para introducir protección en los dispositivos. Al igual que ocurría con las tablas de páginas empleadas por la MMU, es posible establecer restricciones de acceso a memoria para los dispositivos de E/S, incrementando la seguridad del sistema ante un dispositivo «malicioso» o mal programado.

Apéndice A

Juego de instrucciones del simulador pyMIPS64

Nomenclatura utilizada

`RsX`	Registro entero de 64 bits que se utiliza como operando fuente.
`Rd`	Registro entero de 64 bits que se utiliza como operando destino.
`Ri`	Registro entero de 64 bits que se utiliza para el cálculo de una dirección en memoria.
`FsX`	Registro de punto flotante de 64 bits que se utiliza como operando fuente.
`Fd`	Registro de punto flotante de 64 bits que se utiliza como operando destino.
`imm_X`	Valor inmediato de X bits.
`sign_ext`	Extensión del signo hasta 64 bits.
`zero_ext`	Extensión hasta 64 bits añadiendo 0 por la izquierda.
`[]`	Posición de memoria.

El juego de instrucciones que aquí se presenta es el soportado por el simulador WinMIPS64, que es una simplificación del soportado por MIPS64. Además, algunos códigos de instrucción difieren. Las discrepancias que pudieran existir entre el juego de instrucciones de MIPS64 y el usado en WinMIPS64 se resaltarán de forma adecuada en las tablas siguientes.

Instrucciones aritméticas

Instrucción	Operación	Descripción
nop		Instrucción nula
daddi Rd, Rs, imm_16	Rd←Rs+sign_ext(imm_16)	Suma doble palabra y valor inmediato, generando excepción si hay desbordamiento
dadd Rd, Rs1, Rs2	Rd←Rs1+Rs2	Suma dobles palabras, generando excepción si hay desbordamiento
dsub Rd, Rs1, Rs2	Rd←Rs1−Rs2	Resta dobles palabras, generando excepción si hay desbordamiento
dmul[1] Rd,Rs1,Rs2	Rd←Rs1×Rs2	Multiplica dobles palabras
ddiv[1] Rd, Rs1, Rs2	Rd←Rs1÷Rs2	Divide dobles palabras
add.d Fd, Fs1, Fs2	Fd←Fs1+Fs2	Suma reales de precisión doble
sub.d Fd, Fs1 ,Fs2	Fd←Fs1−Fs2	Resta reales de precisión doble
mul.d Fd, Fs1, Fs2	Fd←Fs1×Fs2	Multiplica reales de precisión doble
div.d Fd, Fs1, Fs2	Fd←Fs1÷Fs2	Divide reales de precisión doble

Instrucciones lógicas

Instrucción	Operación	Descripción
andi Rd, Rs, imm_16	Rd←Rs AND zero_ext(imm_16)	AND lógico con valor inmediato
ori Rd, Rs, imm_16	Rd←Rs OR zero_ext(imm_16)	OR lógico con valor inmediato
xori Rd, Rs, imm_16	Rd←Rs XOR zero_ext(imm_16)	XOR lógico con valor inmediato
and Rd, Rs1, Rs2	Rd←Rs1 AND Rs2	AND lógico
or Rd, Rs1, Rs2	Rd←Rs1 OR Rs2	OR lógico
xor Rd, Rs1, Rs2	Rd←Rs1 XOR Rs2	XOR lógico
slt Rd, Rs1, Rs2	Si Rs1<Rs2 : Rd←1 si no : Rd←0	Comparación entera con signo

[1] En MIPS64 las instrucciones `dmult`, `dmultu`, `ddiv` y `ddivu` tienen solo dos operandos fuente y almacenan el resultado en sendos registros de 64 bits del procesador: `HI` y `LO`. En la multiplicación se obtiene un resultado de 128 bits que se guarda en ambos registros, mientras que en la división el cociente se almacena en `LO` y el resto en `HI`. Puede accederse a estos registros utilizando las instrucciones `mfhi Rd` y `mflo Rd`. En el caso del simulador pyMIPS64, el resultado de la multiplicación es de 64 bits y en la división solo se obtiene el cociente.

Instrucciones de carga

Instrucción	Operación	Descripción
ld Rd, imm_16(Ri)	Rd←[Ri+sign_ext(imm_16)]	Carga doble palabra
lb Rd, imm_16(Ri)	Rd←sign_ext([Ri+sign_ext(imm_16)]$_{[7..0]}$)	Carga byte con extensión de signo
lh Rd, imm_16(Ri)	Rd←sign_ext([Ri+sign_ext(imm_16)]$_{[15..0]}$)	Carga media palabra con extensión de signo
lw Rd, imm_16(Ri)	Rd←sign_ext([Ri+sign_ext(imm_16)]$_{[31..0]}$)	Carga palabra con extensión de signo
lbu Rd, imm_16(Ri)	Rd←zero_ext([Ri+sign_ext(imm_16)]$_{[7..0]}$)	Carga byte sin extensión de signo
lhu Rd, imm_16(Ri)	Rd←zero_ext([Ri+sign_ext(imm_16)]$_{[15..0]}$)	Carga media palabra sin extensión de signo
lwu Rd, imm_16(Ri)	Rd←zero_ext([Ri+sign_ext(imm_16)]$_{[31..0]}$)	Carga palabra sin extensión de signo
l.d Fd, imm_16(Ri)	Fd←[Ri+sign_ext(imm_16)]	Carga real de precisión doble

Instrucciones de almacenamiento

Instrucción	Operación	Descripción
sd Rs, imm_16(Ri)	[Ri+sign_ext(imm_16)]←Rs	Almacena doble palabra
sb Rs, imm_16(Ri)	[Ri+sign_ext(imm_16)]←Rs$_{[7..0]}$	Almacena byte
sh Rs, imm_16(Ri)	[Ri+sign_ext(imm_16)]←Rs$_{[15..0]}$	Almacena media palabra
sw Rs, imm_16(Ri)	[Ri+sign_ext(imm_16)]←Rs$_{[31..0]}$	Almacena palabra
s.d Fs, imm_16(Ri)	[Ri+sign_ext(imm_16)]←Fs	Almacena real de precisión doble

Instrucciones de salto incondicional

Instrucción	Operación	Descripción
j imm_26	PC←PC$_{63..28}$|imm_26<<2	Salta a dirección dentro de una zona de 256 MiB (2^{28})

Instrucciones de salto condicional

Instrucción	Operación	Descripción
beq Rs1, Rs2, imm_16	Si Rs1=Rs2 : PC←PC+sign_ext(imm_16<<2)	Salta si son iguales a dirección dentro de zona de ±128 kiB
bne Rs1, Rs2, imm_16	Si Rs1≠Rs2 : PC←PC+sign_ext(imm_16<<2)	Salta si son distintos a dirección dentro de zona de ±128 kiB
beqz Rs, imm_16	Si Rs=0 : PC←PC+sign_ext(imm_16<<2)	Salta si es 0 a dirección dentro de zona de ±128 kiB
bnez Rs, imm_16	Si Rs≠0 : PC←PC+sign_ext(imm_16<<2)	Salta si no es 0 a dirección dentro de zona de ±128 kiB

Apéndice B

La unidad de control

En este apéndice se diseña la unidad de control necesaria para gestionar el camino de datos presentado en el libro.

B.1. Unidad de control monociclo

El control de todas las operaciones que se llevan a cabo sobre el camino de datos corre a cargo de la unidad de control. Existen, principalmente, dos tipos de implementaciones de una unidad de control: cableada y microprogramada. En una unidad de control cableada cada señal de control se obtiene como resultado de una función lógica que tiene como entradas el código de instrucción y los *flags* generados por la ALU. Se diseña e implementa como cualquier circuito combinacional. En cambio, en una unidad microprogramada el estado de todas las señales de control se representa mediante una palabra de control. La unidad de control dispone de una memoria interna donde almacena estas palabras de control y cada código de instrucción referencia a una de estas palabras (para el caso monociclo).

Indepedientemente de la forma de implementación elegida, la labor de la unidad de control es la misma: activar y desactivar las señales de control que guían la ejecución de las instrucciones sobre el camino de datos. Por simplicidad, se dejará a un lado la implementación y solo se planteará el diseño a través de la tabla de verdad que debe satisfacer la unidad de control.

Una técnica muy habitual para mejorar el rendimiento de la unidad de control, y en general de la propia CPU, es dividir el control en varias unidades. Se puede plantear inicialmente una unidad de control reducida para gestionar la ALU. Esta se encargaría de generar las señales de control (4 líneas) que llegan a la ALU en función de la instrucción a ejecutar.

Las instrucciones de carga y almacenamiento necesitan utilizar la ALU para calcular la dirección de memoria a leer/escribir como el resultado de la suma del registro base y el valor inmediato. Las operaciones aritméticas y lógicas son de tipo R y tienen todas el mismo valor en el campo `op`, por lo que únicamente se distinguen por el campo `función`, tal como se ilustra en la figura 2.5. Utilizan la ALU según

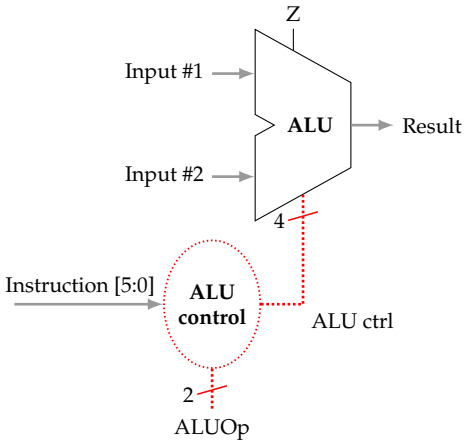


Figura B.1: Unidad de control de la ALU

Instrucción	Campo `op`	Campo `función`	Operación ALU	ALUOp	ALU ctrl
`ld`	carga	–	suma	`00`	`0010`
`sd`	almacenamiento	–	suma	`00`	`0010`
`beq`	salto si igual	–	resta	`01`	`0110`
`daddi`	suma inm. (I)	–	suma	`00`	`0010`
`dadd`	tipo R	`100000`	suma	`10`	`0010`
`dsub`	tipo R	`100010`	resta	`10`	`0110`
`and`	tipo R	`100100`	*and*	`10`	`0000`
`or`	tipo R	`100101`	*or*	`10`	`0001`
`slt`	tipo R	`101010`	*set on less than*	`10`	`0111`

Tabla B.1: Operación de la ALU en función de las señales de control

la función especificada (suma, resta, *and*, *or*, etc.). Por último, la instrucción de salto condicional utiliza la ALU para comparar dos registros y comprobar si se satisface la condición.

Bajo estas condiciones, es posible definir una señal de control de dos líneas, denominada `ALUOp`, que permita a la unidad de control indicarle a la unidad de control de la ALU la operación a realizar: `00` para la suma, `01` para la resta y `10` para realizar la operación indicada por el campo `función` de una instrucción aritmético-lógica. Así, la unidad de control reducida para la ALU quedaría tal como indica la figura B.1. Asimismo, en la tabla B.1 se enumeran las instrucciones, del subconjunto de MIPS64 implementado, que hacen uso de la ALU y los valores necesarios de las señales de control en función de los campos `op` y `función` del código de instrucción.

Para el diseño de la unidad de control principal deben considerarse todas las señales de control que se han ido añadiendo al camino de datos durante su construcción. Muchas de ellas gobiernan multiplexores para seleccionar entre dos fuentes de datos. Al tratarse de multiplexores con solo dos entradas, basta una señal de control de una sola línea para gobernarlos. En la tabla B.2 se muestra un listado con las señales de control que aparecen en el camino de datos de la figura 2.16 y el efecto que producen cuando valen `1` o `0`.

Señal	Efecto desactivada (0)	Efecto activada (1)
RegDst	El número de registro destino viene de los bits 20 a 16	El número de registro destino viene de los bits 15 a 11
RegWrite	Ninguno	Se escribe el registro indicado con Write register con los datos Write data
ALUSrc	El segundo operando de la ALU viene del segundo registro	El segundo operando de la ALU viene del valor inmediato extendido a 64 bits
PCSrc	Se calcula el valor del PC como PC + 4	Se calcula el valor del PC como la dirección de salto condicional
Jump	El registro PC se carga con el valor ya calculado	El registro PC se carga con la dirección de salto incondicional
MemRead	Ninguno	Se solicita la lectura de la dirección de memoria indicada en Address
MemWrite	Ninguno	Se solicita la escritura de la dirección de memoria indicada en Address con Write data
MemToReg	El valor a escribir en el fichero de registros viene de la ALU	El valor a escribir en el fichero de registros viene de la memoria de datos (Read data)

Tabla B.2: Señales de control generadas por la unidad de control

Inst.	RegDst	RegWrite	ALUSrc	Branch	Jump	MemRead	MemWrite	MemToReg	ALUOp
tipo R	1	1	0	0	0	0	0	0	10
ld	0	1	1	0	0	1	0	1	00
sd	X	0	1	0	0	0	1	X	00
beq	X	0	0	1	0	0	0	X	01
daddi	0	1	1	0	0	0	0	0	00
j	X	0	X	X	1	0	0	X	XX

Tabla B.3: Valores de las señales de control para cada instrucción

A partir del comportamiento del camino de datos en la ejecución de las instrucciones descrito a lo largo de la sección 2.2.2, puede inferirse el valor que deben tomar las distintas señales. En la tabla B.3 se muestran los valores que toman para las diferentes instrucciones implementadas (una X indica que el valor de la señal puede ser indistintamente 1 o 0). Debe tenerse en cuenta que en la tabla se considera la señal Branch en lugar de la genérica PCSrc.

Por tanto, el camino de datos con el control necesario quedaría como se refleja en la figura B.2. A continuación, a modo de ejemplo, se enumeran las señales de control generadas durante la ejecución de la siguiente instrucción:

```
dadd r2, r3, r6
```

La ejecución de esta instrucción requiere un único ciclo de reloj donde se siguen las siguientes operaciones:

- Tan pronto como se produce el flanco ascendente en la señal de reloj se busca en la memoria de instrucciones el código de la instrucción a ejecutar usando la dirección contenida en PC.
- El código de instrucción pasará a estar disponible en las líneas Instruction.
- Se leen los registros (**r3** y **r6**) en el fichero de registros (Read #1 y Read #2).
- Los contenidos de los registros pasarán a estar disponibles en las líneas Read data 1 y Read data 2.

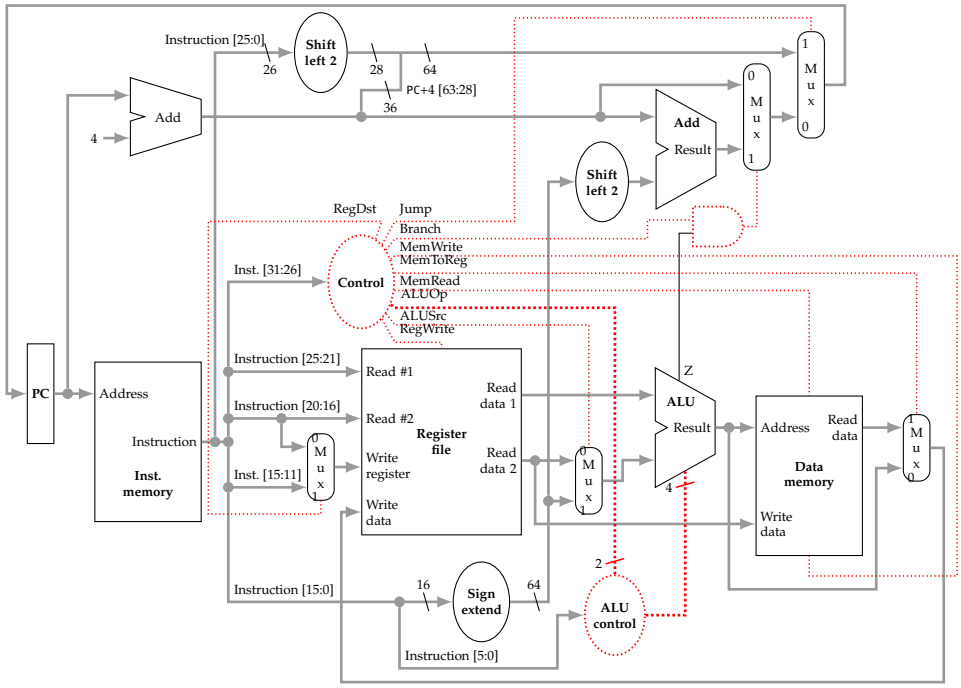


Figura B.2: Camino de datos para la ejecución de instrucciones de tipo R, acceso a memoria, salto condicional **`beq`** y salto incondicional `j` con la lógica de control

- La línea `ALUSrc` está a `0` para indicar que se opera sobre el contenido de ambos registros.
- Se envía la combinación `10` a las líneas `ALUOp` para indicar a la unidad de control de la ALU que se trata de una instrucción de tipo R. Esta última, en base al valor del campo `función` del código de instrucción, envía a su vez la combinación `0010` a la ALU para que realice la suma de los operandos.
- La línea `MemToReg` está a `0` para indicar que el valor a escribir en el fichero de registros es el resultado de la ALU.
- La línea `RegDst` está a `1` para indicar que el número de registro destino viene dado por los bits 15 a 11 del código de instrucción.
- La línea `RegWrite` está a `1` para escribir en el fichero de registros.
- Paralelamente, las líneas `Branch` y `Jump` están a `0` para que el contador de programa se incremente en 4, lo mismo que las líneas asociadas a la memoria de datos, `MemRead` y `MemWrite`, ya que no se hace uso de la memoria.

B.2. Unidad de control segmentada

Para implementar la unidad de control segmentada se partirá de la unidad de control monociclo vista y el camino de datos mostrado en la figura 2.20. En la figura B.3 se muestra el camino de datos con las señales de control identificadas, así como una pequeña modificación para poder ejecutar correctamente las instrucciones que almacenen su resultado en el fichero de registros (carga y aritmético-lógicas). El problema viene de que estas instrucciones requieren la parte del código de instrucción que indica el registro a escribir en el fichero de registros durante la etapa `WB`. Por esta razón, es necesario propagar a lo largo del camino de datos, utilizando los registros de segmentación, las partes del código de instrucción necesarias en etapas posteriores a `ID`. Esta modificación incrementa el ancho del registro de segmentación `ID/EX` que pasa de 256 bits a 266 bits.

De esta representación del camino de datos se desprende que casi todas las instrucciones se comportan de la misma forma durante las dos primeras etapas (`IF` e `ID`). Las instrucciones se cargan en `IF` y se decodifican en `ID`. Por esta razón, en la primera etapa y parte de la segunda no se conoce qué instrucción se está ejecutando. Solo durante la etapa `ID` se identifica el tipo de instrucción a ejecutar. Un caso excepcional es la instrucción de salto incondicional. Durante la etapa `ID` se calcula la dirección de destino del salto y, tan pronto como se decodifica la instrucción y se identifica el salto, se modifica el registro contador de programa. No obstante, teniendo en cuenta que no se identifica la instrucción hasta que no se decodifica, la dirección de salto se calcula para todas las instrucciones, independientemente de que se trate de un salto incondicional o no.

Las señales de control que se generan para ejecutar una instrucción deben propagarse por el camino de datos de la misma forma que se propagan los datos. Así, es necesario incrementar los tamaños de los registros de segmentación para incluir estas señales de control, que son generadas durante la etapa de decodificación de la instrucción `ID`. Como las etapas `IF` e `ID` son iguales para todas las instrucciones, salvo para el salto incondicional, la mayor parte de señales se propagan a lo largo de los registros de segmentación `ID/EX`, `EX/MEM` y `MEM/WB`, tal como se ilustra en la figura B.4.

El camino de datos segmentado resultante, incluyendo la unidad de control, es el que se muestra en la figura B.5. Esta implementación asume que no existen riesgos de la segmentación, y que las instrucciones se pueden ejecutar idealmente con un CPI de 1.

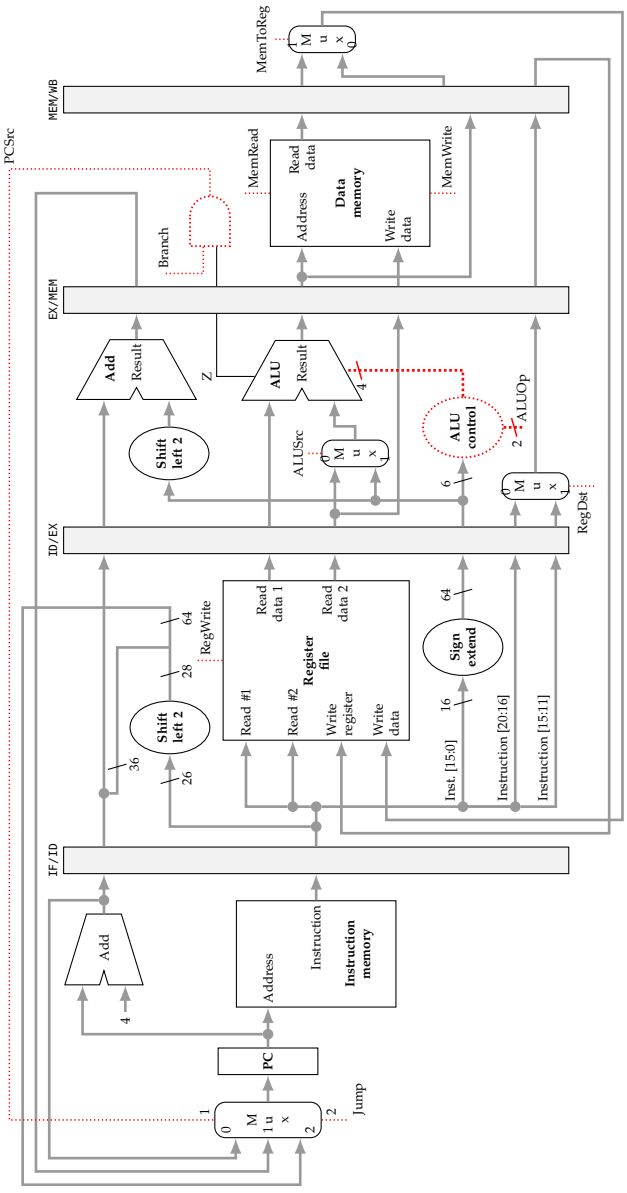


Figura B.3: Camino de datos segmentado identificando las señales de control

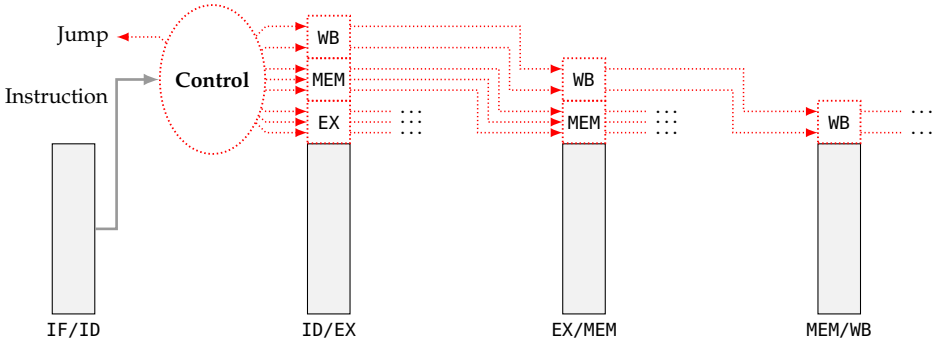


Figura B.4: Propagación de las señales de control en el camino de datos segmentado

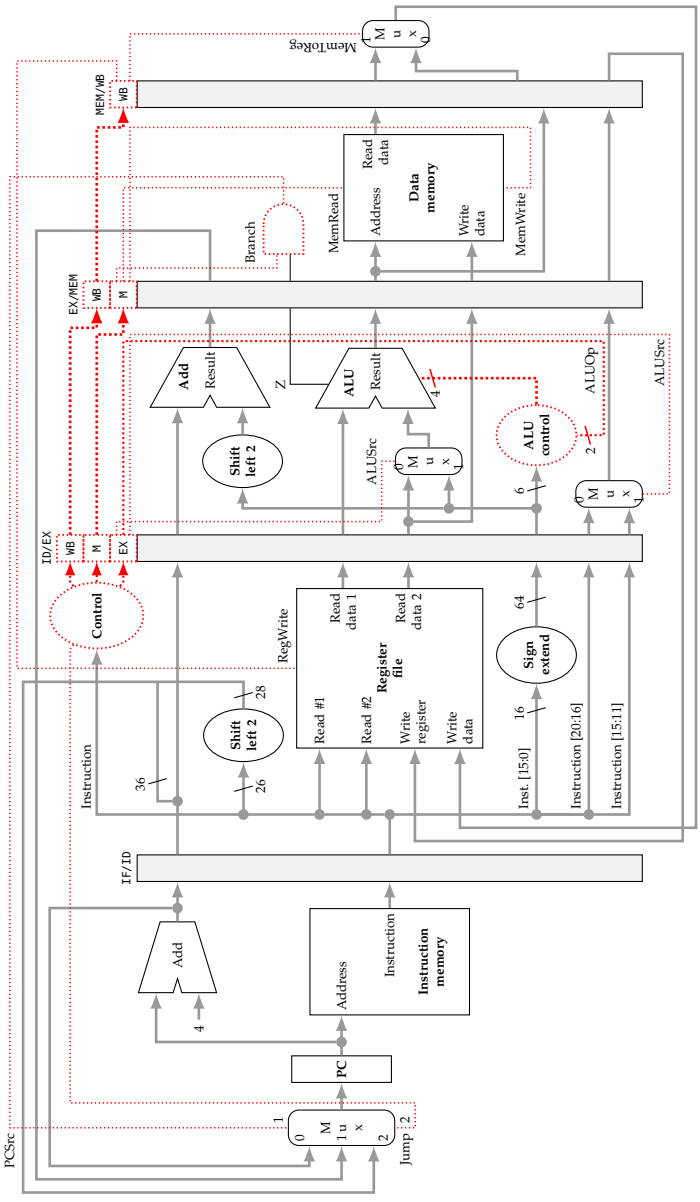


Figura B.5: Camino de datos segmentado y unidad de control asumiendo que no existen riesgos de la segmentación

Bibliografía

[1] D.A. Patterson, J.L. Hennessy. *Computer organization and design. The hardware/-software interface, 5th edition*. Morgan Kaufmann, 2014. ISBN: 978-0124077263.

[2] J.L. Hennessy, D.A. Patterson. *Computer architecture. A quantitative approach, 5th edition*. Morgan Kaufmann, 2012. ISBN: 978-0123838728.

[3] G.E. Moore. *Moore's law at 40*. Understanding Moore's law: four decades of innovation. Chemical Heritage Foundation, 2006. ISBN: 978-0941901413.

[4] Julio Ortega, Mancia Anguita, Alberto Prieto. *Arquitectura de computadores*. Paraninfo, 2005. ISBN: 849-7322746.

[5] W. Stallings, A.C. Vargas. *Organización y arquitectura de computadores: diseño para optimizar prestaciones*. Prentice Hall, 2001. ISBN: 978-8420529936.

[6] R.R. Schaller. *Moore's law: past, present and future*. IEEE Spectrum, vol. 34(6), pp. 52–59, 1997.

[7] V.C. Hamacher, Z.G. Vranesic, S.G. Zaky, M.L.F. García, G.Q. Vieyra. *Organización de computadoras*. McGraw-Hill, 1987. ISBN: 968-4220588.

Arquitectura de Computadores

Hoy en día, los computadores están en todas partes. Es posible encontrarlos en infinidad de situaciones y aplicaciones; desde los tradicionales ordenadores personales, pasando por los teléfonos inteligentes hasta sistemas empotrados para el control de procesos. Estos últimos son el hilo conductor de una nueva revolución digital que está permitiendo, entre otros desarrollos, las ciudades inteligentes, la conducción autónoma y el internet de las cosas.

El objetivo de este libro es presentar las técnicas que implementan los computadores actuales bajo tres perspectivas: la mejora del rendimiento, el soporte a los sistemas operativos multitarea y el soporte a la virtualización. Para ello, se organizan los contenidos en cuatro grandes capítulos: una introducción a la arquitectura de los computadores, la CPU, la jerarquía de memoria y el sistema de entrada/salida.

Este libro está pensado para servir como bibliografía en asignaturas universitarias sobre Arquitectura de Computadores, dentro de los estudios para obtener el Grado en Ingeniería en Informática, donde se asume un conocimiento básico del funcionamiento del computador.

ISBN 978-84-17445-49-2

ediuno

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo