Arquitectura de Computadoras (notas de clase)

José Galaviz Casas

Departamento de Matemáticas
Universidad Nacional Autónoma de México

Índice General

CAPÍTULO 1

Introducción

1.1 Tecnología y diseño

A fines de la década de los 50's en el siglo XX una computadora PDP-8 de *Digital Equipment Corporation* (la desaparecida DEC, ahora parte de *Compaq*), costaba unos $18,000 dolares y poseía 4K palabras de memoria de 12 bits, el equivalente a 6 KB[1]. y podía hacer unas 35,000 operaciones por segundo. Hoy en día es posible adquirir por unos $1,500 dolares una computadora con 128 KB. de memoria y capaz de hacer varios millones de operaciones por segundo. Además esta computadora puede ponerse sobre una mesa de 50×50 cm. con todo y monitor, cosa que no podía haberse hecho con la PDP-8.

[1]Denotaremos con "KB" el término *kilo bytes* (1024 octetos de bits) mientras que con "Kb" denotaremos el término *kilo bits* (1000 bits).

Es notable el decremento en el costo y en el tamaño, y el incremento en las capacidades que han sufrido nuestros dispositivos de cómputo durante las últimas décadas. Este incremento en la capacidad redunda en un incremento en el desempeño del equipo, qué tanto puede hacer y qué tan rápido.

Desempeño durante la década de 1970.

Durante la década de 1970 el desempeño de las computadoras creció a una tasa del 25% al 30% anual [3]. La causa principal de este fenómeno fueron las mejoras sustanciales en la tecnología de circuitos integrados. Estas mejoras hicieron posible la producción masiva de microprocesadores y por ende una disminución en los costos. Si un producto tiene mucha damanda, se vende mucho, entonces se producen más unidades de él. Si para producirlo se hizo una cierta inversión inicial x, entonces, mientras mayor sea el volumen de unidades producidas, menor será el precio de cada una de ellas, dado que el costo de producción se reparte entre más piezas (compárese el precio de un VW sedán, del que se producen millones de unidades, con el de un Rolls Royce hecho sobre pedido).

También en esa época comenzaron a surgir sistemas operativos estándar (como Unix), de forma que diferentes computadoras, con diferente hardware y diferente arquitectura, corrían el mismo sistema, ofrecían la misma interfaz al usuario y el programador. Unix es Unix, ya sea en una IBM o en una HP.

Otro fenómeno que comenzó a ocurrir por ese entonces fue la virtual desaparición de la programación en lenguaje ensamblador. Sólo se continuo con esta práctica en las computadoras personales hasta fines de la década de 1980.

Estos tres factores: el abaratamiento de los procesadores, la estandarización de sistemas operativos y la desaparición de la programación en ensamblador fueron factores claves para impulsar la creación de nuevas arquitecturas y nuevos paradigmas de diseño de computadoras a partir de los 80's.

Si es más barato producir un procesador nuevo, si se debe ofrecer una interfaz estándar sin importar como está hecho el hardware que subyace en la computadora, y si no hay necesidad de mantener compatibilidad de código objeto con las máquinas anteriores, porque nadie programa de forma tan específica, entonces podemos darnos el lujo de experimentar nuevas cosas.

Desempeño durante la década de 1980

De esta manera surgió, por ejemplo, el paradigma RISC (*Reduced Instruction Set Computer*) entre otras innovaciones lo que redundó en un crecimiento superior al 50% anual en el desempeño de las computadoras en la década de 1980 [3]. Si bien el crecimiento de desempeño en los 70's es

atribuible a los avances tecnológicos en los 80's se debió a el surgimiento de nuevas alternativas de diseño.

Consecuencias:

- La capacidad de cómputo que poseen los usuarios actuales (que por otra parte no son lo usuarios típicos de el pasado) es sorprendente.
- El microprocesador ha establecido su dominio tecnológico, lo que ha hecho que las computadoras personales y estaciones de trabajo desplacen a las minicomputadoras y *mainframes* del pasado.

La libertad de no-compatibilidad con diseños viejos y el uso de la tecnología de microporcesadores trajo consigo la innovación arquitectónica (ruptura con el pasado) y el uso eficiente de los recursos tecnológicos (hay nuevas tecnologías aprovechables si se cambia el diseño).

El crecimiento actual del desempeño en los equipos de cómputo se debe a las innovaciones en el diseño más que a las mejoras tecnológicas (que cada vez es más difícil superar). En esta revolución el actor principal son los nuevos métodos de diseño y análisis basados en la observación empírica: experimentación y simulación.

1.2 La labor del diseñador

Dadas:

- Las restricciones de costo estipuladas para un nuevo equipo de cómputo, dictadas por el mercado al que va dirigido.
- Las limitaciones tecnológicas inherentes a los dispositivos a utilizar (dictados también por el punto anterior).

el diseñador debe determinar que atributos son importantes en la computadora, cuales son críticos en su impacto en el desempeño y como maximizar este. Para lograr esto debe afrontar diversas subtareas:

1. El diseño del conjunto de instrucciones.

2. La organización funcional de los diversos componentes.

3. El diseño lógico: interconexiones entre los diversos componentes, circuitos.

4. La implementación: diseño de los circuitos integrados, empaquetado de los mismos, determinación de la capacidad de las fuentes de poder y los dispositivos de enfriamiento.

La interacción hardware-software.

Maximizar el desempeño implica una clara comprensión de una gran variedad de cosas, desde circuitos y diseño lógico hasta sistemas operativos y compiladores. A fin de cuentas es la perfecta coordinación *hardware-software* la que se refleja en el desempeño.

En el pasado el término arquitectura de computadoras significaba esencialmente diseño del conjunto de instrucciones (con base en *qué* se quiere hacer se decide *como* hay que hacerlo). Actualmente es bastante más complicado que eso, hay que estar familiarizado con el funcionamiento de los compiladores y de los sistemas operativos además de saber diseño lógico; sólo tomando en consideración esos factores se puede lograr hoy en día una arquitectura de buen desempeño.

Arquitectura de computadoras no es arquitectura de conjunto de instrucciones solamente.

Hoy en día una arquitectura es mucho más que el simple diseño del conjunto de instrucciones, de hecho hay arquitecturas completamente diferentes que utilizan el mismo procesador y por tanto el mismo conjunto de instrucciones (p.ej. las Macintosh con procesador Motorola 68020 y las NeXT hoy desaparecidas).

1.3 Tendencias tecnológicas y tendencias en el uso

Para que una arquitectura sobreviva debe ser útil, esto es, debe adaptarse a los cambios en la tecnología de hardware y software. El diseñador debe estar prevenido y bien informado acerca de las tendencias en el uso de las computadoras y la tecnología disponible en el momento y el futuro. Debe planear considerando como los cambios tecnológicos pueden prolongar la vida útil de una arquitectura exitosa.

En el diseño de una computadora se ve afectado fundamentalmente por dos factores:

- Como será usada.
- Características de la tecnología de implementación.

Tendencias de uso. Los programas han tendido a lo largo del tiempo a usar cada vez más memoria. Esto es consecuencia de dos cosas:

- Las tareas que efectúan los programas son cada vez más complejas por lo que requieren más memoria. Hace años, por ejemplo, no había interfaces gráficas que son sumamente demandantes.
- Las mejoras en la tecnología de DRAM[2]. Hoy en día cabe más memoria en menos espacio y a menor costo.

Desestimar el tamaño de la memoria accesible al procesador, el tamaño del espacio de direcciones, es una de las causas más frecuentes por las que una arquitectura es tirada a la basura. Por ejemplo el 8086 de Intel (1978) sólo podía direccionar 1 Mb. de memoria.

Arquitectura de computadoras y diseño de compiladores.

En los últimos 25 años paulatinamente se ha reemplazado la programación en lenguaje ensamblador por la de alto nivel. Esto ha hecho que las necesidades de los diseñadores de compiladores hayan cobrado mayor importancia lo que ha traído como consecuencia, la incorporación de características que permiten un comportamiento óptimo de los compiladores en las arquitecturas de las computadoras.

Tendencias tecnológicas.

- En circuitos integrados. La densidad crece un 50% cada año [3], crece la velocidad, lo único que no crece es el desempeño del cableado.
- Semiconductores de DRAM. La densidad se cuadruplica cada 3 años, el tiempo de respuesta mejora 33% cada 10 años [3].
- Tecnología de discos magnéticos. En 1982 los discos eran de 20 Mb. hoy en día son de unos 20 Gb. la densidad ha crecido en un 50% por año.

[2]Memoria RAM dinámica, esencialmente hecha de condensadores que requieren refrescar su estado con cierta frecuencia.

Esto hace que un procesador este vigente en el mercado por unos 5 años más o menos. La tecnología tiene mucho impacto en el diseño. Fue hasta que se pudieron poner de 25 a 50 mil transistores en un solo chip cuando se pudo pensar en hacer microprocesadores de 32 bits, antes no, porque la lógica necesaria no hubiera cabido en una pastilla de tamaño razonable.

1.4 Costo y tendencias en el costo.

Hay circunstancias en las que el costo no es un factor relevante en el diseño de una computadora; cuando se diseña una supercomputadora, por ejemplo. Pero esto no ocurre en general, las computadoras se hacen para venderse, mientras más se vendan mejor, así que el precio debe ser razonable para un amplio mercado (generalmente) lo que implica que el costo de producción también debe serlo.

En general el costo de manufactura de los componentes de la computadora ha caído a lo largo del tiempo. Por ejemplo el costo de la DRAM ha caído un 40% cada año [3].

Otro factor que tiene impacto en el costo es, por supuesto, el volumen de producción. Como hemos dicho, cuanto mayor sea el volumen producido tanto menor será el costo y el precio tenderá a disminuir más rápido porque se amortiza el costo de manufactura a tasas más elevadas.

La competencia es otro factor clave del decremento en el costo.

Parte importante del costo de un equipo de computo se deriva del costo de producción de los circuitos integrados que contiene. Para comprender este factor debemos primero saber como es producido un circuito integrado [3, 1].

1.4.1 Proceso de producción de circuitos integrados

1 Se trituran los pedruscos de mineral de silicio.

2 Se funde el polvo y se destila para eliminar impurezas. El resultado son barras de silicio muy puro pero molecularmente desordenado.

3 La barra se funde nuevamente a $1,400^o\,C$ y se convierte en un cilindro de monocristal de estructura molecular muy organizada de 8 pulgadas de

diámetro.

4 El cilindro se corta con una sierra laser en rebanadas muy delgadas, que luego se pulen para dejarlas completamente planas. El resultado de este proceso son obleas de silicio.

5 Se procede a imprimir los circuitos sobre la oblea. Se imprimen tantos circuitos (cuadrados) como sea posible en una oblea (circular). El proceso consiste de 20 o 30 pasos en los que se hace cíclicamente lo siguiente:

1.1 Se deja formar una capa de dióxido de silicio sobre la oblea.

2.2 Se recubre la capa de oxido con una película fotorresistente.

3.3 Se imprime el circuito, mediante proceso fotográfico en ultravioleta. Donde la luz choca con la película se endurece, donde no queda blanda.

4.4 Las zonas blandas de la película se disuelven en un agente revelador.

5.5 Las zonas de oxido donde la película ha sido removida se disuelven en ácido fluorhídrico. las zonas cubiertas por la película fotorresistente permanecen.

6 Las obleas tienen ahora impresa una matriz de estampas, cada estampa es en esencia un chip individual. Las obleas son circulares y las estampas son cuadradas, así que las estampas incompletas de la orilla son desperdicio. A las obleas estampadas se les denomina *wafer*.

7 Se corta cada estampa individualmente de la matriz.

8 Se pasa cada estampa por un probador que en un lapso de 5 a 90 segundos la prueba. El costo de uso del probador es de unos \$300 a \$500 dolares la hora. La eficiencia de salida de esta etapa es de un 50% en el mejor de los casos. Por ejemplo el cociente (*estampas buenas*)/(*estampas totales*) es 0.15 para un SuperSPARC y 0.48 en MIPS [3].

9 Se desechan las estampas defectuosas y se pasan las buenas al proceso de empacado. Algunas se echan a perder en esta etapa.

a Empaque cuadrado plano de plástico (*plastic quad flat pack*) PQFP, a lo más 208 pins, $1\,cm^2$ y 1 watt, \$2 dolares.

b Arreglo de cuadricula de patas de cerámica (*pin grid array*) PGA. de 300 a 600 pins, \$20 a \$60 dolares.

10 Los chips completos se pasan a una nueva etapa de prueba, se desechan los defectuosos. La eficiencia en esta etapa es de, al menos, un 95% (buenos/total=0.95) [3].

11 Los chips se embarcan.

Por supuesto no todas las estampas impresas en la oblea (paso 5) sirven. Esto se debe a dos razones fundamentalmente:

- Las estampas que quedan truncadas por el borde circular de la oblea no sirven, tal como se indica en el paso 6.
- Algunos errores en el proceso de impresión hacen que algunas estampas, aunque completas, no sirvan.

Suponiendo que las estampas son cuadradas, el número de estampas que quedan en la orilla de la oblea N_{orilla} son aproximadamente [3][3]:

$$N_{orilla} \approx \frac{circunferencia_{oblea}}{diagonal_{estampa}} = \frac{\pi\, d}{\sqrt{2\, A_{estampa}}}$$

donde d es el diámetro de la oblea, $A_{estampa}$ es el área de la estampa y por tanto $\sqrt{2\, A_{estampa}}$ es la longitud de la diagonal de la estampa.

Así que el número de estampas en cada oblea es:

$$N = \frac{A_{oblea}}{A_{estampa}} - N_{orilla} \approx \frac{\pi\, \left(\frac{d}{2}\right)^2}{A_{estampa}} - \frac{\pi\, d}{\sqrt{2\, A_{estampa}}} \tag{1.4.2}$$

N es el número total de estampas completas por oblea. A esta cantidad habría que quitarle el número de estampas defectuosas en la oblea debido al proceso de impresión para obtener el número total de estampas útiles por oblea N_u.

Llamaremos la eficiencia de estampa ($E_{estampa}$) a la siguiente cantidad

$$E_{estampa} = \frac{N_u}{N} \tag{1.4.3}$$

por supuesto esto es un número entre 0 y 1 y en la vida real es inferior a 0.5.

[3]Resultado empírico.

No sólo las estampas buenas cuestan, también se gasta dinero produciendo las incompletas y las defectuosas y el monto gastado en ellas, que constituye el costo de la oblea completa, debe ser recuperado mediante la venta de las estampas buenas en la oblea (N_u). Esta cantidad la podemos obtener de la expresión 1.4.3 y entonces, el costo de estampa ($C_{estampa}$) es:

$$C_{estampa} = \frac{C_{oblea}}{N_u} = \frac{C_{oblea}}{N\,E_{estampa}} \tag{1.4.4}$$

donde C_{oblea} es el costo de producción de una oblea estampada (*wafer*).

A este costo hay que añadir, por supuesto, el costo de probar las estampas (C_t), costo que, nuevamente, debe ser absorbido por los chips buenos al final del proceso.

$$C_t = \frac{Costo\ por\ hora\ de\ prueba\ \times\ tiempo\ promedio\ de\ prueba\ por\ estampa}{E_{estampa}} \tag{1.4.5}$$

En síntesis, el costo de producción de un circuito integrado es:

$$C_c = \frac{C_{estampa} + C_t + C_p}{E_f} \tag{1.4.6}$$

donde:

$C_{estampa}$ Costo de cada estampa.

C_t Costo de prueba de cada estampa.

C_p Costo de empaque y prueba final (del chip ya empacado).

E_f Eficiencia de prueba final. Es decir el número total de chips correctos entre el número total de estampas buenas. Este número, nuevamente, es a lo más uno. Si no es uno se debe a que algunas estampas buenas se estropean durante el proceso de empaque. En la vida real este número es cercano a 0.95.

Un modelo empírico sencillo para la eficiencia de estampa ($E_{estampa}$), suponiendo que los defectos en la oblea se distribuyen uniformemente es [3]:

$$E_{estampa} = E_{oblea}\left(1 + \frac{D_{oblea}}{\alpha}\right)^{-\alpha} \tag{1.4.7}$$

donde: α es un parámetro que depende de la tecnología de producción de las estampas, está relacionado con el número de capas que estas poseen[4] y D_{oblea} es el número de defectos en la oblea. Este número de defectos puede escribirse en términos del número de defectos (promedio) por unidad de área (D_A) durante el proceso de impresión de las estampas:

$$D_{oblea} = D_A \, A_{oblea}$$

usando esto en 1.4.7 se obtiene:

$$E_{estampa} = E_{oblea} \left(1 + \frac{D_A \, A_{oblea}}{\alpha} \right)^{-\alpha} \tag{1.4.9}$$

Típicamente hay de 0.6 a 1.2 defectos por centímetro cuadrado [3].

1.5 Medida del desempeño

Hay dos tipos diferentes de competencias de ciclismo, a saber:

1. Una vuelta ciclista, en la que todos los competidores salen, al mismo tiempo, a recorrer una distancia fija. El que la recorre en el menor tiempo gana.
2. Un ciclista, solo, recorre un circuito corto repetidamente durante un periodo de tiempo fijo. El objetivo es recorrer la mayor distancia posible.

Tiempo de respuesta.

En el primer caso hay una tarea fija, el que la ejecute más rápidamente es el ganador. El análogo en un sistema de cómputo es el el tiempo de respuesta o de ejecución (*response time*).

Rendimiento (*throughput*).

En el segundo caso gana quien sea capaz de ejecutar más trabajo en un tiempo dado. El análogo en sistemas de cómputo es llamado rendimiento (*throughput*).

Generalmente los usuarios están muy interesados en tener sistemas con un buen desempeño en términos de la primera opción, mientras que a los administradores de sistemas les interesa más la segunda opción.

[4] De hecho está relacionado con el número de capas interconectadas, lo que es una medida de complejidad *topológica* del chip. Para CMOS con 3 capas de interconexión $\alpha = 3$.

Por supuesto para que una medida tenga sentido debe estar referida a algo. Medir es, esencialmente, comparar dos cosas. Así que generalmente nos interesa comparar el desempeño de dos máquinas X y Y, nos interesa poder decir cuando "X es más rápida que Y".

En los términos que hemos mencionado cuando digamos "X es n más rápida que Y" significa que si X y Y ejecutan una misma tarea:

$$\frac{Tiempo\ de\ ejecución\ en\ Y}{Tiempo\ de\ ejecución\ en\ X} = n$$

Hay que notar que menor tiempo significa mejor desempeño, es decir son inversamente proporcionales el tiempo de ejecución y el desempeño. Podríamos poner la expresión anterior como:

$$\frac{Desempeño\ de\ X}{Desempeño\ de\ Y} = n$$

Cuando digamos "el *througput* de X es 1.3 veces más alto que el de Y" significa que el número de tareas completadas por X en una cierta unidad de tiempo es 1.3 veces el número de tareas completadas por Y en el mismo tiempo.

Malas métricas.

Ya sea que estemos interesados en *tiempo de ejecución* o en *rendimiento* lo que es importante medir es el tiempo. Desafortunadamente el las comparaciones que nos encontramos frecuentemente no siempre se utiliza el tiempo como la métrica. Se han inventado otras medidas y algunas más se han sacado del contexto en el que fueron originalmente usadas para ser utilizadas como medidas de desempeño que carecen de consistencia y confiabilidad.

Un ejemplo de tales medidas son los MIPS (millones de instrucciones por segundo):

$$MIPS = \frac{Número\ de\ instrucciones}{Tiempo\ de\ ejecución\ \times 10^6} = \frac{velocidad\ del\ reloj}{CPI \times 10^6}$$

donde CPI (*clocks per instruction*) es el número promedio de ciclos de reloj de la máquina, usados para cada instrucción y la velocidad del reloj es el número de ciclos por segundo a los que opera la máquina. Por ejemplo, en una máquina de 700 MHz cuyo CPI sea de 6 ciclos:

$$MIPS = \frac{700,000,000}{6,000,000} = 116.6$$

La ventaja de usar esta medida de desempeño es que es fácil de entender, sobre todo por personas no muy avezadas en el área, es intuitiva.

Las desventajas son que:

- Depende del conjunto de instrucciones de la máquina. Por ejemplo en el procesador 6502 (Mostek) no había instrucción de multiplicación, en el 8086 (Intel) si la hay, así que una sola instrucción del 8086 puede equivaler a toda una rutina del 6502.
- No todas las instrucciones tardan lo mismo, así que dependiendo de las instrucciones que utilice un programa la medida en MIPS varía. Entonces ¿cuales instrucciones del conjunto total del procesador se deben usar? ¿las más usuales? ¿todas? ¿las más económicas en tiempo?

Incluso puede ocurrir que una máquina A con mejor desempeño real que otra B resulte con una medida en MIPS menor. Por ejemplo, si A tiene unidad de punto flotante (*floating point unit* o FPU) y tarda 8 ms. en una multiplicación mientras B tiene que emular la multiplicación a través de una rutina de 200 instrucciones tardándose 16 ms. resultará que $MIPS(B) > MIPS(A)$.

También ocurre que no se considera la "calidad" de las operaciones realizadas. 100 sumas de punto flotante tardan menos que 100 divisiones, pero siguen siendo 100 instrucciones.

Aun el tiempo de ejecución está sujeto a diversas interpretaciones. Supongamos que se ejecuta un programa en una computadora, el tiempo de ejecución puede ser considerado como:

- el tiempo que tarda en completarse el programa incluyendo entrada/salida, accesos a memoria, etc.
- el tiempo efectivo en el que el procesador está ejecutando las instrucciones del programa.

En el primer caso estamos hablando del tiempo que le tomó al sistema completo (incluyendo dispositivos de e/s, sistema operativo, etc.) la ejecución del programa. Estamos midiendo el desempeño del sistema.

En el segundo caso estamos midiendo el desempeño de la unidad central de proceso (CPU) únicamente.

1.5.1 Programas para evaluar el desempeño

¿Como haríamos para saber si una lavadora A es mejor que otra B? un usuario de lavadoras experimentado lo notaría porque el poner su *carga normal de ropa* en A termina más rápido que cuando la pone en B.

Pero no todo mundo usa pantalones de mezclilla y playeras de algodón. Posiblemente a otro usuario con un guardarropa diferente le parecerá más rápida B porque acaba antes con su *carga normal de ropa*.

Como que el meollo del asunto consiste en encontrar una buena definición de *carga normal*. Una genérica, suficientemente general y representativa.

Cargas de trabajo (*workloads*).

Para probar el desempeño de una computadora, cuya labor esencial es ejecutar programas, debemos ponerla a ejecutar programas, claro está. Pero ¿cuales? ¿cual es la *carga representativa* que debemos usar?. A los programas usados como carga de trabajo se les denomina *workloads* y a las pruebas de desempeño usando cargas de trabajo se les llama *benchmarks* así que nuestro problema es encontrar un conjunto de cargas de trabajo o *workloads* que nos permitan hacer benchmarks confiables, que realmente reflejen el desempeño general de la máquina.

Hay varias opciones para elegir:

1. Programas reales. Correr el conjunto de programas de *Office* en dos PC's con *Windows* y comparar los tiempos de ejecución de los programas haciendo las mismas tareas.

2. Kernels. Son trozos de programas reales que aislan el desempeño de ciertas características individuales. Por ejemplo, el famoso *Linpack* que prueba el desempeño en tareas relacionadas con álgebra de matrices o *Livermore loops* para medir la eficiencia en la ejecución de ciclos. Los kernels no son útiles por sí mismos, ningún usuario realizaría una tarea útil con un kernel.

3. Benchmarks de juguete. De 10 a 100 lineas de código que produce resultados conocidos. Por ejemplo la tradicional criba de Eratóstenes o Quicksort. Era usual hace algunas décadas reportar el desempeño de un sistema diciendo cuanto tiempo tardaba obteniendo los primeros n números primos o cuanto tardaba ordenando una secuencia de enteros.

4. Bechmarks sintéticos. La filosofía de estos es similar a la de los kernels. No son programas útiles por si mismos. Tratan de simular la

frecuencia promedio de las instrucciones que se ejecutan. Algunos fabricantes de computadoras comenzaron a hacer modificaciones a sus compiladores de tal forma que al compilar un bechmark de este tipo se hicieran optimizaciones no estándares para obtener mejores resultados en el benchmark falsificando los resultados[5]. Los más famosos son *Whetstone* y *Dhrystone*.

Conjuntos de *bechmarks* (*benchmark suites*)

Los benchmarks de SPEC.

Los conjuntos de *benchmarks* son colecciones de programas o fragmentos de ellos, al estilo de los kernels. Los más famosos y usuales son los de SPEC (*Systems Performance Evaluation Corporation*) una organización internacional no lucrativa. En el conjunto de SPEC hay programas como `gcc` y `compress` a los que se les pone a trabajar sobre conjuntos de datos de entrada predefinidos que hace difícil su labor. La intención es que las deficiencias en las pruebas realizadas por un programa sean subsanadas por otro del conjunto. Generalmente los programas de SPEC se clasifican en los que usan aritmética entera y los que usan punto flotante (SPECint y SPECfp, respectivamente).

Los programas contenidos en el conjunto de SPEC están listados en las tablas 1.1 y 1.2 [5].

1.6 Reporte de resultados de prueba

El objetivo del reporte de resultados del *benchmark* es, como en cualquier experimento científico, hacer reproducible la prueba.

¿Que datos se deben poner acerca de las circunstancias en las que se realizó la prueba? *todos*. No es muy útil decir algo como "la prueba tal tomo 18 segundos en una computadora Pentium III" importan cosas como:

- Los datos de entrada de la prueba (a menos que sean estándar como en SPEC).
- Versión del programa de prueba.

[5]De hecho es posible optimizar tirando a la basura el 25% del código de Dhrystone, uno de los benchmarks más populares.

Nombre	Descripción
`164.gzip`	Data compression utility
`175.vpr`	FPGA circuit placement and routing
`176.gcc`	C compiler
`181.mcf`	Minimum cost network flow solver
`186.crafty`	Chess program
`197.parser`	Natural language processing
`252.eon`	Ray tracing
`253.perlbmk`	Perl
`254.gap`	Computational group theory
`255.vortex`	Object Oriented Database
`256.bzip2`	Data compression utility
`300.twolf`	Place and route simulator

Tabla 1.1: Contenido de las pruebas de SPEC en aritmética entera (CINT2000). Todos los programas están escritos en C salvo `eon` que está en C++.

Nombre	Descripción
`168.wupwise`	Quantum chromodynamics
`171.swim`	Shallow water modeling
`172.mgrid`	Multi-grid solver in 3D potential field
`173.applu`	Parabolic/elliptic partial differential equations
`177.mesa`	3D Graphics library
`178.galgel`	Fluid dynamics: analysis of oscillatory instability
`179.art`	Neural network simulation; adaptive resonance theory
`183.equake`	Finite element simulation; earthquake modeling
`187.facerec`	Computer vision: recognizes faces
`188.ammp`	Computational chemistry
`189.lucas`	Number theory: primality testing
`191.fma3d`	Finite element crash simulation
`200.sixtrack`	Particle accelerator model
`301.apsi`	Solves problems regarding temperature, wind, velocity and distribution of pollutants

Tabla 1.2: Contenido de las pruebas de SPEC en aritmética de punto flotante (CFP2000). Diez de los programas están escritos en Fortran y cuatro en C.

- Banderas usadas en el compilador y nivel de optimización al compilar las pruebas.
- Sistema operativo y versión.
- Tamaño de la memoria
- Tamaño de cache
- Número, capacidad y tipo de discos duros
- Versión del CPU[6]
- Velocidad del reloj

Esto hace posible duplicar los resultados con precisión. *No son confiables los resultados no reproducibles.*

Como algunos fabricantes proveen a sus compiladores de banderas específicas para compilar benchmarks conocidos, SPEC especifica con que banderas deben ser compiladas sus pruebas. Algunos fabricantes pueden añadir sus banderas específicas y reportar el desempeño obtenido con ellas, pero están obligados a reportar, junto con esos resultados, los obtenidos con las banderas estándar de SPEC. Al desempeño obtenido en las condiciones estándar se le denomina *baseline performance.* Nuevamente, no es confiable un reporte de benchmark SPEC que no contenga el *baseline performance.*

1.7 Comparaciones y mecanismos de síntesis de desempeño

En general los diseñadores de computadoras no se han puesto de acuerdo ni siquiera en lo fundamental: qué medir, como medirlo, en que ambiente experimental medirlo y una vez hecha la medición, como sintetizar y comparar resultados, que significa "más rápido".

En la página 24 del libro de Hennessy [3], aparecen unos datos mostrados aquí en la tabla 1.3. De estos datos se desprenden las siguientes afirmaciones:

[6]Esto es más importante de lo que parece. Los primeros reportes de SPEC que comparaban el Intel Pentium III y el AMD Athlon no usaron una versión depurada del Athlon y por tanto este exhibia un desempeño inferior al verdadero.

Programa	Computadora A	Computadora B	Computadora C
P_1	1	10	20
P_2	1000	100	20
Total	1001	110	40

Tabla 1.3: Tiempos de ejecución de dos programas diferentes: P_1 y P_2 en tres diferentes computadoras A, B y C.

- A es 10 veces más rápida que B para P_1
- B es 10 veces más rápida que A para P_2
- A es 20 veces más rápida que C para P_1
- C es 50 veces más rápida que A para P_2
- B es 2 veces más rápida que C para P_1
- C es 5 veces más rápida que B para P_2

Cada afirmación es cierta y puede ser usada para beneficio del primer sistema mencionado en la frase. Para cada sistema hay dos frases que lo comparan favorablemente con cada uno de los otros sistemas. Así que no es claro cuál computadora es mejor.

1.7.1 Tiempo total de ejecución

Síntesis del desempeño.

La situación del ejemplo anterior cambia si sumamos los tiempos como aparecen el el último renglón de la tabla 1.3. Haciéndolo podemos decir;

- B es 9.1 veces más rápida que A para P_1 y P_2.
- C es 25 veces más rápida que A para P_1 y P_2.
- C es 2.75 veces más rápida que B para P_1 y P_2.

Si una carga de trabajo consistente en el conjunto P_1, P_2 fuera ejecutada repetidas veces en los tres sistemas las afirmaciones anteriores hacen posible predecir cuál será el tiempo de ejecución relativo en cada uno con esa carga de trabajo. Y eso es justo lo que queremos, *poder predecir*.

Hay otras posibilidades para sintetizar los resultados y que, de alguna manera, están íntimamente relacionadas con el tiempo total de ejecución de una colección de programas. Por supuesto esta colección es el conjunto de programas en el benchmark que esperamos sean lo suficientemente representativos de las tareas que potencialmente se ejecutarán en la máquina.

1. Media aritmética:
$$\overline{T} = \frac{1}{n} \sum_{i=1}^{n} t_i \tag{1.7.13}$$
donde n es el número de programas en el conjunto de benchmark y t_i es el tiempo que requirió la ejecución del i-ésimo programa en la computadora probada.

2. Media armónica:
$$\overline{N} = \frac{n}{\sum_{i=1}^{n} \frac{1}{\text{Tasa}_i}} \tag{1.7.14}$$
donde n es el número de programas en el conjunto de benchmark y Tasa_i es una función del inverso del tiempo de ejecución del i-ésimo programa. Es decir $\text{Tasa}_i = f\left(\frac{1}{t_i}\right)$ donde t_i es el tiempo que requirió la ejecución del i-ésimo programa en la computadora probada.

Nótese que la media aritmética esta relacionada con medición de tiempo de respuesta y la media armónica esta relacionada con el rendimiento (*throughput*).

El desempeño se prueba con una mezcla heterogénea de programas y cada uno toma un tiempo diferente en su ejecución. De aquí puede desprenderse la pregunta siguiente: ¿es justo considerar igualmente importantes a todos los programas en la prueba? puede ser que no, puede ser que algunos de ellos prueban características menos relevantes que otros o que nos interesen más por alguna razón. De ser así podríamos asignarle más peso a los programas que consideramos más representativos o a aquellos que prueben más exhaustivamente aquella característica que nos parece más relevante en un estudio específico. Así que podemos generalizar el concepto de media aritmética como sigue:
$$\overline{T}_W = \sum_{i=1}^{n} w_i \, t_i \tag{1.7.15}$$
donde $w_i \in [0, 1]$ es el peso que deseamos asignarle al i-ésimo programa del conjunto de benchmark. La suma de los pesos debe ser 1.

La decisión de los pesos puede ser, por supuesto, tan arbitraria como se desee, pero tendrá más sentido si se eligen los pesos de una manera razonable. Por ejemplo tomando a una máquina como patrón de referencia. Sea t_i^A el tiempo que le tomó a la máquina A ejecutar el programa i-ésimo de la prueba. Elegimos el peso asociado a ese programa como:

$$w_i^A = \frac{1}{t_i^A \sum_{k=1}^{n} \left(\frac{1}{t_k^A}\right)} \tag{1.7.16}$$

y lo utilizamos con todas las máquinas en la comparación. Evidentemente $\sum_{i=1}^{n} w_i = 1$.

Cómo hacer trampa.

En la tabla 1.5 se muestra el resultado de ponderar los tiempos de ejecución de cada programa tomando como referencia distintas máquinas. Nótese que cuando la máquina A es tomada como referencia, usando los pesos W_A para todas las máquinas, entonces A resulta ser justamente la mejor máquina. Si en cambio se tomen los pesos W_B, que significa que tomamos a B como máquina de referencia, entonces B resulta ser la mejor máquina. De hecho ocurre lo siguiente ([4], cap. 11):

- Si un sistema A es mejor en todos los tiempos de los programas del benchmark, no importa el sistema de referencia, A resultará siempre mejor a los demás. Lo único que se puede hacer es, a lo más, resaltar sus bondades.
- Si un sistema A tiene mejores tiempos que otro B en unos programas y peores en otros, entonces las distintas elecciones de la máquina de referencia pueden dar resultados contradictorios. Para ganar:
  1. si la métrica es LB (*Lower is Better*, como tiempo de ejecución, por ejemplo) elige el sistema que deseas que gane como referencia.
  2. Si la métrica es HB (*Higher is Better*, como tareas por segundo, *throughput*) elige a tu oponente.

Así que no es trivial elegir una base de tal forma que sean objetivos los resultados.

La suma de los tiempos de ejecución y la media aritmética están en proporción directa con el tiempo de ejecución, que es lo que queríamos medir. En ese sentido son idóneas. Pero como todos sabemos un número demasiado grande o demasiado pequeño en relación a los demás "jala" el promedio o

W	$\mathbf{W_A}$	$\mathbf{W_B}$
0.5	0.999	0.909
0.5	0.001	0.091

Tabla 1.4: Pesos obtenidos mediante la expresión 1.7.16 con los datos de la tabla 1.3. Para la columna $\mathbf{W_A}$ se utilizaron los tiempos de la máquina A y para la columna $\mathbf{W_B}$ se utilizaron los de la máquina B. La primera columna establece pesos iguales para ambos programas.

Programa	Computadora A	Computadora B	Computadora C
P_1	1	10	20
P_2	1000	100	20
$\overline{T}_W$	500.5	55.0	20
$\overline{T}_{W_A}$	2	10.09	20
$\overline{T}_{W_B}$	91.91	18.19	20

Tabla 1.5: Tiempos medios pesados para las máquinas y programas de la tabla 1.3 usando los pesos de la tabla 1.4.

la suma hacia algún lado, son medidas muy dependientes de la magnitud relativa de los tiempos de ejecución. Es por eso que se usan los promedios pesados aunque, como hemos visto, pueden dar resultados contradictorios.

1.7.2 Tiempo de ejecución normalizado

Definición 1 Sean $t_1, t_2, \ldots, t_n$ los n tiempos de ejecución de n diferentes programas en una máquina cualquiera A y sean $t_1^R, t_2^R, \ldots, t_n^R$ los tiempos de ejecución de los mismos programas en una máquina de referencia R. Definimos el *tiempo normalizado de ejecución del i-ésimo programa*, $\hat{t}_i$, como:

$$\hat{t}_i = \frac{t_i}{t_i^R}$$

Definición 2 La *media geométrica* de un conjunto T de n tiempos de ejecución normalizados es:

$$G(T) = \sqrt[n]{\prod_{i=1}^{n} \hat{t}_i}$$

La media geométrica se utiliza porque:

- Tiene propiedades estadísticas deseables. Por ejemplo, dadas dos muestras X y Y, resulta que:

$$\frac{G(X)}{G(Y)} = G\left(\frac{X}{Y}\right)$$

- Proporciona resultados consistentes. No importa la elección de la base ni las magnitudes relativas de las mediciones.

De hecho las medidas más confiables de desempeño, las proporcionadas por las pruebas de SPEC son obtenidas mediante la media geométrica utilizando una máquina de referencia, en SPEC92 era la VAX-11/780, en SPEC2000 es una Sun Ultra 5 a 300 MHz. [5].

Pero no todo puede ser bonito. Las desventajas de este método de síntesis son:

- No es una medida intuitiva. De hecho puede mal interpretarse como el factor por el que hay que multiplicar un tiempo de ejecución para obtener otro.
- Se pierde información.
- Estimula a los fabricantes a mejorar el desempeño de los programas que *ya tienen un buen desempeño* relativo a los demás. Proporciona mejores resultados bajar el tiempo de ejecución de un programa de 2 seg. a 1 seg. que bajar el de otro de 10,000 a 7,000 seg. Además los programas pequeños son más fáciles de optimizar por un compilador para que den mejores resultados, aunque no sean representativos.

1.8 Principios cuantitativos de diseño de computadoras

La regla fundamental es *hacer más rápido lo más común.* Si se mejora el desempeño al efectuar una operación que se hace frecuentemente entonces se mejora el desempeño global del sistema de modo importante. Si se mejora

Regla fundamental.

el desempeño en un caso extraño, que casi nunca se usa entonces la mejora no se va a notar, casi no hay impacto en el desempeño global.

De que manera hay que decidir ahora cuál es el caso frecuente y como cuantificar la ganancia de desempeño que se obtiene.

Ley de Amdahl La ley de Amdahl permite cuantificar el impacto que tiene una mejora en un sistema, imaginamos que se ejecuta una tarea dada en un sistema antes y después de implementar una mejora que incrementa el desempeño del sistema. La mejora puede ser cuantificada calculando S, la "aceleración" (*speedup*), como sigue:

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

donde t_{sin} es el tiempo de ejecución de la tarea sin usar la mejora y t_{com} es el tiempo de ejecución de la tarea usando la mejora siempre que fue posible. Nótese que el término "aceleración" no esta siendo usado propiamente en el sentido físico. S carece de unidades mientras que la aceleración en este caso debería ser algo así como $tareas/seg^2$.

El speedup nos indica que tanto más rápido correrá un programa en una máquina con mejoras contra el tiempo que tardaba en la máquina sin las mejoras incorporadas.

Hay dos factores que influyen en el speedup:

1. La fracción del tiempo total de ejecución del programa en la máquina sin mejora en que puede usarse la mejora. Es decir: Sea t_o el tiempo que tarda en ejecutarse el programa X en la máquina original (sin mejora). Dentro de ese tiempo el programa podría usar la mejora durante t_{pu}. La fracción de mejora (*fraction enhanced*) F es:

$$F = \frac{t_{pu}}{t_o} \tag{1.8.18}$$

2. Qué tanto más rápido corre un programa que use todo el tiempo la mejora comparado con el mismo programa en la máquina original. A esto se le denomina "aceleración de mejora" (*speedup enhanced*). Sean t_o el tiempo que tarda en ejecutarse un programa en la máquina original. Este programa podría usar durante toda su ejecución la mejora

y en ese caso tarda t_m unidades de tiempo, así que:

$$s = \frac{t_o}{t_m} \tag{1.8.19}$$

El tiempo de ejecución de un programa en un sistema con la mejora incorporada se ve modificado de la siguiente forma:

$$t_n = t_o \left[(1 - F) + \frac{F}{s} \right] = t_o(1 - F) + t_o F \frac{1}{s} \tag{1.8.20}$$

donde t_n es el tiempo "nuevo" que tarda el programa en la máquina mejorada.

Es posible entonces definir la aceleración general (*speedup overall*):

$$S = \frac{t_o}{t_n} = \frac{1}{(1 - F) + \frac{F}{s}} \tag{1.8.21}$$

1.9 Desempeño de CPU

Las computadoras no operan continuamente, lo hacen por pasos, discretamente. En cada paso efectúan una operación elemental, de hecho más elemental que las de lenguaje de máquina generalmente. El ritmo al que se ejecutan los pasos los marca el equivalente a un metrónomo de los que se utilizan en música; un reloj que emite pulsos con una frecuencia fija. Salvo el pequeño error derivado de la operación intrínseca de nuestros dispositivos electrónicos los pulsos de reloj son cuadrados, así que en todo instante de tiempo el reloj está en uno de dos posibles estados, alto (1) o bajo (0). A cada pulso del reloj se le llama en inglés *tick* o *clock*.

Mientras mas alta sea la frecuencia del reloj de una computadora es mayor la cantidad de tiempo que realmente está operando, así que siempre es útil saber cuál es la frecuencia de operación de una CPU. Es a esta frecuencia a la que se hace alusión cuando se habla de un procesador a 700 MHz. lo que significa que emite 700 millones de pulsos en un segundo, una a 2 GHz (algo frecuente en un futuro cercano) emite 2 mil millones de pulsos de reloj por segundo.

Considerando la frecuencia de operación de una CPU es posible calcular el tiempo efectivo de operación del procesador en la ejecución de un

Prefijo	**Valor**
atto	10^{-18}
femto	10^{-15}
pico	10^{-12}
nano	10^{-9}
micro	10^{-6}
mili	10^{-3}
kilo	10^{3} o 2^{10}
mega	10^{6} o 2^{20}
giga	10^{9} o 2^{30}
tera	10^{12} o 2^{40}

Tabla 1.6: Prefijos comunes y su valor.

programa:

$$CPU_{time} = ciclos\ por\ programa \times duración\ de\ ciclo \tag{1.9.22}$$

Por ejemplo un programa que tarda en ejecutarse 7 millones de ciclos en una computadora cuyo ciclo dura 2 ns. tarda: 0.014 s. de CPU_{time}.

Otra manera de medir el tiempo de CPU es, evidentemente:

$$CPU_{time} = \frac{ciclos\ por\ programa}{frecuencia} \tag{1.9.23}$$

Un programa que tarda 17 millones de ciclos en una máquina a 640 MHz, tarda 0.0265 s. de tiempo de CPU.

Pero es difícil saber cuantos ciclos en total toma la ejecución de un programa, así que podemos contar mejor el número de instrucciones (IC o *instruction count*). Si sabemos el tiempo promedio de ciclos por cada instrucción (CPI o *clocks per instruction*).

$$CPI = \frac{Ciclos\ para\ el\ programa}{IC} \tag{1.9.24}$$

Esta es una medida de que tan meritorio es un procesador, que tan baratas, en ciclos de reloj, son sus instrucciones.

De 1.9.24:

$$ciclos = CPI \times IC$$

en 1.9.22:

$$CPU_{time} = CPI \times IC \times duración\ de\ ciclo \tag{1.9.25}$$

o bien en 1.9.23:

$$CPU_{time} = \frac{CPI \times IC}{frecuencia} \tag{1.9.26}$$

Hagamos un análisis de unidades de la expresión 1.9.25.

$$\frac{ciclos}{instrucción} \times \frac{instrucciones}{programa} \times \frac{segundos}{ciclo} = \frac{segundos}{programa}$$

En síntesis el desempeño depende de:

Los tres factores de desempeño.

1. Frecuencia del reloj [7]
2. Cantidad de ciclos por cada instrucción.
3. Contador de instrucciones por programa.

y depende en la misma medida de cada uno de los tres elementos, así que una mejora del 10% en alguno de los rubros mencionados genera una mejora del 10% en el desempeño total.

¿De qué dependen estos factores?

Frecuencia. Hardware, tecnología y organización del procesador (calor generado, densidad de componentes).

CPI. Organización, arquitectura del conjunto de instrucciones.

Instrucciones por programa. Arquitectura del conjunto de instrucciones y eficiencia del compilador.

[7] De hecho, basándose en el hecho de que frecuentemente los resultados de benchmarks son falseados, algunas personas de prestigio piensan que debería usarse la frecuencia de reloj como una medida importante de desempeño. Véase por ejemplo [2].

Algunas veces es útil contar el número de ciclos de reloj como sigue:

$$Ciclos = \sum_{i=1}^{n} CPI_i \times IC_i$$

i es el índice de la instrucción en el catálogo de instrucciones de la máquina, IC_i es el número de veces que la instrucción i se ejecuta en el programa y CPI_i es el número de ciclos de reloj que tarda en ejecutarse la instrucción i. De donde:

$$CPU_{time} = \left(\sum_{i=1}^{n} CPI_i \times IC_i \right) \times \textit{duración de ciclo} \tag{1.9.27}$$

Esto porque, de acuerdo a la definición de valor esperado de una variable aleatoria (en este caso en número de clocks por instrucción): $E(X) = \sum_{\textit{espacio muestral}} xp(x)$. Como nuestro valor de CPI es, de hecho, un promedio:

$$CPI = \sum_{i=1}^{n} CPI_i \times \frac{IC_i}{IC}$$

Si sustituimos esto en 1.9.25 nos da 1.9.27.

CAPÍTULO 2

El conjunto de instrucciones

Ya hemos dicho que hasta hace algún tiempo la arquitectura de computadoras estaba avocada exclusivamente a diseñar el conjunto de instrucciones de una máquina y la manera en que estas instrucciones se implementaban. Actualmente el área es mucho más amplia pero el diseño del conjunto de instrucciones sigue siendo parte fundamental de ella. Nos dedicaremos ahora a estudiar el diseño del conjunto de instrucciones, las diversas alternativas que existen para ello y las implicaciones que trae consigo cada diseño particular.

2.1 Clasificación de la arquitectura del conjunto de instrucciones.

La primera distinción que es posible hacer entre diferentes conjuntos de instrucciones esta dictada por el modo en que los operandos de las instruc-

ciones son almacenados y accedidos por el CPU.

Operandos implícitos y explícitos.

En primera instancia los operandos pueden ser nombrados *explícitamente* o *implícitamente*. En la primera alternativa se le dice al CPU donde se encuentran almacenados los datos de la operación a realizar, en la segunda opción el CPU ya sabe de antemano donde se encuentran.

Arquitectura de Von Neumann.

Nuestras computadoras, en general, requieren de almacenar tanto los datos como el programa en memoria para poder operar. Esto es en esencia lo que significa que nuestras computadoras tengan una *arquitectura de Von Neumann*[1]. La memoria de laa computadora, como veremos más adelante se organiza en una jerarquía de acuerdo a su velocidad de acceso su tamaño y su costo. Por ahora sólo la dividiremos en dos: la memoria principal del sistema (RAM) y una muy pequeña cantidad de memoria de acceso casi inmediato que se encuentra, de hecho, dentro del CPU y que se divide en pequeños bloques de unas cuantas decenas de bits llamados *registros*.

Si los operandos son nombrados implícitamente (lo que, de hecho, significa que no son nombrados) estos se encuentran almacenados en una pila (*stack*) o en un registro especial del CPU llamado *acumulador*. Cada operación entonces sólo tiene que tener enfrente a lo más el nombre de un operando: cuando hay que traer algo de la memoria principal para almacenarlo en la pila o el acumulador. Una instrucción aritmética no requeriría de ningún operando en el caso de una pila, por ejemplo, ya que ambos operandos son los dos últimos datos almacenados en esta y el resultado se coloca como el nuevo tope.

Es posible también pensar en distintos grados de hacer explícitos los operandos, es decir, considerar cuantos operandos es necesario nombrar en cada instrucción. Así podemos tener los operandos completamente implícitos, como en el caso de un CPU de pila, o bien podemos tener que nombrar sólo uno de ellos, como en el caso de un CPU con acumulador. En este caso una operación aritmética debe tomar uno de sus operandos de la memoria y tanto el otro dato como el resultado se almacenan en el acumulador.

Si, en cambio, los operandos son nombrados explícitamente entonces estos pueden estar almacenados en la memoria principal del sistema o bien

[1]John Von Neumann fue consultor en el proyecto de ENIAC (Eckert y Mauchly). La programación de ENIAC se hacía cambiando el cableado, cada programa se alambraba. En 1945, luego de ENIAC, Von Neumann escribió "First Draft on a Report on the EDVAC" donde se propone que el programa ejecutado por la computadora se almacene en la memoria de esta.

Stack	Acumulador	Registro-Mem	Mem-Mem	load-store
push A	lda A	mov R1,A	mov C,A	load R1,A
push B	add B	add R1,B	add C,B	load R2,B
add	sta C	mov C,R1		add R3,R1,R2
pop C				store C,R3

Tabla 2.1: ejemplos de realización de una suma en diferentes arquitecturas de conjunto de instrucciones.

en los registros del procesador o bien en ambas partes. Es posible clasificar ahora la arquitectura del conjunto de instrucciones de acuerdo al número de operandos que una instrucción puede acceder de la memoria principal.

En principio se podría pensar en que todos los operandos de la instrucción estén almacenados en memoria principal. Nuevamente tenemos la opción de que haya tres operandos (dos operandos propiamente dichos y un tercero para guardar el resultado) o bien dos (el resultado se almacena en uno de ellos luego de hacer la operación).

Arquitectura *load-store*.

Es posible también que uno o dos de los operandos sean registros del procesador y el resto celdas de memoria o bien que forzosamente uno de los operandos tenga que ser un registro y el otro un registro o celda de memoria. Esto significa que para poder hacer una operación es necesario primero almacenar al menos uno de los operandos en los registros del procesador. A estas arquitecturas se les denomina de carga-almacenamiento (*load-strore*).

En la tabla 2.1 se muestran ejemplos de como se vería la realización de una instrucción suma en máquinas con diferentes tipos de arquitectura de conjunto de instrucciones.

Esquemáticamente tenemos la siguiente taxonomía de las arquitecturas del conjunto de instrucciones:

1 Operandos explícitos (completamente). Pueden ser registro-memoria, memoria-memoria o *load-store*.

  1.1 Registros de propósito general.

  2.2 Extendida de acumulador.

2 Con operandos implícitos. Tienen sólo un operando en las instrucciones.

  2.1 De *stack*.

2.2 De acumulador.

Casi todas las arquitecturas nuevas (después de 1980) son de registros de propósito general (GPR) y de carga y guarda (*load-store*), por ejemplo MIPS o Sparc. Las máquinas de *stack* han desaparecido, ejemplos de ellas eran las Burroughs B5000 o B6500 y los coprocesadores de punto flotante de Intel. Las arquitecturas de acumulador fueron muy populares al comienzo de la era de las computadoras personales, por ejemplo Mostek 6502, el procesador de 8 bits usado por las Apple II y Commodore 16 y 64. Las unicas arquitecturas conocidas de tres operandos, en las que incluso los tres podían ser direcciones de memoria fueron algunas VAX. Arquitectura de acumulador extendida es, por ejemplo, la de Intel 80X86 donde hay registros que, en principio sirven para todo o casi todo, pero están mejor provistos para hacer ciertas operaciones e incluso hay operaciones que requieren de ciertos registros para hacerse (por ejemplo el contador de ciclos siempre debía estar en ECX en estos procesadores de Intel), a lo más un operando podía ser una dirección de memoria.

¿Por qué usar registros?

Cabe ahora preguntarse: si los datos están a fin de cuentas en la memoria principal, ¿para qué traerlos al CPU y guardarlos en registros?, tanto los datos como los resultados deben terminar en RAM, ¿por qué no los dejamos ahí? ¿por qué se siguen diseñando máquinas con registros de propósito general que de hecho han desplazado a las de *stack*, acumulador o aquellas que pueden acceder a memoria en cualquier instrucción? ¿por qué son cada día más las arquitecturas de *load-store*?

Las principales razones son:

1. Es más rápido acceder a un registro interno del procesador que a una celda de memoria que está fuera del CPU y cuyo tiempo de acceso es mayor y no depende del CPU.
2. Se reduce la densidad de las instrucciones: ¿Cuantos registros se les antoja que tenga un procesador? decenas, cientos, ¿cuantas celdas de memoria se les antoja que tenga un sistema (celdas de 1 byte)? decenas de millones, cientos de millones. ¿cuantos bits necesitamos para decir un número (el nombre de) un registro? 5, 8 a lo más. y ¿para decir una dirección de memoria? 32, 38, quizás más.
3. Los escritores de compiladores prefieren registros versátiles, que sirvan para todo en vez de registros que sirvan mejor para algunas cosas o de propósito específico.

Tipo	Ventajas	Desventajas
(3,0)	Las instrucciones casi tienen tamaño fijo (la mayoría sólo especifica la operación y tres registros), casi todas tienen el mismo CPI	El código generado es largo (hay que hacer cargas por cada operando)
(2,1)	Los datos se pueden acceder sin carga previa. Instrucciones fáciles de decodificar	Generalmente el número de registros es insuficiente. CPI varía considerablemente. Un operando es destruido siempre.
(3,3)	Código compacto. Los resultados temporales no ocupan espacio vital (como registros).	Gran variabilidad de tamaños y formatos de instrucción, difícil decodificación. La memoria se convierte en un cuello de botella, muchos accesos simultáneos.

Tabla 2.2: Comparación de tres diferentes alternativas de diseño de conjunto de instrucciones. La notación (a, b) significa a operandos, b de ellos pueden ser direcciones de memoria.

4. Es más fácil evaluar expresiones aritméticas que en una máquina con *stack*. Piénsese por ejemplo en evaluar una expresión como (AB)-(CD) en una máquina de *stack*.

En la tabla 2.2 se muestran algunas ventajas y desventajas de tres alternativas diferentes de arquitecturas.

En el resto del texto adoptaremos las siguientes convenciones de tamaño de operandos:

byte 8 bits

halfword 16 bits

word 32 bits

doubleword 64 bits

Si se utilizan operandos de más de un byte, que en general es la unidad mínima de almacenamiento en memoria (cada celda de memoria es un byte, cada byte tiene su propia dirección), entonces otra de las cosas que hay que decidir es como serán acomodados los bytes del operando en memoria. Generalmente a los bytes (de hecho también a los bits) de un número se les dice más o menos significativos en función de los pesos que tengan asociados en el sistema numérico posicional binario. Así que hay esencialmente dos maneras diferentes de acomodar los bytes, a saber: colocando en la dirección de memoria más baja el byte menos significativo del dato (*little endian*) y la complementaria, que en la dirección de memoria más baja coloca el byte más significativo del dato (*big endian*[2]).

little-endian y *big-endian.*

Es decir en una máquina *little endian* si una instrucción dice: `storehw R1, 0X03FD` que indica que se almacenará el contenido de la primera media palabra del registro R1 en la dirección 03FD en hexadecimal, esto significa que el byte menos significativo de R1 se guarda en 0X3FD y el siguiente byte más significativo en 0X3FE. Si en cambio la máquina fuera *big endian*, entonces el byte menos significativo se almacenaría en 0X3FE y el siguiente más significativo en 0X3FD.

Ejemplo: Supóngase que se pretende guardar un 300_{10} en la dirección 0002. $300_{10} = 012C_{16}$ esto son dos bytes, el más significativo es 01 el menos significativo es $2C$, así que en una máquina little endian se almacenaría en la dirección 0002 el $2C$ y en la dirección 0003 el 01.

Aparentemente nadie nota el orden de las cosas y a nadie le importa. En realidad esto se transforma en un dolor de cabeza cuando se pretende intercambiar información entre diferentes máquinas con diferente convención.

Alinemiento.

También en el caso de operandos mayores de un byte es necesario decidir si poner restricciones de alineamiento o no. Las restricciones de alineamiento son aquellas que especifican que un operando de tamaño k bytes debe empezar en una dirección múltiplo de k. Es decir un objeto de tamaño k está alineado en la dirección A si y sólo si

$$A \pmod{k} \equiv 0$$

Así por ejemplo una palabra (4 bytes) está alineada si comienza en una

[2]Los términos *little endian* y *big endian* provienen de los viajes de Guliver de Swift, corresponden a una clasificación de acuerdo a el lado del huevo que se rompe para comerlo, el ancho o el angosto.

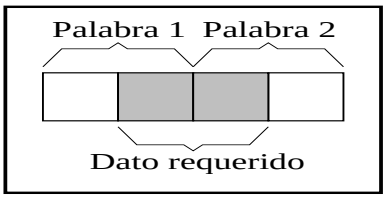

Figura 2.1: Esquema de un acceso no alineado. Se requiere una palabra no alineada que posee sus dos mitades en palabras distintas de una memoria alineada en palabras, el primer acceso hace que se deba desechar la mitad inferior de la palabra accedida, el segundo acceso hace que se deba desechar la mitad más significativa de la segunda palabra accedida.

dirección múltiplo de 4, es decir: $0, 4, 8, 12, \ldots$ o equivalentemente todas las direcciones que tienen ceros en los dos bits menos significativos.

En el ensamblador del 80X86 se utilizaban las palabras reservadas `BYTE`, `WORD`, `PARA` o `PAGE` para especificar que lo que seguía debía comenzar en una dirección múltiplo de: uno, 2, 16 y 128 respectivamente.

¿Por qué establecer estas restricciones? básicamente por dos razones: los datos no alineados hacen más lentas las operaciones y porque proveer a las máquinas de lo necesario para que no sea relevante la alineación es una complicación de hardware.

Las memorias vienen alineadas típicamente en palabras o dobles palabras así que un acceso no alineado ocasiona múltiples accesos alineados, que son los naturales.

Aun si los datos están alineados no es trivial el acceso a datos de longitud menor al tamaño de palabra de memoria. Si una máquina soporta acceso a bytes o medias palabras requiere de hardware extra. Por ejemplo supóngase que se desea acceder a un byte en la memoria que contiene un $FF = -1$ y que se desea almacenar en un registro del procesador cuya longitud es de una palabra para hacer una operación aritmética. Necesitamos que los ocho bits de memoria se copien con todo y signo al registro de 32 bits que debe resultar con $FFFFFFFF$.

2.2 Modos de direccionamiento

Cuando ponemos una operación con sus operandos ¿como especificamos los operandos? ¿como decimos donde exactamente está el operando? A esto es a lo que se refieren los modos de direccionamiento justamente.

Ventaja de usar modos de direccionamiento.

La ventaja de utilizar modos de direccionamiento es que en una sola instrucción es posible resumir varios cálculos para determinar la dirección efectiva de un operando. Por ejemplo, si un operando de una suma es el contenido del registro `R5` de un procesador de arquitectura *load-store* y el otro está en la dirección de memoria que resulta de sumarle 100 al contenido de la dirección de memoria guardadaen R1 (véase figura 2.2), entonces para hacer la suma tendríamos que hacer algo como esto:

```
load R2, [R1] ; carga en R2 el contenido de la dirección de
              ; memoria apuntada por R1
add  R2, 100  ; suma R2 con 100
load R4, [R2] ; carga en R4 el operando
add  R5, R4   ; realiza la suma finalmente
```

Pero en alguna otra máquina podría existir una instrucción para acceder al operando que está especificado indirectamente y podría verse tal como se muestra en la figura 2.2.

Así que, en general el usar modos de direccionamiento diferentes reduce el contador de instrucciones (IC) por programa, lo que, como vimos en el capítulo anterior, tiene un impacto favorable en el desempeño.

Pero todo cuesta, el tener modos de direccionamiento diferentes aumenta la complejidad de las instrucciones (a fin de cuentas sí hay que hacer lo que hicimos a pie en el código anterior) y aumenta la dificultad de decodificación. Nuevamente tenemos, como ya es costumbre, un compromiso, *ni tanto que queme al santo, ni tan poco que no alumbre.*

Para decidir que modos de direccionamiento incluir en una arquitectura nos ayuda saber cuáles modos son lo más usuales. Así sólo incluimos aquellos que sean de indispensables a muy usados y desechar los que raramente se usan.

Modos de direccionamiento.

Los diferentes modos de direccionamiento con sus frecuencias aproximadas son:

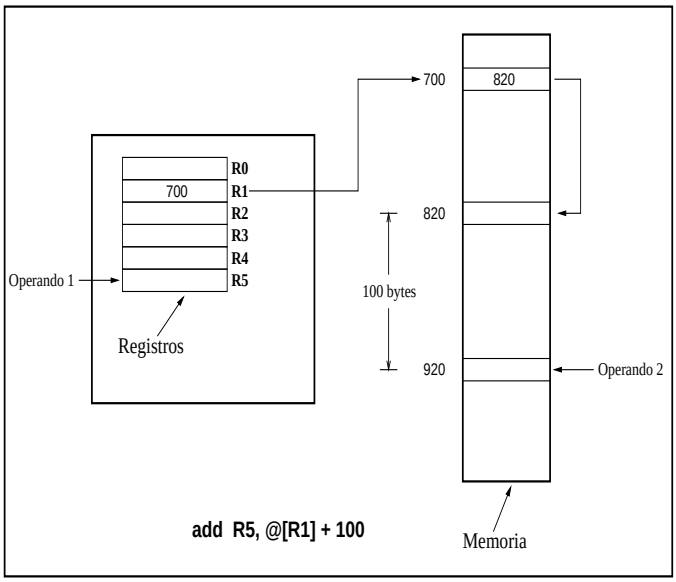

Figura 2.2: Direccionamiento de memoria indirecto con desplazamiento.

desplazamiento La dirección del operando se obtiene sumando el contenido de algún registro con una constante. Se usa el 40% de las veces.

inmediato El operando aparece en la instrucción misma. Se usa un 39% de las veces.

indirecto La dirección del operando se encuentra almacenada en un registro. Se usa el 11% de las veces.

escalado El operando se encuentra en la dirección de memoria obtenida sumando una constante, un registro y un múltiplo de otro registro. Se usa el 6% de las veces.

indirecto de memoria Parecido al del ejemplo: el operando esta almacenado en una celda de memoria cuya dirección esta contenida en otra dirección de memoria indicada por un registro. Se usa el 1% de las veces.

Estas medidas y modos son tomados de la arquitectura MIPS. en otros sistemas no existen algunos de los modos mencionados y puede haber algunos que no se mencionan. En un 80X86 (16 bits), por ejemplo:

```
MOV AX, 0F84CH  ; inmediato

MOV AX, BX      ; de registro

MOV AX, Var1    ; si Var1 es la etiqueta de la dirección X,
                ; y en X hay 0A y en X+1 hay FF entonces
                ; AH=FF, AL=0A

MOV BX, Offset Var1 ; indirecto de registro
MOV AX, [BX]        ; sólo se puede con SI, DI, BX o BP

MOV AL, [BX]+4      ; en terminología Intel indexado, pero
                    ; comunmente llamado de desplazamiento

MOV AL, Arreglo[SI] ; indexado, sólo se pueden usar SI o DI
                    ; como índices

MOV AX, Elem[BX][DI]; Base indexado. Dirección:
                    ; [Elem]+[BX]+[DI] (parece escalado con
                    ; factor 1)
```

Por supuesto todos los modos presentes en la tabla 2.3 son para direccionar datos. Direccionar código (transferencias de control) es algo que se verá más adelante.

Modos de direccionamiento más usuales.

Al parecer el modo de direccionamiento más usual es el de desplazamiento, que toma el contenido de un registro y lo suma con una constante para obtener la dirección efectiva de un operando. Ahora bien ¿de que tamaño debe ser el rango de la constante de desplazamiento? a fin de cuentas esta constante debe aparecer en la instrucción así que es recomendable saber de que tamaño debe ser el campo de la instrucción asociado al desplazamiento. En el libro de Hennessy ([3], pag. 77, fig. 2.7) se muestra la distribución (frecuencia) de uso de diferentes magnitudes de desplazamiento (de hecho el tamaño en bits del campo de desplazamiento) que utilizan los programas de SPEC, tanto los de aritmética entera como los de punto flotante. En general se observa que están muy dispersos. Pero también se puede ver que usando 12 bits se cubre el 75% de los casos y usando 16 bits el 99%. Así que podemos pensar en asignar una longitud máxima para el campo de desplazamiento de 16 bits y dejar que en el 1% de los casos no cubiertos se resuelva con dos indirecciones en vez de una.

Modo	Ejemplo	Significado	Comentarios
Registro	`add R4, R3`	regs[R4] += regs[R3]	Operando en un registro
Inmediato	`add R4, #3`	regs[R4] += 3	Operando constante
Desplazamiento	`add R4, 100(R1)`	regs[R4] += mem[100+regs[R1]]	Acceso a variables locales
Indirecto	`add R4, (R1)`	regs[R4] += mem[regs[R1]]	El contenido de R1 es la dirección de la celda donde está el operando
Indexado	`add R4, (R1+R2)`	regs[R4] += mem[regs[R1]+regs[R2]]	Arreglos: R1=inicio, R2 = índice (tamaño=1 byte)
Directo	`add R4, (1001)`	regs[R4] += mem[1001]	Datos estáticos
Indirecto de memoria	`add R4, @(R3)`	regs[R4] += mem[mem[regs[R3]]]	R3 contiene la dirección de un apuntador
Autoincremento	`add R4 (R2)+`	regs[R4] += mem[regs[R2]]; regs[R2] += d	Arreglos: cada elemento mide d bytes
Autodecremento	`add R4, -(R2)`	regs[R2] -= d; regs[R4] += mem[regs[R2]]	
Escalado	`add R4, 100(R2)[R3]`	regs[R4] += mem[100+regs[R2]+regs[R3]*d]	Arreglos bidimensionales

Tabla 2.3: Modos de direccionamiento.

El siguiente modo más usado es el inmediato, en el que el operando aparece en la instrucción. ¿Que tipo de instrucciones usan este modo de direccionamiento? se utiliza generalmente en comparaciones, en operaciones aritméticas con constantes y en operaciones de carga (para establecer con una constante el contenido de un registro). Cuando la constante es usada para operaciones en general es pequeña, cuando será utilizada para especificar direcciones de memoria en el futuro, tiende a ser grande. En el libro ya mencionado ([3], pag. 78, fig. 2.8) se observa que en general las instrucciones de aritmética entera usan este direccionamiento un 35% del tiempo mientras que las de punto flotante los usan un 10%. Las instrucciones que más lo usan son las comparaciones (87% aritmética entera, 77% punto flotante) y las operaciones de ALU (58% y 78% aritmética entera y punto flotante respectivamente).

Debemos preguntarnos ahora ¿de que tamaño (en bits) debe ser el campo de la instrucción asociado al operando? En el libro (pag. 79, fig. 2.9) se observa la distribución para esta "variable aleatoria" en algunos programas de la *suite* de SPEC. Se puede apreciar que, nuevamente, estan dispersos los datos pero tienden a estar, en su mayoría, antes de 24 bits. De hecho, de un 50% a un 70% de los operandos inmediantos caben en 8 bits y de un 75% a un 80% caben en 16 bits. Así que sería razonable poner 16 bits en el campo disponible para operandos inmediatos de la instrucción.

Conjuntos de instrucciones ortogonales.

Ya hemos puesto algunos ejemplos de modos de direccionamiento en el 80X86 de Intel. Su caso es curioso, observando superficielmente la documentación técnica de los procesadores parece que tienen modo de direccionamiento inmediato, pero en realidad esto no es así. Formalmente hablando, si observamos los códigos de operación (*opcodes*) asociados a las instrucciones en mnemónicos nos damos cuenta de que, en realidad, hay 9 diferentes códigos de operación de `ADD`, por ejemplo. Cinco de ellos son para operandos inmediatos y varian de acuerdo a la longitud del operando (8, 16, 32 bits en el 80386) y del operando destino (memoria o registros diferentes). Es decir, no hay en las instrucciones un campo específico distinto del asociado al código de operación, en el que se pueda especificar con unos cuantos bits, cuál es el modo de direccionamiento que se utiliza. Para hacerlo se debe cambiar el código de la instrucción, lo que en esencia significa que hay *instruciones diferentes* para diferentes modos de direccionamiento. A un juego de instrucciones en el que el código de operación es independiente del modo de direccionamiento se le denomina *ortogonal*. El 80X86 de Intel tiene, pues, un conjunto de instrucciones *no-ortogonal*. Los conjuntos ortogonales son

buenos para los escritores de compiladores: en cuanto se ve una suma en el código fuente se sabe que *opcode* poner no hay que meterse en un *switch* gigantesco analizando los operandos para determinarlo.

2.3 Instrucciones de contro de flujo

Estadísticamente las frecuencias de los distintos tipos de instrucciones de control de flujo son las siguientes [3]:

- Saltos condicionales de 81 a 86% de las veces.
- Llamadas a subrutinas de un 11 a 13%.
- Saltos incondicionales de un 4 a 6%.

Además resulta más económico poner, en una instrucción de salto un desplazamiento relativo a la instrucción actual (mejor dicho la que sería la siguiente en caso de no saltar) que poner la dirección absoluta de memoria de la instrucción a la que se salta. Porque en general las distancias de salto son expresables en pocos bits ([3], fig. 2.13, pag. 83). Aproximadamente un 50% de los saltos son a distancias menores de las expresables en 6 bits, con 16 basta y sobra para expresar casi cualquier salto.

Tamaño de salto.

Del mismo modo, como se puede ver en la figura 2.15 ([3], pag. 84). En aritmética entera el 86% de los saltos son usando la condición igual o no igual, el restante 14% se divide en partes iguales entre dos cateegorías: (a) menor que, mayor o igual que y (b) mayor que, menor o igual que. En aritmética de punto flotante (a) tiene el 40% del uso, el 23% para (b) y el 37% para igual o no igual.

2.4 Tipo y tamaño de los operandos

Ha llegado el momento de hacer explícito algo que hemos estado suponiendo desde hace un rato. Que las instrucciones de lenguaje de máquina son cadenas de ceros y unos con cierta estructura. Una instrucción está entonces dividida en campos de tamaño diferente y cada campo sirve para especificar ciertas características de la operación que se desea ejecutar. Ya

nos hemos referido al código de operación, este es un número que identifica a cada instrucción del repertorio del procesador. Bien pudiera ser que $A8_{16}$ fuera "suma" (`add`) y $A9_{16}$ fuera "resta" (`sub`), por ejemplo. El resto de los campos sirven para identificar a los operandos y en general es posible que existen campos para especificar el tipo y modo de direccionamiento usado con cada operando.

Incluir campos en la instrucción para especificar el tipo, tamaño y direccionamiento de los operandos complica el formato general del repertorio de instrucciones porque entonces habría diferentes tamaños de los campos dependiendo de las características de los operandos. Así que, en general se procura, para facilitar el proceso de decodificación de instrucciones, incluir el tamaño y tipo de operandos en el código de instrucción[3]. Es decir tendríamos algo así como una suma para palabras, una suma para medias palabras, una suma para números de punto flotante, etc.

El parecido de esto con lo que hemos dicho acerca del Intel 80X86 pudiera hacernos pensar en que la práctica mencionada en el párrafo anterior genera conjuntos de instrucciones no ortogonales, pero no es así. Nótese que estamos diciendo que *el código de instrucción depende de el tipo y tamaño de los operandos* no del *modo en que estos son accedidos.* La ortogonalidad es una propiedad que tiene que ver entre el conjunto de códigos de operación y el conjunto de modos de direccionamiento. Cada instrucción debe tener una componente (viéndola como vector) en el "eje" de los códigos de operación y otra en el de modos de direccionamiento para que podamos decir que ambos conjuntos son ortogonales. Si, en cambio una instrucción es una mezcla en donde el código de operación y el modo de direccionamiento son indistinguibles, entonces ambos conjuntos no son ortogonales.

Los tipos y tamaños de datos que se consideran usualmente en el diseño de arquitecturas son:

caracter 1 byte (ASCII)

halfword 2 bytes

word 4 bytes

double word 8 bytes

[3]En máquinas viejas si solía ponerse un campo en la instrucción que especificaba el tipo y tamaño de los operandos.

enteros 4 bytes, complemento a 2

punto flotante con precisión

simple 32 bits

doble 64 bits

Antes de 1980 cada arquitectura tenía su propio formato y longitud de números de punto flotante, pero ahora todos usan el estándar del IEEE.

Algunas arquitecturas, como Intel 80X86, incluyen además soporte específico para cadenas (mover y comparar generalmente). Pero en realidad no pueden hacer otra cosa que tratar cada byte de la cadena individualmente. Se reduce el número de instrucciones para hacer una operación complicada, pero la complejidad de las instrucciones crece. Algunas otras instrucciones de Intel 80X86 incluyen soporte para BCD, en las que cada 4 bits constituyen un dígito decimal.

Hay que recordar nuestra premisa: *hacer más rápido lo más frecuente.* Con base en ello debemos decidir que tipos de datos deben ser usados y cuáles deben accederse más eficientemente.

Supongamos la siguiente situación: tenemos dos diferentes opciones de chips de memoria uno A que entrega bloques de 32 bits con retardo promedio de 1 ns y otro B que entrega bloques de 64 bits con retardo promedio de 1.3 ns. ¿Cuál usamos? para responder algo como esto hay que responder primero: ¿que tan usuales son los datos de longitud de 64 bits? si son muy usuales entonces vale la pena poner B y un bus de 64 bits porque de un jalón, en un solo acceso se puede traer un operando de esos que son los más usuales (1.3 ns) y con A tendríamos que hacer dos accesos (2 ns). Pero si son mas usuales los operandos de 32 bits, entonces sería mejor poner A y un bus de 32 bits para que de un jalón se traigan 32 bits (1 ns), en vez de poner B, lo que significaría que la mayor parte del tiempo estaríamos trayendo bloques de 64 bits, ya que no se puede traer menos (1.3 ns), de los que tiraríamos a la basura la mitad.

Tamaño de operandos.

Las frecuencias de uso son aproximadamente las siguientes (MIPS, SPEC92):

1 Aritmética entera

1.1 byte, 7%

2.2 halfword, 19%

3.3 word, 74%

2 Aritmética de punto flotante

2.1 word, 31%

2.2 double word, 69%

Nótese por ejemplo que los operandos de un byte de longitud son raros. Es por eso que, por ejemplo, la Alpha no posee instrucciones de acceso a bytes, para hacerlo se requiere hacer varias operaciones, es lento, porque es inusual.

2.5 Codificando el conjunto de instrucciones

Para diseñar el conjunto de instrucciones de la máquina tenemos que decidir qué instrucciónes debe tener, qué modos de direccionamiento, que tipos y tamaño de operandos, cuál debe ser el formato de las instrucciones (qué campos debe tener y de que longitud) y en relación directa con esto último debemos decidir también como se va a codificar la instrucción misma (código de operación u *opcode*)), cómo codificar las direcciones en la instrucción y que grado de independencia le damos al opcode respecto al modo de direccionamiento, es decir, que tan ortogonal queremos que sea el conjunto de instrucciones.

Ortogonalidad Vs. no-ortogonalidad.

Lo deseable es que cuando es tomada una nueva instrucción, el CPU pueda determinar lo más rápido posible que operación se efectuará y sus operandos, algo que veremos en la parte de pipeline es que mientras más temprano, en el ciclo de ejecución de instrucción, tengamos la mayor cantidad de información posible acerca de lo que vamos a hacer, mejor. En máquinas con 5 operandos (más que operandos cosas como indirecciones, constantes de desplazamiento, factores de direccionamiento escalado para cada operando) y 10 modos de direccionamiento, poner el opcode mezclado con el modo de direccionamiento es criminal. Para decodificar eso se va a necesitar mucho trabajo. En cambio en una máquina load-store con dos modos de direccionamiento es posible pensar en que el conjunto de instrucciones no sea ortogonal. En general cuando el producto:

$$\textit{número de operandos en instrucción} \times \textit{modos de direccionamiento}$$

es pequeño, el modo de direccionamiento puede ir mezclado con el opcode. Si en cambio este producto es grande es mejor hacer que el conjunto de instrucciones sea ortogonal respecto a los modos de direccionamiento.

También quisiéramos tener instrucciones breves. Los factores que influyen en la longitud de una instrucción son el número de registros (8 registros se dicen con 3 bits pero 32 registros se dicen con 5) porque en las instrucciones hay que poner el índice del registro que se accederá como operando y el número de modos de direccionamiento.

Número de registros.

Sin embargo, intuitivamente es bueno tener en un CPU muchos registros para tener acceso expedito a varios datos y es bueno tener muchos modos de direccionamiento para ahorrarnos trabajo al acceder a datos muy indirectos.

Otra vez tenemos un compromiso entre dos cosas que hay que balancear:

- El deseo natural de tener tantos registros y modos de direccionamiento como sea posible.
- El impacto que esto tiene en la longitud de los campos de registro y modo de direccionamiento o de opcode y por tanto en el tamaño promedio de instrucción y de programa (recordemos que es bueno tener un bajo IC).
- Tener instrucciones de tamaño "bonito" (múltiplo de un byte).

Es muy importante también tener instrucciones fácilmente decodificables y esto es más fácil mientras más sencillas sean, esto se discutirá más delante.

Codificación de las instrucciones.

Hay tres maneras comunes de codificar el conjunto de instrucciones. Dados los modos de direccionamiento de una máquina es posible elegir alguno de los siguientes esquemas:

- El más flexible. Tener un formato variable que cambie de acuerdo al modo de direccionamiento. (VAX).
- El fijo. En el opcode se codifica también el modo de direccionamiento (el extremo de la no-ortogonalidad). (Sparc, MIPS Power PC).
- Híbrido. Algunos modos (lo más elementales) están codificados junto con las operaciones en el opcode y otros se deben especificar en un campo aparte. (IBM 370, Intel 80X86).

2.6 El papel de los compiladores

Hoy en día casi todo se hace en lenguaje de alto nivel. Antes las decisiones arquitectónicas se tomaban con base en las necesidades de un programador en ensamblador. Ahora ya no es así, por lo que comprender como funcionan los compiladores se ha vuelto crítico para diseñar e implementar un conjunto de instrucciones.

De hecho antes se procuraba aislar el compilador y sus efectos al generar código de la arquitectura. Es decir se procuraba encontrar mecanismos de evaluar el desempeño de una arquitectura independientes de el comportamiento de los compiladores y el código generado por ellos (por ejemplo los MIPS buscan abstraer el desempeño independientemente del código que se ejecute).

Lo cierto es que las decisiones arquitectónicas afectan la calidad del código generado y es este a fin de cuentas el que los usuarios quieren ejecutar. El diseño de la arquitectura puede facilitar o dificultar la labor del compilador.

El primer objetivo de todo programador de compiladores es la correctez: que todo programa válido sea compilado correctamente. El segundo objetivo es que el código generado sea óptimo, veloz, corto.

En la tabla 2.4 se muestran las diferentes etapas de funcionamiento de un compilador, sus funciones y los factores que influyen en su desempeño. Siendo más claros:

- Las optimizaciones de alto nivel se hacen sobre el código fuente y la salida alimenta la etapa posterior, por ejemplo una llamada a una función puede reemplazarse por el código de la función misma si es pequeña, esto ahorra el tiempo de crear el registro de activación.
- Las optimizaciones locales tienen lugar sobre segmentos lineales de código (bloque básico), por ejemplo reemplazar dos instancias del mismo cálculo por una única copia, reemplazar el uso de variables con valor constante por la constante misma, reacomodar expresiones para reducir el número de recursos de almacenamiento necesarios para calcularlas (reacomodar el árbol de la expresión).
- Las optimizaciones globales se encargan de optimizar saltos, transferencias de control y ciclos, por ejemplo buscar expresiones comunes

#	Etapa	Función	Dependencias
1	Front-end (por lenguaje)	Transformar código de un lenguaje a una forma intermedia común.	**D:** Lenguaje. **I:** Máquina.
2	Optimización de alto nivel	Transformación de loops y procedimientos inline	**D:** Lenguaje. **I:** Máquina.
3	Optimizador global	Optimizaciones globales y locales, register allocation	**D:** Algo de la máquina. Poco del lenguaje
4	Generador de código	Selección detallada de instrucciones, optimizaciones dependientes de la máquina	**D:** Máquina. **I:** Lenguaje

Tabla 2.4: Las etapas de un compilador, sus funciones y los factores que influyen en ellas.

(como en las optimizaciones locales) pero que son válidas aun a través de saltos, reemplazar las apariciones de una variable A que ha recibido una sola asignación con otra variable (si sólo se ha hecho $A = X$ entonces A no tiene razón de ser, la cambiamos por X), sacar de un ciclo código que siempre calcula el mismo valor.

- Las optimizaciones dependientes de la máquina aprovechan características propias de la arquitectura, por ejemplo usar unidades de punto flotante en vez de emularlas, alteraciones del orden de código para aprovechar mejor el pipeline, reemplazar multiplicaciones por sumas y desplazamientos, elegir el tamaño óptimo de saltos.

Ahora bien, ¿como son alojadas y direccionadas las variables? ¿cuantos registros se necesitan para almacenarlas? Para responder estas preguntas primero debemos especificar que hay tres áreas donde las variables son almacenadas:

- El stack usado para alojar variables locales. Crece y decrece con las llamadas a funciones. Los objetos almacenados en él se acceden relativamente al apuntador el tope (stack pointer). Sólo se guardan escalares (tipos simples, no estructuras, no cadenas). En el stack básicamente

se ponen registros de activación, no se hacen prácticamente push's o pop's adicionales.

- El área de datos globales. Usada para alojar objetos declarados estáticamente: variables globales, constantes. Un porcentaje significativo de esta área se ocupa en arreglos o estructuras de datos agregadas.
- El *heap*. Usado para alojar objetos dinámicos que se acceden mediante apuntadores, típicamente no son escalares.

Cada área almacena ciertos tipos específicos de variables. ¿Cuáles de ellas podríamos poner en los registros del procesador? cualquiera, pero hay algunos problemas con las variables globales y con las cosas del heap. Por ejemplo si hay variables con alias, a las que puede uno referirse de varias maneras, entonces no podemos ponerlas en los registros porque se generaría código incorrecto. Por ejemplo, en el siguiente código:

```
p = &a;
a = 3+c;
...
*p = 2*r + k;
```

no puede ponerse la variable `a` en un registro del procesador porque es posible referirse a ella mediante `p` así que podría ser inconsistente el valor de `*p`.

En general las variables que casi siempre se pueden poner en registros son las del stack, las variables locales y parámetros de funciones.

2.6.1 Alojamiento en registros

Este es un problema interesante, queremos alojar el mayor número de variables posibles en los registros. Pero en general estos no nos van a alcanzar así que deberemos asignar las variables críticas, las que más se usen, en los registros y luego cambiar las variables asociadas a los registros cuando el régimen de uso cambie. Debo poder seleccionar que registro regresar a memoria y cual variable va a ir en él a partir de este momento. A esto se le llama *spill* (derramar).

Variables vivas y alojamiento en registros.

Cabe preguntarse, dado un segmento de código ¿cuantos registros nece-

sito para almacenar las variables?. Se dice que una variable está *viva* entre el punto del código donde recibió su *ultima* asignación[4] y el punto donde su valor deja de ser útil. Si se determinan los tiempos de vida de todas las variables es posible entonces hacer una gráfica en la que pongamos un vértice por cada variable y una arista entre dos vértices A y B si las variables asociadas a esos vértices están vivas simultáneamente en algún momento del código. Ahora, con este truco, es posible reducir el problema de encontrar el número de registros necesarios encontrando el mínimo número de colores con los que es posible pintar los vértices de la gráfica de tal forma que dos vértices unidos por una arista no tengan el mismo color. El número de colores será el número de registros necesario.

Usando este esquema es posible decidir cuando debe ser cambiada la variable asociada a un registro para ocupar el mínimo número de registros en todo momento. Por desgracia el problema de la colorabilidad de gráficas es NP-completo. Se usan sin embargo algunas heurísticas para resolverlo.

En el caso del código siguiente:

```
1   int mas_ceros (int n)
2   {
3     int num, unos, ceros, mitad, res;
4     num = n*n;
5     unos = ceros = 0;
6     while (num > 0)  {
7        mitad = num/2;
8        if (num == 2*mitad)
9           ceros++;
10       else
11          unos++;
12       num = mitad;
13    }
14    if (ceros > unos)
15      res = 1;
16    else
17      res = 0;
18    return res;
19  }
```

[4]En el caso de parámetros, este punto es el inicio de la función o procedimiento del que son parámetros.

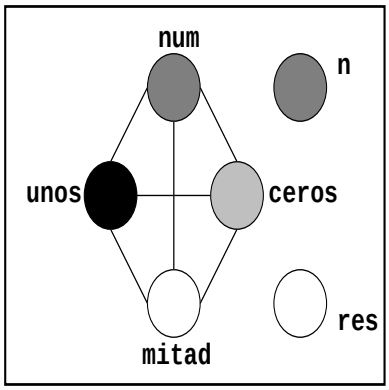

Figura 2.3: La gráfica correspondiente al programa de ejemplo. Cuatro colores son necesarios para colorearla, aquí se muestra una posible solución.

Los tiempos de vida de las variables (especificando línea de inicio y línea final) son:

`n`: 1 a 4

`num`: ¡Ah! pequeño problemita, esta variable está en un ciclo su última asignación leyendo linealmente el código sería la 12, pero deja de ser útil en la 6 que es anterior. A fin de cuentas le asignaríamos un registro desde la 4 para que en la 6 sea útil. Es decir su rango de vida es de la 4 a la 12.

`unos`: 5 a 14

`ceros`: 5 a 14

`mitad`: 7 a 12

`res`: 15 a 18

2.7 Formatos de instrucción de DLX

Registros en DLX.

Habiendo hecho el análisis anterior podemos diseñar el conjunto de instrucciones de una computadora ficticia, la del libro de texto llamada DLX (*De LuXe*).

Nuestra máquina tendrá 32 registros que llamaremos R0, R0, ..., R31. 32 es una buena cantidad de registros. Además estos registros serán de propósito general, lo que como hemos visto, gusta a los programadores de compiladores, salvo R0 que siempre contendrá la constante cero, lo que resultara conveniente.

Registros de punto flotante.

Adicionalmente incluiremos 32 registros de propósito general pero de uso exclusivo de una unidad de punto flotante. Será posible aparearlos de tal forma que un registro par y uno impar formen uno de 64 bits para números de punto flotante de doble precisión. Los llamaremos F0, ... F31, cuando esten apareados sólo nos referiremos a los pares.

Modos de direccionamiento en DLX.

Dado nuestro análisis de modos de direccionamiento sólo proveeremos de dos modos, inmediato y de desplazamiento, ambos con campos de 16 bits. Nótese que los modos de direccionamiento indirecto de registro y directo son casos particulares del modo de desplazamiento. En el modo de direccionamiento de desplazamiento se suma una constante al contenido de un registro y eso es usado como la dirección del operando. Así que si la constante es cero la dirección del operando es el contenido del registro (direccionamiento indirecto de registro). Si en cambio el registro tiene cero (para eso esta R0), entonces el operando está en la dirección de memoria indicada por la constante (direccionamiento directo). Así que con sólo proveer de dos modos, obtenemos otros dos gratuitamente

Además nuestra máquina será del tipo *load-store*, las únicas instrucciones que pueden acceder a memoria son `load` y `store`, las demás están obligadas a usar los registros para los operandos o a ponerlos inmediatos. Será de tres "direcciones" es decir en cada instrucción se dicen explícitamente los operandos y el lugar para guardar el resultado.

Formatos de instrucción en DLX.

Los formatos de las instrucciones se pueden ver en la figura 2.4. cada formato será usado como se describe a continuación:

Tipo I Para `load` y `store` de bytes, halfword y word. Para todas las instrucciones con operando inmediato (rd = rs1 op inmediato). Instrucciones de salto condicional, rs1 es el registro cuyo valor decide el salto, rd no se usa.

Tipo R Para instrucciones de la ALU registro-registro rd = rs1 func rs2. ¿Y entonces para que sirve el código de operación? en este caso el

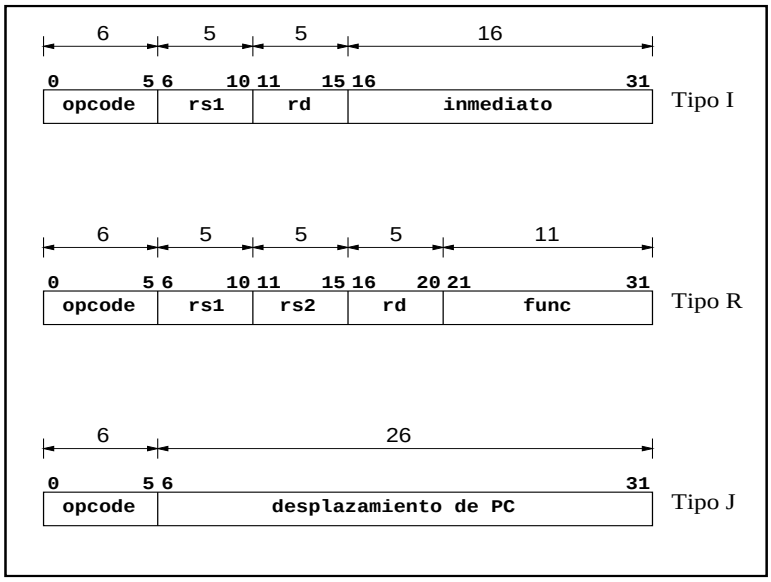

Figura 2.4: Formatos de instrucción de DLX.

opcode sólo dice que es una operación de ALU.[5]

Tipo J Saltos

2.7.1 Ejemplos

Cuando aparece un número como sibíndice de un "=" representa el número de bits transferidos. Cuando el sibíndice aparece en una expresión denota el o los bits que intervienen en la operación, 0 es el más significativo. Cuando aparece un número como superíndice de una expresión, denota cuantos bits de la expresión son establecidos, comenzando con el más significativo. el signo $\uplus$ denota yuxtaposición o concatenación de las cadenas binarias que intervienen en la operación.

[5]DLX está basada en la arquitectura real de MIPS (no en vano la diseño Hennessy) en MIPS las instrucciones tipo R, si tienen el opcode igual a 0 significa que esa operación corresponde a la ALU. Eso hubiera sido bueno hacerlo con todas las de la ALU, pero las que tenian operandos inmediatos no cabian en 32 bits, así que esas tuvieron que codificarse de otro modo.

Tipo I

1. `LW R1, 120(R6)`
   $\mathsf{regs[R1]} =_{32} \mathsf{mem[120 + regs[R6]]}$
   `rs1=R6`, `rd=R1`, `inmediato=120`
   direccionamiento de desplazamiento.

2. `LB R1, 1000(R2)`
   $\mathsf{regs[R1]} =_{32} (\mathsf{mem[1000 + regs[R2]]_0})^{24} \uplus (\mathsf{mem[1000 + regs[R2]]})$
   `rs1=R2`, `rd=R1`, `inmediato=1000`
   direccionamiento de desplazamiento.

3. `LH R1, 1000(R2)`
   $\mathsf{regs[R1]} =_{32} (\mathsf{mem[1000 + regs[R2]]_0})^{16} \uplus (\mathsf{mem[1000 + regs[R2]]})$
   $\uplus(\mathsf{mem[1001 + regs[R2]]})$
   `rs1=R2`, `rd=R1`, `inmediato=1000`
   direccionamiento de desplazamiento.

4. `LW R1, 120(R0)`
   $\mathsf{regs[R1]} =_{32} \mathsf{mem[120 + regs[R0]]} = \mathsf{mem[120]}$
   `rs1=R0`, `rd=R1`, `inmediato=120`
   direccionamiento directo o absuluto.

5. `LW R1, (R6)`
   $\mathsf{regs[R1]} =_{32} \mathsf{mem[regs[R6]]}$
   `rs1=R6`, `rd=R1`, `inmediato=0`
   direccionamiento indirecto o diferido con registro.

6. `SH R3, 512(R2)`
   $\mathsf{mem[512 + regs[R2]]} =_{16} \mathsf{regs[R3]_{16,\ldots,31}}$
   `rs1=R2`, `rd=R3`, `inmediato=512`
   direccionamiento de desplazamiento.

7. `ADD R1, R2, 320`
   $\mathsf{regs[R1]} = \mathsf{regs[R2]} + 320$
   `rs1=R2`, `rd=R1`, `inmediato=320`
   Direccionamiento inmediato, el `opcode` especifica suma (suma con inmediato).

8. `JR R4`
   $\mathsf{PC} = \mathsf{regs[R4]}$
   `rs1=R4`, `rd=inmediato=0`.

9. `BEQZ R4, etiqueta`
   Si R4 = 0 entonces
   PC = PC + 4 + distancia(PC + 4, etiqueta)
   si no PC = PC + 4
   `rs=R4` al comparador en vez de a la ALU,
   `rd=0`, `inmediato=distancia(PC+4, etiqueta)`.

10. `JALR R2`, "Jump and Link Register"
    regs[R31] = PC + 4, PC = R2
    `rs1=R2`, `rd=0`, `inmediato=0`
    nótese que `R31` se usa implícitamente como dirección de retorno.

Tipo R

1. `ADD R2, R3, R1`
   regs[R2] = regs[R3] + regs[R1]
   `rd=R2`, `rs1=R3`, `rs2=R1`
   direccionamiento de registro, el campo `func` especifica la operación a realizar, en este caso el `opcode` es cero (MIPS), significa operación de ALU con registros.

Tipo J

1. `J etiqueta`
   PC = PC + 4 + distancia(PC + 4, etiqueta)
   `desplazamiento=distancia(PC+4,etiqueta)`
   salto incondicional, en MIPS se utilizan estos 26 bits para pegarlos a los 4 bits más significativos de PC+4 y pegar luego dos bits menos significativos en cero (i.e. las direcciones de salto, de hecho todas las instrucciones, están alineadas, las direcciones son múltiplos de 4).

2. `JAL etiqueta`
   regs[R31] = PC + 4, PC = PC + 4 + distancia(PC + 4, etiqueta)
   `desplazamiento=distancia (PC+4,etiqueta)`
   llamada a subrutina.

CAPÍTULO 3

La ruta de datos y la microarquitectura

Ahora veremos que se necesita en un procesador para poder ejecutar las instrucciones de lenguaje de máquina y cuál es el proceso de ejecución de las mismas.

Ejecución de una instrucción.

A fin de cuentas cada instrucción de lenguaje de máquina debe traducirse de alguna manera en una serie de señales que viajan por diferentes cables. Algunas de estas señales le dicen a ciertos circuitos que hacer con otras señales en las que son traducidos los datos. La instrucción de suma que leímos de la memoria debe generar una señal para la ALU que le indica que debe hacer una suma y los otros campos de la instrucción deben ocasionar que los operandos lleguen al ALU por otro conjunto de lineas. La circuitería de la ALU calcula la suma y debe a su vez generar nuevas señales eléctricas que viajan hasta donde sea necesario para almacenar el resultado.

A los distintos subsistemas de un procesador que se encargan de labores específicas se les suele llamar unidades funcionales. Así por ejemplo la ALU es una unidad funcional y la unidad aritmética de punto flotante es otra. Pueden existir otras encargadas de calcular direcciones o de hacer las veces de interfaz con la memoria.

Nuestras computadoras actuales, en su mayoría son de arquitectura de Von Neumann. Eso significa que su arquitectura es, en esencia, la de la máquina IAS que Von Neumann diseño en el Instituto de Estudios Avanzados de Princeton. Se tiene una memoria donde se almacenan los datos y el programa a ejecutar, se tiene una unidad de control que secuencia las instrucciones que lee de la memoria, obtiene los datos y le dice a una unidad aritmético-lógica que operación efectuar, se tiene un registro para almacenar los resultados de la ALU (un acumulador a fin de cuentas) y la unidad de control transfiere también los resultados a la memoria. La característica distintiva de las arquitecturas de Von Neumann es que el programa es guardado también en la memoria.

Program Counter.

Para ejecutar un programa es necesario, entonces, tener en algún lugar la dirección en memoria de la siguiente instrucción a ejecutar, para saber que pedirle a la memoria cuando sea necesario traer de ella la siguiente instrucción (*fetch*). Tendremos entonces un registro especial en nuestro procesador que llamaremos PC (*program counter*) que apunta a la siguiente instrucción a a ejecutar.

Para asegurarnos de que el PC siempre apunta a la siguiente instrucción, debemos actualizarlo cada vez que ya hemos hecho el fetch de una instrucción. Como casi siempre la siguiente instrucción a ejecutar será la inmediata siguiente a la actual, cuya dirección está actualmente en PC y como todas nuestras instrucciones en DLX miden 4 bytes, entonces no hay más que incrementar en 4 el valor actual de PC. Además PC necesita decirle a la memoria que dirección quiere así que debe estar conectado con ella. Esto lo podemos poner gráficamente como se muestra en la figura 3.1.

Por supuesto en ocasiones habrá que cambiar PC de otra manera, cuando haya que hacer un salto. Pero de eso nos encargaremos luego.

Instruction Register.

Ya que la memoria nos entrega la instrucción la guardamos en un registro especial al que llamaremos *instruction register* o simplemente IR. Este está capacitado para partir la instrucción en campos, o mejor dicho, es el registro de trabajo del decodificador de instrucciones, una unidad encargada de partir la instrucción y determinar las acciones a tomar con base en el

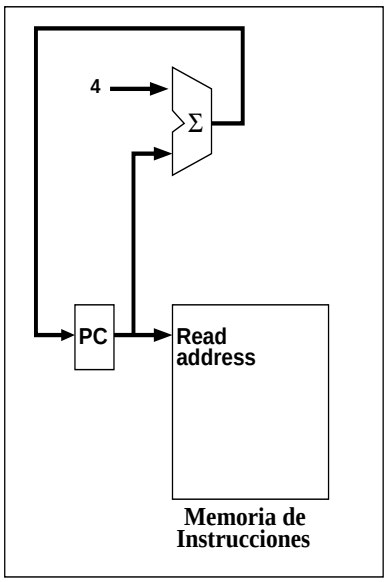

Figura 3.1: El PC y sus conexiones.

análisis de la instrucción.

Imaginemos que en IR hay una instrucción de suma, algo como `ADD R3, R2, R1`, suma R1 con R2 y guarda el resultado en R3 . La instrucción es del tipo R, y los números de registro están en los campos respectivos de la instrucción. Así que IR debe ser capaz de pedirle al banco de registros del CPU los dos operandos necesarios y de informarle que también deberá permitir la escritura en un tercer registro. IR debe estar conectado con el banco de registros mediante tres lineas, una para cada operando.

¿Qué tal si la instrucción es del tipo I, con operando inmediato?, quizás una operación aritmética. Entonces uno de los operandos aparece en la instrucción misma y su valor deberá pasar a la ALU. Así que IR además está conectado con la ALU mediante una linea. Aquí hay un pequeño problema. Decidimos poner 16 bits en el campo inmediato de la instrucción de tipo I porque determinamos que, estadísticamente, en la mayoría de los casos esa longitud basta. Pero nuestra ALU debe poder operar con los registros, que miden 32 bits, así que de alguna manera debemos "agrandar" nuestro campo inmediato para que sea de 32 bits. La manera de hacer eso es meter nuestro inmediato en un registro de 32 bits, uno especial al que llamaremos Imm , concretamente en los 16 bits menos significativos de Imm y luego añadirle a

ese registro sus 16 bits más significativos. Como vamos a hacer operaciones aritméticas, es entonces necesario asegurarnos de que no se altere el signo del operando inmediato, así que le pegamos 1's si era negativo y 0's si no lo era.

En síntesis IR tiene cuatro lineas de salida: tres al banco de registros y una indirecta a la ALU, que pasa antes por un dispositivo de extensión de signo (al que llegan 16 bits y salen 32) y luego por un registro de 32 bits llamado Imm , que entra como operando a la ALU.

El banco de registros debe entregar el contenido de los registros que le fueron solicitados por IR y entregarlos como operandos de la ALU. Además, si se ejecuta una instrucción aritmético lógica debe poder recibir el resultado para guardarlo en un registro, así que debe recibir como entrada la salida de la ALU. ¡Oops! pero que tal si se ejecuta un `load`, entonces debe recibir entrada de la memoria también, ambas entradas: la de la ALU y la de memoria, contienen el dato que hay que escribir en un registro, mismo que está especificado en uno de los campos de IR . ¡Ah! pero no pueden ocurrir simultáneamente ambas cosas, el banco de registros recibe el dato a escribir de una y sólo una de dos fuentes: la ALU y la memoria. Sólo una de las dos fuentes envía algo al banco de registros en un instante determinado, nunca ambas a la vez. Así que básicamente lo que hay que hacer es *multiplexar* la salida de la ALU y la entrada de la memoria para sacar una sola línea de entrada al banco de registros.

Quien hace todo el trabajo interesante es la ALU, que recibe sus dos entradas, a las que llamaremos A y B de:

- Ambas del banco de registros. Operación aritmético-lógica, instrucción tipo R.
- Una del banco de registros, otra del registro Imm . Operación aritmético-lógica, Instrucción tipo I.
- Una del registro Imm y otra del resultado de sumar 4 a PC . En el caso de saltos donde la dirección de salto esta indicada en el campo inmediato. Instrucción tipo I. En este caso el registro que hay que comparar con cero pasa a un comparador, no a la ALU.
- Una de NPC (así llamaremos a un registro donde almacenamos el valor de PC +4) y otra del banco de registros. Esta opción existe como posibilidad, pero no la utiliza DLX. Sería útil añadirla si tuviéramos

una instrucción que hiciera algo así como sumar NPC con el registro para obtener la dirección de un salto, habría que distinguirla de la instrucción JR actual, donde no se suma NPC al registro.

Con base en este análisis podemos determinar ponerle a la ALU dos entradas, cada una de ellas *multiplexa* dos entradas. Uno de los multiplexores recibe un registro y el resultado de PC luego de sumarle 4, que almacenaremos en un registro llamado NPC . El otro multiplexor recibe como entradas un registro del banco y el registro Imm . Cada multiplexor deja pasar sólo una de sus entradas, así tenemos cubiertas las cuatro posibilidades mencionadas arriba.

Saltos.

Conviene ahora discutir acerca de PC nuevamente. Como dijimos antes si la instrucción a ejecutar es un salto entonces el resultado de PC incrementado en 4 no es realmente la dirección de la siguiente instrucción, o mejor dicho, puede no serlo. Lo que el ensamblador debe hacer al encontrarse un salto a una etiqueta del programa es determinar cuál es la distancia del salto, es mucho más barato poner una distancia que poner una dirección absoluta. Esta distancia es la que se deberá poner en la instrucción y es, de hecho, lo que hay que sumarle al PC ya incrementado para obtener la dirección efectiva de la siguiente instrucción. Esto en algunos casos, en otros se debe tomar el contenido de un registro como el nuevo valor de PC .

Hay dos tipos de salto, condicional o incondicional. En nuestra arquitectura pondremos de ambos y en el caso de los condicionales pondremos instrucción de "salta si cero" y "salta si no cero" que verifican el contenido de un registro para determinar si saltar o no.

Así que una de las salidas del banco de registros debe pasar por un dispositivo de verificación que determina si es cero o no. La salida de A es la que debe entrar al comparador, es allí donde se guarda el operando `rs1` que se usa para estos saltos (instrucción tipo I). Luego el resultado de esta verificación sirve para determinar si la dirección de la siguiente instrucción a ejecutar será la de NPC (PC +4) o bien la otra alternativa: el resultado, obtenido por la ALU, de sumar una cantidad extra al contenido de NPC .

Sinteticemos. Ya habíamos dicho que PC , luego de traerse la instrucción a ejecutar debía incrementar su valor en 4, este valor nuevo lo guardamos en un registro llamado NPC , así que no tenemos más que copiar NPC a PC y ya. Pero esto no es cierto siempre. Si la instrucción a ejecutar es un salto entonces el contenido de NPC puede o no, ser la dirección de la siguiente

instrucción. Si no lo es, se debe a que un registro del procesador cumplió con la condición necesaria para saltar o bien es un salto incondicional y entonces es el resultado de la ALU lo que debe quedar como nuevo PC . Entonces la entrada de PC es, en realidad, la salida de un multiplexor que recibe, por una parte, NPC y por otra la salida de la ALU, el multiplexor decide a quién dejar pasar en función de la salida de un dispositivo de verificación de condición.

Bien, luego de todo este rollo mareador podemos ver el diagrama que resulta en la figura 3.2. A este tipo de diagramas se les conoce como ruta de datos (*datapath*).

Load Memory Data.

Si la instrucción a ejecutar es una de carga de memoria, entonces, luego de calcular la dirección absoluta del dato requerido de la memoria, esta debe ser pasada a la memoria junto con una petición de lectura. Una vez que la memoria trae el dato requerido de regreso debe poder guardarse en algún lugar temporalmente, este registro se llama LMD (*Load Memory Data*).

En las instrucciones de carga y en las aritmético-lógicas debemos escribir el resultado de la operación en un registro del CPU. Así que la entrada de datos del banco de registros proviene de una de dos fuentes: LMD o la salida de la ALU. Otra vez pasamos ambas por un multiplexor del que sólo sale una de las dos para escribirse en el registro del procesador.

3.1 Señales de control

Hemos dicho que ejecutar una instrucción significa, esencialmente, distribuir una serie de señales por diferentes lineas de transmisión que se encargan de: habilitar o deshabilitar la entrada a ciertas unidades funcionales de otras lineas (por ejemplo, determinar que debe salir de un multiplexor), indicar a ciertas unidades funcionales que operación realizar (por ejemplo, decirle a la ALU que operación efectuar).

Unidad de control.

La encargada de determinar que señales enviar y en que secuencia es llamada *unidad de control*. Por supuesto su entrada es el IR (puede haber otras entradas más, pero esta es la fundamental) y su salida son todas las lineas transportadoras de señales de control que llegan a las distintas unidades funcionales.

Nos toca decidir ahora cuáles señales son necesarias y posteriormente,

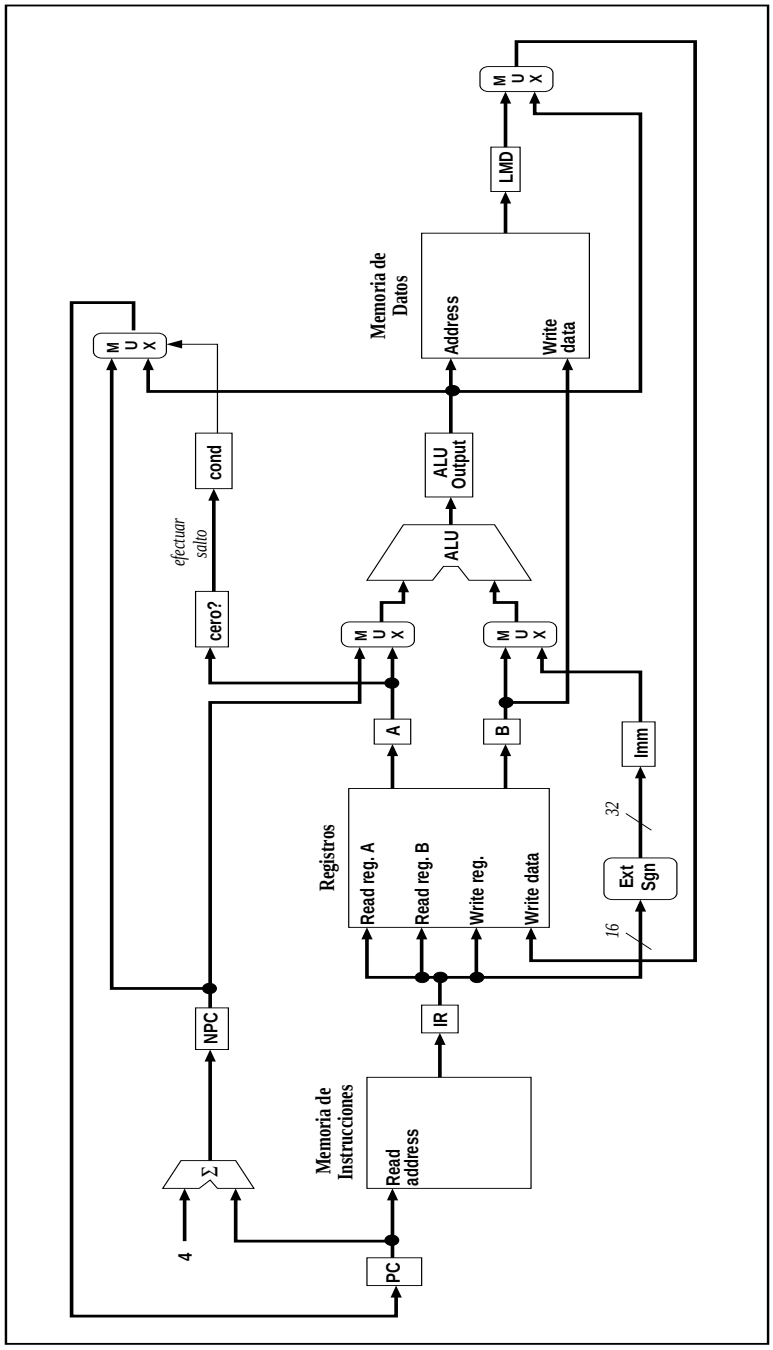

Figura 3.2: La ruta de datos (*datapath*) de DLX.

bajo que condiciones una señal debe estar prendida.

Señales de control.

En nuestra ruta de datos de DLX se requiere de las siguientes señales de control:

RegWrite Para especificar que el banco de registros debe aceptar la entrada `WriteData` y escribirla en el registro especificado en `RegWrite`. 1=escribir, 0=ignorar.

ALUSrc1 0 significa que la entrada de la ALU debe ser NPC . 1 que la entrada debe ser el registro A .

ALUSrc2 0 significa que la entrada de la ALU debe ser Imm . 1 que la entrada debe ser el registro B .[1]

ALUCtrl Especifica la operación que debe efectuar la ALU. Deben ser unos tres o cuatro bits.

PCsrc Decide quien debe ser el contenido de PC . 0 significa que el contenido debe ser forzosamente NPC , 1 significa que puede ser NPC o la salida de la ALU dependiendo de si la condición de salto se cumple o no.

MemRead Le indica a la memoria que lea la localidad indicada en `Adress` y regrese su contenido en LMD . 1 significa lectura, 0 no lectura.

MemWrite La indica a la memoria si debe o no ser escrito el dato `WriteData` en `Adress`. 1 significa escritura, 0 no escritura[2]

RegSrc Indica quien debe pasar como el dato a escribir en el banco de registros. 0 significa que pasa LMD y 1 que pasa el resultado de la ALU.

¿Como se vería entonces la ejecución de una instrucción? Esencialmente se ve entonces como una secuencia de señales de control. En la tabla 3.1 se muestran las señales de control asociadas con ciertas instrucciones.

[1] Recuérdese que no es válida la combinación ALUSrc1=0, ALUSrc2=1.

[2] Esto significa que si la instrucción ejecutándose no es `load` o `store` tanto esta señal como la anterior deben ser cero.

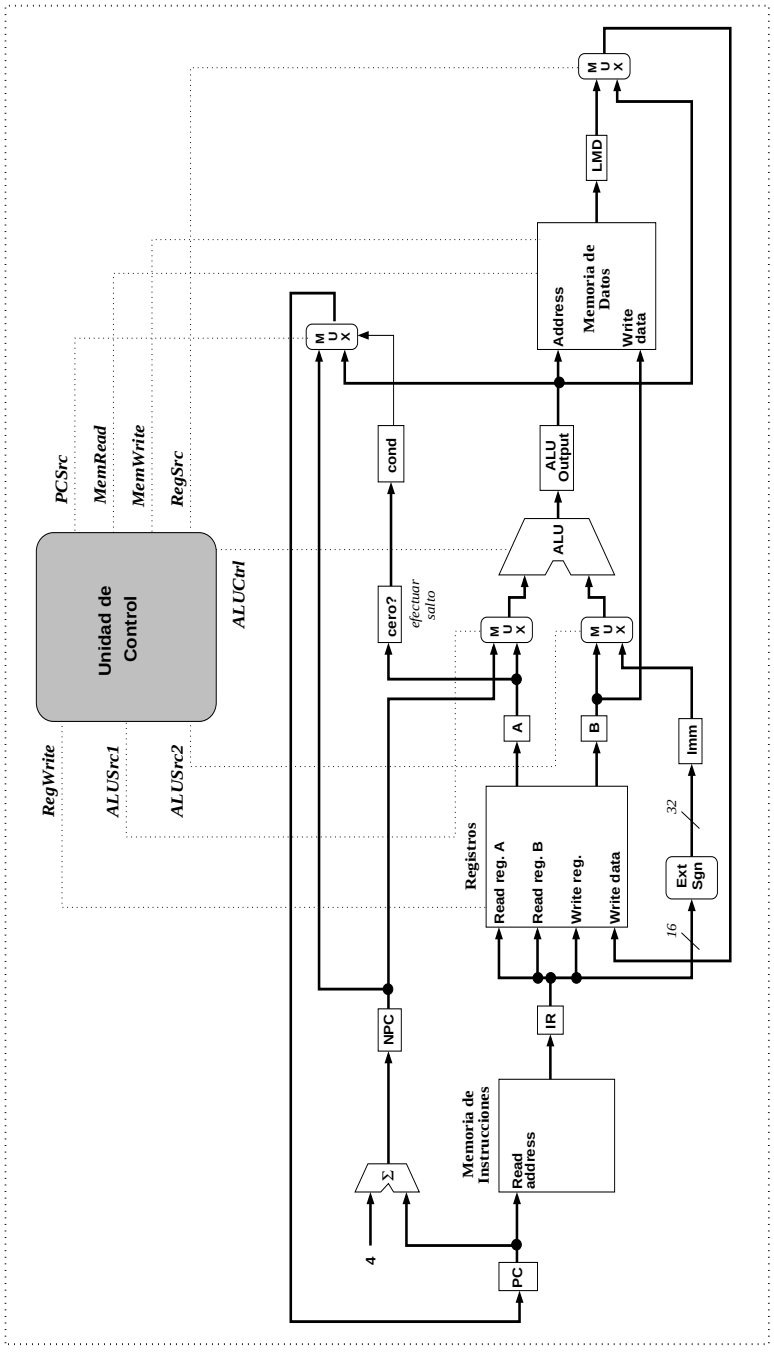

Figura 3.3: La ruta de datos (*datapath*) de DLX con señales de control.

Instr.	RegWr	ALUSr1	ALUSr2	ALUCtrl	PCSr	MRd	MWr	RegSrc
`ADD R3,R2,R1`	1	1	1	Suma	0	0	0	1
`SUB R3,R2,R1`	1	1	0	Resta	0	0	0	1
`LW R3,120(R2)`	1	1	0	Suma	0	1	0	0
`SW R3,120(R2)`	0	1	0	Suma	0	0	1	X
`BEQZ R3,salto1`	0	0	0	Suma	1	0	0	X

Tabla 3.1: Ejemplos de instrucciones y las señales de control que generan.

3.2 Controladores, clasificación

Ejecutar una instrucción es traducirla a señales de control que se generan en un cierto orden. Es decir, de alguna manera se debe secuenciar el envío de estas señales a través de las lineas de salida de la unidad de control. En el caso de DLX no es tan evidente.

Imaginemos que tenemos una máquina cuya ruta de datos está determinada por un único canal bidireccional que comparten todas las unidades funcionales, un bus. Si este fuera el caso entonces para hacer algo como el fetch deberíamos poder sacar al canal el contenido de PC , luego dar permiso de que esta señal pase como entrada a la ALU junto con un 4, luego hacer la suma y posteriormente sacar esa suma al canal para que sea aceptada nuevamente por PC . Mientras esto se hace no es posible sacar el contenido de IR , dado que hay un solo canal habría colisión de datos en él si intentáramos hacer ambas cosas al mismo tiempo. Las señales de control deben seguir un orden.

Control microprogramado.

Lo primero que nos viene a la mente es entonces poner un área de memoria ROM dentro del CPU. Esa memoria esté constituida de palabras de una longitud tal que permite poner todas las señales de control en una palabra. Con este esquema tendríamos entonces una serie de programas, secuencias de señales de control, que al ejecutarse hacen realidad las instrucciones de lenguaje de máquina. Tendríamos entonces un pequeño programa que ejecutar para llevar a cabo un `ADD` con direccionamiento registro-inmediato, por ejemplo, uno para hacer cada posible `LW` dependiendo de el modo de direccionamiento usado, etc.

A esta memoria donde se almacenan las secuencias de señales de control se le conoce como micromemoria y a los programas almacenados allí microprogramas.

Horizontal Vs. vertical.

Hay dos posibilidades para almacenar microprogramas. Una es hacer la palabra de micromemoria tan larga como sea necesario para que quepan, bit a bit, todas las señales de control en cada palabra de la micromemoria. Otra es codificar estas señales para abreviar la expresión de todas las posibles señales. Algo así estamos haciendo al decir que con tres bits podemos decir la operación que la ALU debe llevar a cabo. En principio podríamos, si quisiéramos, poner tantas señales como operaciones tiene la ALU en vez de codificarlas. Si tuviéramos 16 operaciones tenemos la opción de enviar un uno por exactamente una de 16 lineas de un bit, una por cada operación, o bien enviar sólo 4 bits expresando un número entre 0 y 15 y que indica inequívocamente, la operación a realizar.

Si se ponen todas las señales de control sin codificación se dice que el controlador microprogramado tiene organización horizontal. Si se codifican las señales de control agrupándolas, entonces se dice que tiene organización vertical.

Por supuesto hay un compromiso. Mucha horizontalidad hace que se necesiten palabras más largas de memoria, es decir, más memoria a fin de cuentas; pero hace trivial el envío de las señales de control, sólo hay que sacar bit por bit la palabra de memoria a las lineas de transmisión de señales. Si posee mucha verticalidad entonces hay que decodificar cada subconjunto de bits y luego traducir eso a las señales sobre las lineas de transmisión, eso es más tardado, pero por supuesto se requiere de menos memoria.

Controladores alambrados (*hardwired*).

Los controladores microprogramados son sólo una posibilidad. Con el advenimiento del paradigma RISC, los conjuntos de instrucciones se han simplificado: poca diversidad de formatos, longitud pequeña y fija, catálogo pequeño, operaciones sencillas, pocos modos de direccionamiento. Esto hace posible que, en vez de tener una micromemoria con una multitud de microprogramas diferentes, se puedan hacer circuitos combinacionales temporizados o secuenciales (máquina de estados finitos) que implementen el envío de señales de control. Es decir, si hay pocas señales, pocas combinaciones válidas y pocas secuencias, como en una máquina RISC, entonces es posible saltarse la microprogramación y hacer un controlador alambrado (*hardwired controller*). Esto, por supuesto, acelera la ejecución de las instrucciones y mejora notablemente el desempeño.

CAPÍTULO 4

Pipelining

4.1 El concepto de Pipeline

El concepto de *pipeline*[1] es realmente anterior a las computadoras digitales. En realidad es el mismo concepto que el de linea de producción que Henry Ford puso en práctica durante la fabricación del modelo T a principios del siglo XX.

En esencia pipeline significa que se comienza a ejecutar una instrucción antes de que la inmediata anterior termine de ejecutarse. En el capítulo pasado planteamos la ejecución de una instrucción como el envío de diversas señales de control y datos a través de la ruta de datos de la computadora. No

[1] La traducción de *pipeline* es "tubería" y entonces *pipelining* sería "entubamiento", nada que ver con el concepto que necesitamos. Así que en general usaremos el término en inglés.

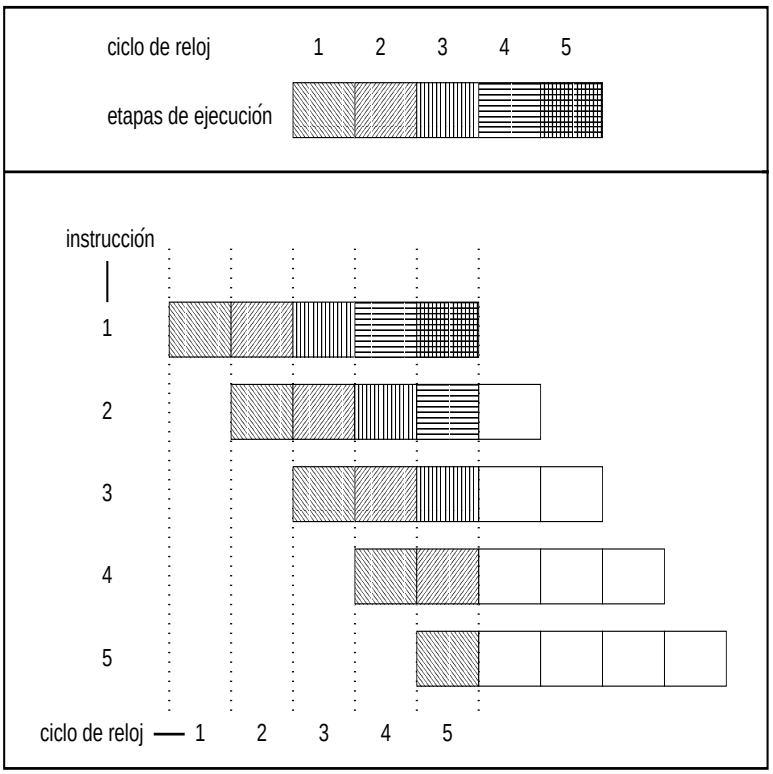

Figura 4.1: En una máquina X normalmente una instrucción toma 5 ciclos de reloj en su ejecución y hasta que se termina de ejecutar es posible ejecutar la siguiente. Si la máquina en cuestión tuviera pipeline entonces es posible ejecutar el mismo tiempo diferentes trozos de diferentes instrucciones.

es posible enviar todas las señales de una sola vez sino que hay que hacerlo por etapas. Cada instrucción de lenguaje de máquina es, en realidad, la ejecución de un pequeño "programa"[2]. Como vimos también cada uno de estos pasos puede ser hecho en un solo ciclo de reloj, así que podríamos pensar en ejecutar el paso i de la instrucción j al mismo tiempo que el paso $i-1$ de la $j+1$ y que el $i-2$ de la $j+2$ y así hasta tener tantas instrucciones ejecutándose como ciclos de reloj son necesarios para ejecutarlas. Esto se muestra esquemáticamente en la figura 4.1.

Pipeline y rendimiento (*throughput*).

Hay que hacer énfasis en que el pipeline no disminuye el tiempo necesario

[2]Usamos aquí la palabra programa para hacer claro el proceso secuencial, paso a paso. Como ya sabemos, en realidad este "programa" puede estar alambrado.

para ejecutar una instrucción dada. En el caso de la figura 4.1, la instrucción 1 sale del procesador luego de cinco ciclos de reloj, haya o no pipeline. La mejora del pipeline consiste en aumentar el número de instrucciones que salen del procesador por unidad de tiempo, el *throughput*. Para ejecutar 5 instrucciones en la máquina de la figura, sin pipeline, se requiere de 25 ciclos de reloj, 5 por cada instrucción. En la misma máquina con pipeline las cinco instrucciones requieren 5 ciclos de reloj para ejecutarse con el pipeline completamente lleno o 9 en el peor de los casos, cuando el pipeline empieza a llenarse, como se muestra en la figura 4.1.

Para que el pipeline funcione bien es necesario hacer que ninguna de las etapas de la ejecución de una instrucción sea más tardada que las demás. Como no podemos ir amontonando instrucciones a la entrada de esa etapa tendríamos que retardar todas las etapas anteriores. Este es el reto más importante para diseñar un pipeline, balancear lo mejor posible la duración de las etapas. Si todo está balanceado:

$$T_{ins}^{pipeline} = \frac{T_{ins}}{N}$$

donde $T_{ins}^{pipeline}$ es el tiempo requerido por instrucción en la máquina con pipeline (si me paro con un reloj junto al procesador, cada cuanto tiempo veo que sale una instrucción), T_{ins} es el tiempo por instrucción sin pipeline y N es el número de etapas en el pipeline. El pipeline tiene el efecto de reducir CPI.

En general las etapas de pipeline, que son etapas de ejecución de instrucción tardan uno o a lo más dos ciclos de reloj.

4.2 Revisión del ciclo de ejecución

En DLX podemos dividir naturalmente el ciclo total de ejecución de una instrucción en cinco etapas:

1. Fetch. Se carga la instrucción a ejecutar y se incrementa el PC .
2. Decodificación. Se determina la instrucción a ejecutar, se transfieren los operandos a la ALU.
3. Ejecución. Se efectúa la operación requerida en la ALU.

4. Acceso a memoria. Si se trata de un load o un store se transfiere el resultado de la ALU a la memoria como dirección.

5. Escritura de resultados. El resultado de la ALU o el valor regresado por la memoria se escribe en los registros del procesador.

Para ser más precisos lo que se hace en cada etapa es:

IF (*Instruction Fetch*)
IR = mem[PC], NPC = PC + 4

ID (*Instruction Decode*)
A = regs[$IR_{6,\ldots,10}$], B = regs[$IR_{11,\ldots,15}$], Imm = $((IR_{16})^{16} \uplus IR_{16,\ldots,31})$. (recuérdese los formatos de la figura 2.4, A es `rs1` y B es `rd`).

EX (*Execution*).

- Referencia a memoria. ALUOutput = A + Imm, luego de esto ALUOutput tiene la dirección efectiva del load o store.
- Instrucción aritmético-lógica registro-registro.
  ALUOutput = A func B.
- Operación aritmético-lógica registro-inmediato.
  ALUOutput = A oper Imm.
- Salto. ALUOutput = NPC + Imm, Cond = (A comp 0).

MEM (*Memory*) Acceso a memoria o finalización de salto. PC = NPC

- Referencia a memoria.
  LMD = mem[ALUOutput] o bien
  mem[ALUOutput] = B
- Salto. if (cond) PC = ALUOutput.

WB (*Write Back*).

- instrucción aritmético-lógica, registro-registro.
  regs[$IR_{16,\ldots,20}$] = ALUOutput
- Instrucción aritmético-lógica, registro-inmediato.
  regs[$IR_{11,\ldots,15}$] = ALUOutput
- Instrucción de carga. regs[$IR_{11,\ldots,15}$] = LMD

Ejemplos

- `LW R1, 30(R2)`

  IF IR = mem[PC], NPC = PC + 4

  ID A = regs[$IR_{6,...,10}$] = regs[R2] (`rs1`), B = regs[$IR_{11,...,15}$] = regs[R1] (`rd`), Imm = 30

  EX ALUOutput = A + Imm = 30 + regs[R2]

  MEM PC = NPC, LMD = mem[ALUOutput] = mem[30 + regs[R2]]

  WB regs[$IR_{11,...,15}$] = regs[R1] = LMD

- `SW R1, 30(R2)`

  IF IR = mem[PC], NPC = PC + 4

  ID A = regs[$IR_{6,...,10}$] = regs[R2] (`rs1`), B = regs[$IR_{11,...,15}$] = regs[R1] (`rd`), Imm = 30

  EX ALUOutput = A + Imm = 30 + regs[R2]

  MEM PC = NPC, mem[ALUOutput] = mem[30 + regs[R2]] = B = regs[R1]

  WB No se usa

- `J etiqueta`

  IF IR = mem[PC], NPC = PC + 4

  ID Imm = dist(PC + 4, etiqueta)

  EX ALUOutput = NPC + Imm

  MEM PC = ALUOutput

  WB No se usa

- `BEQZ R4, etiqueta`

  IF IR = mem[PC], NPC = PC + 4

  ID A = regs[$IR_{6,...,10}$] = regs[R4], B = regs[R0], Imm = dist(PC + 4, etiqueta)

  EX ALUOutput = NPC + Imm

  MEM PC = NPC, if (regs[R4] == 0)PC = ALUOutput

  WB No se usa

- `ADD R1, R2, R3`

IF IR = mem[PC], NPC = PC + 4

ID A = regs[$IR_{6,...,10}$] = regs[R2], B = regs[R3], Imm = regs[R1] + func, esto último no tiene sentido, pero no importa, no se usará.

EX ALUOutput = A + B

MEM PC = NPC

WB regs[$IR_{16,...,20}$] = regs[R1] = ALUOutput

4.3 Registros de pipelining

Con este esquema de división de tareas a realizar para la ejecución de una instrucción, podemos pensar en ponerle un pipeline a DLX. Cada etapa: IF, ID, EX, MEM y WB puede verse como una etapa de pipeline, cada una llevándose a cabo en un ciclo de reloj. Esto significa que cuando el pipeline este completamente lleno, luego de que se estén ejecutando las primeras 5 instrucciones, tendremos, en cada ciclo de reloj, una instrucción en cada etapa. Todas las etapas ocurren en cada ciclo de reloj, cada una sobre diferente instrucción. Esto también trae algunos problemas.

En el ciclo ID se leen dos registros del banco de registros, en WB se escribe uno de ellos. Hay que habilitar entonces el banco de registros para que pueda leer dos registros y escribir uno en un solo ciclo de reloj, no hay problema, esto se puede hacer, pero, ¿qué tal si una escritura y la lectura involucran al mismo registro?. Hablaremos de esto un poco más adelante.

En cada ciclo hay que traer una nueva instrucción, pero el PC se actualiza hasta el ciclo cuatro, MEM, ¿qué tal si hay un salto? ¿como direccionamos la siguiente instrucción? Esto lo vamos a medio solucionar permitiendo que en la etapa IF misma se actualice el valor de PC con PC +4. Si la instrucción es un salto ya veremos como hacer mas adelante.

Imaginemos que en un solo ciclo de reloj se ejecuta la etapa ID de un `ADD` registro-registro y la etapa EX de un `load`. En el caso del `ADD` en A y B se guardarán los operandos. En el caso de el `load` en ese mismo ciclo la ALU calcula la suma de A e Imm . ¿Cual valor de A se toma?.

Para resolver el último problema mencionado en particular y en general, los problemas que involucran el acceso a registros temporales (A , B , Imm , ALUOutput, IR , NPC , Cond y LMD), se añadirán registros grandes que permitan mantener el estado de los registros temporales entre etapas del

pipeline. Estos registros se nombrarán usando el identificador de las etapas entre las que se encuentran: IF/ID, ID/EX, EX/MEM y MEM/WB. Cualquier valor temporal almacenado en uno de estos registros y que sea necesario acceder en etapas posteriores de ejecución de la instrucción actual, deberá ser copiado al registro siguiente acarreandolo hasta que ya no sea necesario.

En el caso mencionado, por ejemplo, suponiendo que el `load` es la instrucción más avanzada en su ejecución y que el `ADD` es más reciente: cuando el `load` pasó por la etapa ID, se cargó al principio de ella, en el campo correspondiente al registro A de IF/ID, el registro que será usado como parámetro, cuando load pasa a la siguiente etapa, se copia el registro IF/ID completo al registro ID/EX donde se guarda el valor útil de A para el load, mientras que en ese mismo ciclo se escribe también el registro A o mejor dicho el campo A , pero del registro IF/ID, con el valor que le será útil al `ADD`. Podría objetarse que queremos leer y escribir en un solo ciclo de reloj en el mismo registro, en este caso, en el campo A del registro IF/ID, pero esto se puede, siempre queremos leerlo en ID/EX antes de escribirlo, podemos hacerlo con unos flip-flops de disparo de flanco, de tal forma que cuando suba el reloj se lea de ellos y cuando baje el reloj, en ese mismo ciclo, se escriban. El resultado de las modificaciones hechas se muestra en la figura 4.2.

Haremos referencia a los distintos campos de estos registros usando la notación Ee/Es.campo donde Ee es la etapa de entrada, Es es la de salida y campo es el nombre del campo del registro que nos interesa y que generalmente es el nombre de algunos de los registros temporales que teníamos en el flujo de datos sin pipeline.

Con esto en mente podemos describir lo que ocurre en cada etapa de ejecución:

IF IF/ID.IR = mem[PC]
IF/ID.NPC, PC = ((EX/MEM.opcode == branch) & EX/MEM.cond)?
EX/MEM.ALUOutput : PC +4

ID ID/EX.A = regs[IF/ID.IR$_{6,\ldots,10}$]
ID/EX.B = regs[IF/ID.IR$_{11,\ldots,15}$]
ID/EX.NPC = IF/ID.NPC
ID/EX.IR = IF/ID.IR
ID/EX.Imm = (IF/ID.IR$_{16}$)^{16} $\uplus$ IF/ID.IR$_{16,\ldots,31}$

EX • Instrucción aritmético-lógica

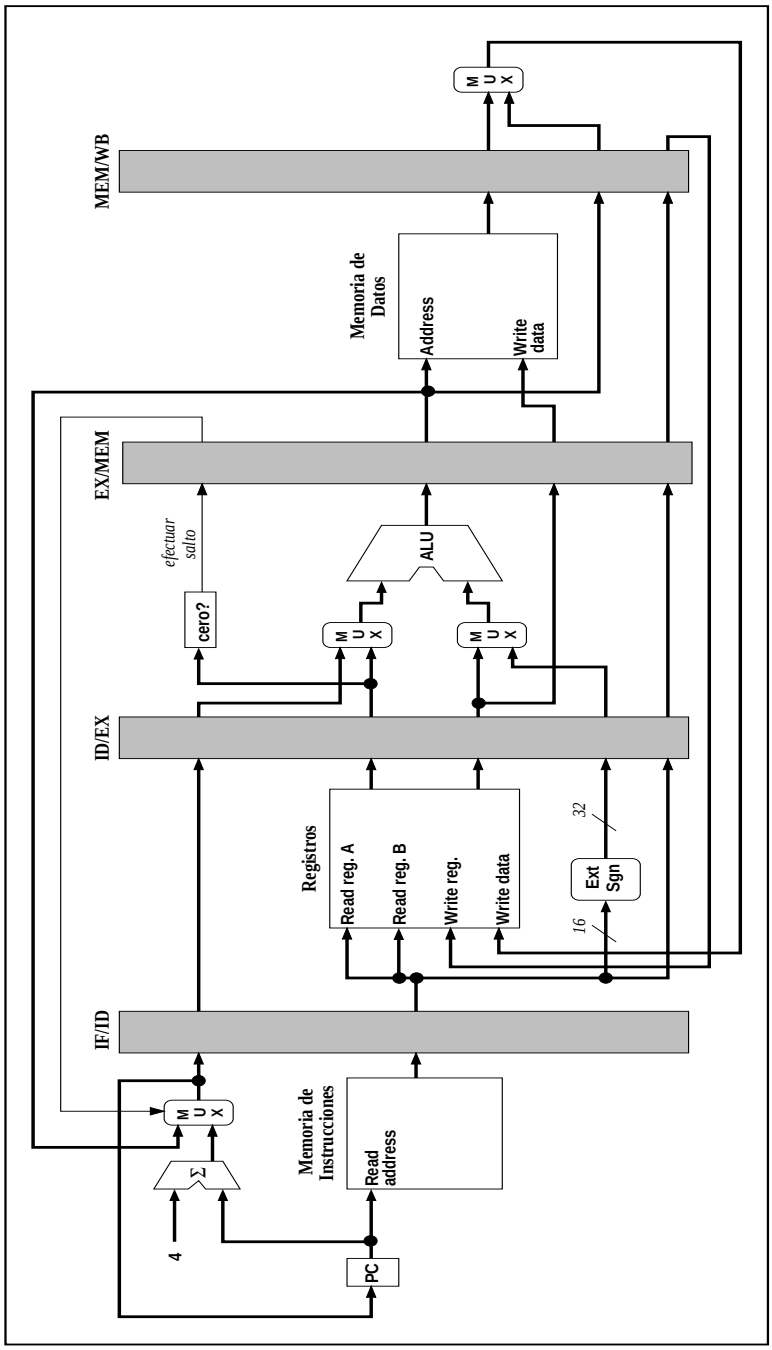

Figura 4.2: DLX preparada para pipeline.

  - registro-registro
    EX/MEM.IR = ID/EX.IR
    EX/MEM.ALUOutput = ID/EX.A func ID/EX.B
    EX/MEM.cond = 0

  - registro-inmediato
    EX/MEM.IR = ID/EX.IR
    EX/MEM.ALUOutput = ID/EX.A op ID/EX.Imm
    EX/MEM.cond = 0

- load o store
  EX/MEM.IR = ID/EX.IR
  EX/MEM.ALUOutput = ID/EX.A + ID/EX.Imm
  EX/MEM.cond = 0
  EX/MEM.B = ID/EX.B

- salto
  EX/MEM.ALUOutput = ID/EX.NPC+ID/EX.Imm
  EX/MEM.cond = (ID/EX.A comp 0)

MEM

- Instrucción aritmético-lógica
  MEM/WB.IR = EX/MEM.IR
  MEM/WB.ALUOutput = EX/MEM.ALUOutput
- load
  MEM/WB.IR = EX/MEM.IR
  MEM/WB.LMD = mem[EX/MEM.ALUOutput]
- store
  MEM/WB.IR = EX/MEM.IR
  mem[EX/MEM.ALUOutput] = EX/MEM.B

WB

- Instrucción aritmético-lógica

  - registro-registro
    regs[MEM/WB.$IR_{16,...,20}$] = MEM/WB.ALUOutput
  - registro-inmediato
    regs[MEM/WB.$IR_{11,...,15}$] = MEM/WB.ALUOutput

- load
  regs[MEM/WB.IR$_{11,\ldots,15}$] = MEM/WB.LMD

4.4 Desempeño de pipeline

Ya hemos dicho que el pipeline incrementa el rendimiento (*throughput*) del procesador y que no se reduce el tiempo que una instrucción cualquiera tarda en ejecutarse. De hecho el tiempo que toma la ejecución de una instrucción se incrementa por la carga extra que implica el pipeline. Por ejemplo en nuestra arquitectura sin pipeline no teníamos que copiar e ir cargando los registros temporales de etapa en etapa.

Supongamos, por ejemplo, que tenemos una máquina sin pipeline cuyos ciclos de reloj son de 10 ns. y que la frecuencia de uso y duración de sus instrucciones es la siguiente:

- Operaciones de ALU, 40%, 4 ciclos
- Saltos, 20%, 4 ciclos
- Operaciones de memoria, 40%, 5 ciclos

El tiempo promedio de ejecución de instrucción es:

$$\overline{T} = 10 \times [4 \times (40 + 20) + 5 \times 40] = 44\ ns.$$

Supongamos que le añadimos pipeline a la máquina y que eso genera un retraso de 1 ns. por cada ciclo de reloj. Entonces los ciclos quedan de 11 ns. y cada 11 ns. una instrucción se termina de ejecutar. Así que el *speedup* ganado es:

$$S = \frac{44}{11} = 4$$

La máquina con pipeline es 4 veces mejor que la original.

4.5 Diseño para facilitar el pipeline

Hay varios factores a considerar para hacer efectivo el pipeline:

1. Si la longitud de instrucción es muy variable es difícil establecer una linea de pipeline fija. Si hay mucha variedad en la longitud de las instrucciones hay, en general, mucha variedad en tiempos de ejecución. En un 80X86, por ejemplo, las instrucciones miden de 1 a 17 bytes de longitud, simplemente para hacer fetch ya hay que haber comenzado a decodificar, tendremos que frenar el proceso de ejecución para traer lo que falta de una instrucción larga.

2. Si la arquitectura no es load-store entonces puede no haber una única etapa del ciclo de ejecución en la que se accede a la memoria, pueden ser varias y puede ser que haya que hacerlas de manera exclusiva. Si es una máquina registro-memoria, en las que a lo más un operando está en memoria, entonces puede ser que exista una sola etapa de acceso a memoria, pero si se usan modos de direccionamiento que no sea el directo de registro, hay que calcular la dirección y hacer la operación con una sola ALU (si es el caso), las cosas se complican.

3. Si el formato de instrucción es fijo entonces es fácil saber tempranamente cuál es el destino y cuales son los operandos y se pueden cargar en los registros y tenerlos listos para la operación, cualquiera que esta sea. Es decir esto se puede hacer al mismo tiempo que se decodifica la instrucción. De otro modo hay que determinar primero que instrucción es, para determinar donde están los operandos. Se añade una etapa.

4. Si se fuerza a que los datos estén alineados en memoria cada lectura/escritura implica un solo acceso a memoria. Si esto no es así es posible requerir de más de un acceso implícitamente en cada lectura/escritura, esto retarda toda la línea de pipeline.

4.6 Imponderables (*hazards*)

Al inicio de este capítulo ya hemos apuntado algunos problemas que pueden ocurrir al implementar un pipeline. Nos avocaremos ahora a analizar estos y otros problemas llamados imponderables o *hazards*.

Clasificación de *hazards*.

Estos problemas que, de hecho disminuyen el desempeño del pipeline pueden clasificarse de acuerdo a su origen:

- Hazards estructurales. Son aquellos ocasionados por conflictos en el

```
0   ; for (i=0 i<num; i++) {
1            addi R1, R0, 0
2            lw   R2, num(R0)
3   repite:  slt  R3, R1, R2
4            beqz R3, fuera
5            ; instrucciones del ciclo
6            ; ....
7            addi R1, R1, 1
8            j    repite
9   fuera:   sw   R7, resultado(R0)
10           ;.....
```

Figura 4.3: Un ejemplo de hazard de control. Si es tomado (es decir, ocurre) el salto de la línea 4, entonces la instrucción que sigue es la de la línea 9, sin embargo las instrucciones que se han traído y que están en proceso de ejecución son las que se encuentren después de la línea 4, las del ciclo.

uso de hardware. Por ejemplo, si no tenemos dividida de alguna manera la memoria de almacenamiento de datos de la de almacenamiento de código, entonces en un solo ciclo de reloj trataríamos de ejecutar el ciclo IF y el MEM de diferentes instrucciones y eso ocasionaría un conflicto irresoluble, no es posible acceder dos veces a la memoria en un solo ciclo de reloj. La única manera de solucionar esto es retrasando toda la linea de pipeline, todas las instrucciones que siguen a la que se encuentra con el conflicto. A esto se le conoce como retrasar o "atorar" (*stall*) el pipeline. A los ciclos de reloj desocupados se les llama burbujas (*bubbles*).

- Hazards de control. Se toma una decisión que cambia el control de flujo del programa, esta decisión la toma una instrucción mientras las que, en principio, le seguirían están en proceso de ejecución (ver figura 4.3).

- Hazards de datos. Un resultado anterior se necesita más adelante, antes de que se haya obtenido (ver figura 4.4).

Para contribuir a resolver un hazard de control, como el del ejemplo, es que movimos el multiplexor que interviene en la actualización de PC a la izquierda dos etapas (compárense las figuras 3.2 o 3.3 con la 4.2). Con esto la línea de pipeline se entera, lo antes posible, de que se deben ejecutar las

```
1       add  R1, R2, R3  ; establece R1=R2+R3
2       sub  R6, R1, R4  ; requiere R1
```

Figura 4.4: Un ejemplo de hazard de datos. La primera instrucción establece el valor de R1, la segunda lo utiliza. En una DLX sin pipeline no hay problema. En DLX con pipeline R1 es establecido por la primera instrucción en la etapa WB, pero la segunda instrucción lo requiere en la etapa ID. Cuando la primera esté en WB la segunda ya debería estar en EX, una etapa adelante de ID.

instrucciones de salto. En DLX la distancia entre EX, que determina el nuevo contenido de PC e IF, etapa en la que se encuentra la operación que se da cuenta de que la siguiente instrucción e ejecutar es la de destino del salto, es dos, así que hay que tirar a la basura lo que se ha hecho de dos instrucciones: la que está en ese momento en IF y la que está en ID, dos ciclos de reloj.

Solución de *hazards* de control.

Hay dos posibilidades para tratar de resolver un hazard de control:

1. Atorar el pipeline, siempre que se traiga una instrucción de salto, hasta determinar si se va a saltar o no.
2. Predecir si el salto será tomado o no. Por ejemplo, en un ciclo como el mostrado en la figura 4.3 las instrucciones del ciclo se ejecutan `num` veces. Es decir: de `num`+1 veces que se pasa por la instrucción de la línea 4, `num` veces no se salta, así que si predecimos que no se saltará nuestro pronóstico será acertado casi siempre y entonces casi nunca perdemos los dos ciclos de reloj mencionados. En cambio en el salto de la línea 8 acertaremos la mayor parte del tiempo si predecimos que el salto será tomado, entonces, en lugar de traer y decodificar la instrucción de la línea 9 y siguientes, mejor traemos las de la línea 3 y siguientes.
3. Hacer algo útil mientras se determina si se debe saltar o no. Ejecutar instrucciones que, de todos modos debemos ejecutar sin importar si se salta o no (*delayed branch*).

Forwarding.

Para resolver un hazard de datos como el de la figura 4.4 necesitaríamos que el valor que se obtuvo de la ALU en la etapa EX sea capaz de regresar a la ALU sin tener que esperar hasta que en WB se escriba en los registros.

Cuando la instrucción de la línea 2 lo necesita es en ID, que es la etapa inmediata anterior a EX. Es decir, formalmente hablando el dato necesario para la instrucción de la línea 2 ya está calculado al final de su ciclo ID, porque para ese entonces ya se terminó también el ciclo EX de la instrucción anterior. Entonces todo lo que hay que hacer es retroalimentar a la ALU con su salida, podemos hacer que la salida de la ALU sea obtenida en la subida del reloj y que los operandos de la ALU sean establecidos a la bajada del reloj, así todo está bien. A esta técnica se le llama *forwarding*, *bypassing* o *short circuiting*.

Por supuesto para hacer *forwarding* necesitamos un poco más de hardware. Un dispositivo que sea capaz de notar que se ha producido el hazard de datos y que habilite el que la salida de la ALU regrese a la entrada al mismo tiempo que continúa su camino habitual. Este dispositivo hace parecer que el operando recibido proviene de los registros cuando en realidad es el resultado de la ALU. Además debe ser capaz de retomar el resultado de la ALU luego de la etapa EX o de MEM para hacer posible que no atoren ninguna de las dos instrucciones siguientes.

4.7 Clasificación de hazards de datos

Clasificación de *hazards* de datos.

Los hazards de datos pueden clasificarse de acuerdo con el orden de las escrituras y lecturas de datos que tienen lugar. Supongamos que la instrucción j es posterior a la i:

RAW *Read After Write.* j quiere requiere un dato que aun no ha sido escrito por i. Este es el hazard más usual, pero afortunadamente puede ser resuelto con *forwarding*.

WAW *Write After Write.* j trata de escribir un dato que antes de que haya sido escrito por i. Esto ocurre en pipelines en los que no hay una única etapa de acceso a memoria o si esta etapa es muy lenta y tarda más de un ciclo. En DLX no ocurre, ocurriría y en casos extraños, si se adelantara la etapa WB de algunas operaciones y de hecho sólo se puede adelantar el WB de las aritmético-lógicas (ya que no acceden a memoria se puede omitir la etapa MEM) y además los accesos a memoria tardaran dos ciclos de reloj.

```
lw  R1, (R2)   ; IF ID EX MEM1 MEM2 WB
add R1, R3, R6 ;    IF ID EX   WB
```

En principio estas instrucciones no tienen sentido, pero son válidas.

WAR *Write After Read.* La instrucción j trata de escribir un dato que aún no ha sido leído por i (i leería un dato "demasiado nuevo"). Nuevamente, este tipo de hazard no ocurre en DLX porque se lee en la etapa ID y se escribe en WB que es posterior. Estos hazards ocurren en máquinas en las que algunas instrucciones pueden escribir muy temprano en la ejecución y otras leen muy tarde. Como en general es natural leer antes de escribir resultados, estos hazards son raros.

```
sw  R2, (R1) ; guarda R2 en mem. IF ID EX MEM1 MEM2 WB
add R2, R3, R5 ;                    IF ID EX   WB
```

Si el store guarda en la segunda mitad del ciclo de MEM2 y la suma guarda el resultado en la primera mitad de su WB estamos en problemas.

Hay que notar que RAR *Read After Read* no es un hazard.

4.8 Cuando hay que atorar

Hay situaciones en las que a fuerza hay que atorar el pipeline. Por ejemplo:

```
1   lw   R1, (R2)   ; IF  ID EX MEM WB
2   sub  R4, R1, R5 ;     IF ID EX  MEM WB
3   and  R6, R1, R7 ;        IF ID  EX  MEM WB
4   or   R8, R1, R9 ;           IF  ID  EX  MEM WB
```

2 necesita R1 al principio de su ciclo EX
1 lo obtiene hasta el final de ese mismo ciclo en su MEM
sólo es posible hacer *forwarding* con 3 y 4
la instrucción 2 se debe atorar un ciclo, lo que por supuesto retrasa las 3 y 4 también en un ciclo. Cuando se inserta una burbuja en el paso i de la instrucción j también hay burbuja en el paso $i-1$ de la $j+1$, en el $i-2$

de la $j+2$ y así sucesivamente hasta llegar a una instrucción que empieza a ejecutarse después de la burbuja.

4.9 Desempeño de pipeline revisado

Ya con lo que hemos hecho es posible revisar nuestra estimación del desempeño de un pipeline.

En DLX sin pipeline sale una instrucción cada cinco ciclos de reloj. Con pipeline salen 5 en el mismo intervalo. Es decir, como que

$$Speedup_{pipeline} = etapas$$

Usemos la ley de Amdahl:

$$Speedup_{pipeline} = \frac{Tiempo\ por\ instrucción_{sin}}{Tiempo\ por\ instrucción_{con}} = \frac{CPI_{sin}\ C_{sin}}{CPI_{con}\ C_{con}} \qquad (4.9.2)$$

Donde C es la duración de cada ciclo de reloj[3].

Ahora:

$$CPI_{con} = CPI_{ideal} + ciclos\ atorados = 1 + ciclos\ atorados$$

también idealmente:

$$CPI_{sin} = etapas$$

sustituyendo esto en 4.9.2:

$$Speedup_{pipeline} = \frac{etapas}{1 + ciclos\ atorados} \qquad (4.9.3)$$

Desempeño proporcional al número de etapas.

Así que en efecto, la ganancia en rapidez es proporcional al número de etapas de pipeline. Sin embargo se ha demostrado teóricamente que es bueno tener de 8 a 9 etapas de pipeline[4]. Si una máquina posee demasiadas etapas de pipeline se dice que esta *overpipelined*.

[3] Ya vimos que en realidad los ciclos son un poco más grandes en la máquina con pipeline, pero esto lo despreciaremos por el momento.

[4] Esto porque cuando hay que ejecutar una instrucción de salto nos enteramos si hay que saltar o no en una etapa, en general, avanzada, así que hay que tirar a la basura muchas instrucciones, una por cada etapa previa.

4.10 Hazards de datos

Una instrucción que pasa de la etapa ID a EX se dice que ha sido despachada (*issued*). En DLX todos los hazards de datos se detectan en ID (donde se obtienen los operandos), así que la instrucción es atorada antes de despacharla[5]. También podemos determinar si hace falta hacer *forwarding* (tomar lo que sale de la ALU por adelantado) y enviar las señales de control que lo hacen posible.

Revisemos las posibles situaciones de hazard de datos. Hay dos posibilidades: atorar o hacer *forwarding*. Ejemplifiquemos:

- Atorar.

  ```
  lw  R1, 45(R2)
  add R5, R1, R7
  sub R8, R6, R7
  or  R9, R6, R7
  ```

  R1 se usa en el `add`, esto lo puede notar un comparador (luego veremos con detalle). Atoramos el `add` antes de despachar la instrucción. Desatoramos cuando el `lw` haya hecho WB o, mejor aun, hacemos *forwarding* desde el ciclo MEM y atoramos un ciclo menos.

- *forwarding*

  ```
  lw  R1, 45(R2)
  add R5, R6, R7
  sub R8, R1, R7
  ```

  `sub` usa R1 que es escrito por el `lw` de dos instrucciones anteriores. El mismo comparador mencionado arriba puede detectar esto.

4.11 Hazards de control

En general los hazards de control causan más pérdida de desempeño que los de datos. El problema principal consiste en que la decisión de saltar o no

[5] Por cierto esto lo hace un dispositivo de hardware llamado *pipeline interlock*.

Instr. en ID	Instr. fuente	Reg. fuente	Reg. dest.	Coinciden
ALU R, load, store, branch	ALU R	EX/MEM.	A	EX/MEM.IR(16,20) ID/EX.IR(6,10) `rd, rs1`
ALU R	ALU R	EX/MEM.	B	EX/MEM.IR(16,20) ID/EX.IR(11,15) `rd, rs2`
ALU R, load, store, branch	ALU R	MEM/WB.	A	MEM/WB.IR(16,20) ID/EX.IR(6,10) `rd, rs1`
ALU R	ALU R	MEM/WB.	B	MEM/WB.IR(16,20) ID/EX.IR(11,15) `rd, rs2`
ALU R, load, store, branch	ALU I	EX/MEM.	A	EX/MEM.IR(11,15) ID/EX.IR(6,10) `rs2, rs1`
ALU R	ALU I	EX/MEM.	B	EX/MEM.IR(11,15) ID/EX.IR(11,15) `rs2, rs2`
ALU R, load, store, branch	ALU I	MEM/WB.	A	MEM/WB.IR(11,15) ID/EX.IR(6,10) `rs2, rs1`
ALU R	ALU I	MEM/WB.	B	MEM/WB.IR(11,15) ID/EX.IR(11,15) `rs2, rs2`
ALU R, load, store, branch	load	MEM/WB.	A	MEM/WB.IR(11,15) ID/EX.IR(6,10) `rs2, rs1`
ALU R	load	MEM/WB.	B	MEM/WB.IR(11,15) ID/EX.IR(11,15) `rs2, rs2`

Tabla 4.1: Tabla de solución de hazards mediante *forwarding*. La notación (X,Y) indica los bits de índice X a Y del campo especificado.

se toma con base en alguna comparación que se efectúa en etapas adelantadas del ciclo de ejecución y la dirección de salto es calculada por la ALU, lo que también significa que se obtiene etapas adelantadas de ejecución, cuando ya se tienen los operandos y, lo que es más importante, ya se ha hecho el fetch de las instrucciones posteriores al salto.

Como siempre el método de solución más simple es el mismo para cualquier hazard, atorar el pipeline hasta que se termine de ejecutar la instrucción de salto. Por supuesto no queremos hacer esto.

En una primera aproximación a una solución razonable tendríamos que atorar el pipeline hasta que ya se tenga información de la dirección de salto y si este se va a tomar o no. En DLX esto es hasta que la instrucción de salto termine su ciclo MEM, que es cuando se pasa el resultado de la ALU al multiplexor de PC .

```
instr. de salto (i)  IF  ID  EX  MEM  WB
(i+1)                    IF  bur bur  IF  ID  EX  MEM WB
(i+2)                                    IF  ID  EX  ...
```

En el ciclo WB de la instrucción de salto se procede a cargar la verdadera siguiente instrucción. Pero si no se salta sale sobrando el segundo IF de la instrucción $i + 1$ ya que es la misma que ya se había cargado, se gastan entonces 3 ciclos de reloj.

Tres ciclos de reloj son muchos, en ese intervalo podrían salir tres instrucciones. Dado que la frecuencia de los saltos es de un 30% del total de instrucciones esto es bastante malo. Hay dos cosas que podemos hacer:

1. Saber si el salto será tomado o no lo más temprano posible.
2. Calcular el valor que deberá tener PC (en caso de que el salto sea tomado) lo más pronto posible.

Para lograr un desempeño óptimo deben hacerse ambas cosas.

En DLX los saltos requieren probar si un registro es cero o no. Esto puede hacerse, lo más temprano, en ID. Pero la ALU, que calcula la dirección de salto, es accedida hasta EX. No podemos adelantar más el conocer si un registro es cero, así que lo único que es posible hacer es poner algún dispositivo que pueda calcular en ID la dirección de salto, si hacemos esto

el pipeline sólo tendría que atorarse un ciclo. Pero surge entonces otro problema. Si la instrucción que antecede al salto altera el valor del registro que es probado entonces se incurrirá en un hazard de datos RAW. Que de inmediato pensamos en solucionar haciendo forwarding, pero no podemos porque el valor se obtiene al final del ciclo EX de la instrucción previa al salto y lo necesitamos al principio del ciclo ID de la siguiente, ambas cosas ocurren al mismo tiempo, en el mismo ciclo. Así que tendríamos que viajar hacia atrás en el tiempo. Perdemos un ciclo más en este caso y ni modo.

En otras máquinas el costo por salto (*branch penalty*) es peor, por ejemplo si el ciclo de fetch y el de decodificación tienen un retardo mayor (instrucciones complicadas).

Predicción de saltos.

En SPECint el 13% de los saltos son condicionales hacia adelante, el 3% son condicionales hacia atrás y un 4% son incondicionales[6], en promedio el 67% de los saltos condicionales son tomados así que para reducir la penalización por salto podemos suponer que siempre que se llega a una instrucción de salto este será tomado y en cuanto tengamos acceso a la dirección de salto hagamos fetch de las instrucciones que creemos serán las siguientes. Esto lo podemos hacer si ponemos, como ya habíamos dicho, un sumador que calcule la dirección de salto desde el ciclo de ID de la instrucción de salto. Al mismo tiempo se estaría haciendo el fetch de la instrucción bajo el salto, lo que tiramos a la basura y hacemos fetch del destino de salto.

En general tenemos pues las siguientes opciones:

- No predecimos nada, cargamos la instrucción de abajo del salto, Si este es tomado tiramos a la basura las instrucciones que empezaron a ejecutarse (*freeze* o *flush*).
- Predecimos. Hay varias opciones, las más simples son predecir siempre lo mismo: siempre supongo que será tomado o siempre supongo que no será tomado. En ambos casos si atino pierdo un ciclo de reloj, si me equivoco pierdo dos. Hay opciones más interesantes que revisaremos un poco más adelante.

Predecir no-tomado Hay que tener cuidado de no alterar el estado del CPU hasta saber si el salto será tomado o no (i.e. no se vale hacer WB). Si el pipeline es muy complejo (muchas etapas, enterarse del resultado del

[6]Nuevamente las medidas se tomaron con SPECint en MIPS.

salto muy tarde) entonces se debería tener un mecanismo para revertir el resultado de las instrucciones que no debieron ejecutarse (como el rollback de Transmeta Crusoe). En el caso de DLX podemos enterarnos en ID si será tomado el salto o no, la instrucción siguiente va en fetch apenas así que basta con borrar el registro IF/ID.

Predecir tomado Esta alternativa funciona para arquitecturas en las que la dirección destino del salto se conoce temprano. En DLX no sirve porque no conocemos la dirección de salto antes del resultado del salto, si ya sabemos el resultado no tiene sentido predecir.

Predicción de salto de dos bits Esta es una alternativa usual. Consiste en manejar una máquina de estados finita con cuatro estados: tomado (T), no tomado (NT), tomado con un error (TE) y no tomado con un error (NTE). Predecimos que el salto será tomado, si acertamos permanecemos en T, si no nos pasamos a TE, la próxima vez que pasemos por ese mismo salto predecimos que lo tomaremos otra vez, si acertamos nos pasamos a T pero si no, pasamos al estado NT. Estando en NT predecimos no tomado, si acertamos permanecemos en ese estado, pero si no pasamos a NTE en el que predecimos, también no tomar el salto si acertamos regresamos a NT y si no, pasamos a T. La gráfica del autómata se muestra en la figura 4.5.

Esto permite, en principio hacer predicciones más precisas considerando, de alguna manera la historia de cada salto, la desventaja de este esquema es que hay que guardar en el CPU una tabla hash con una entrada para cada salto que contenga dos bits indicando el estado actual de la máquina de estados finita para ese salto en particular.

4.11.1 Salto retardado

El esquema de salto retardado o *delayed branch* consiste en aprovechar el tiempo mientras se determina si hay que saltar o no, haciendo algo que de todas maneras hay que hacer.

- Salto hacia adelante. Tomamos alguna(s) instrucción(es) que había que hacer antes. Algo como esto:

```
add  R1, R2, R3
```

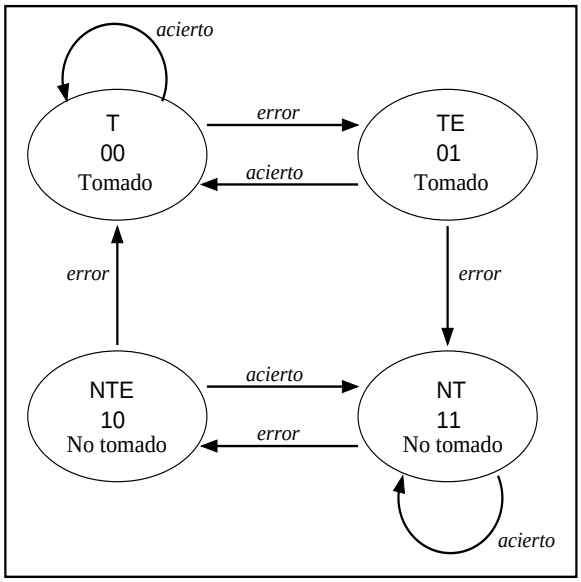

Figura 4.5: Máquina de estados finita para *2-bit branch prediction*

```
beqz R2, salto1
```

lo reacomodamos como:

```
beqz R2, salto1
add  R1, R2, R3
```

El resultado es el mismo y podemos aprovechar los ciclos de reloj que, de otra manera, habría que tirar a la basura probablemente con un flush. El lugar donde se inserta la instrucción se denomina ranura de retardo (*delayed slot*).

- Salto hacia atrás. Tomamos algo que ya hubo que hacer y hay que repetir. Transformamos algo como esto:

```
salto1: add  R4, R5, R6
        ;....
        ;....
        beqz R2, salto1
```

en esto:

```
        add  R4, R5, R6
salto1: ;....
        ;....
        beqz R2, salto1
        add  R4, R5, R6
```

De esta manera la instrucción siguiente del salto (en la ranura de retardo) también es aprovechable (siempre que se salta).

Ranuras de retardo y el compilador.

Evidentemente el más indicado para hacer estas alteraciones del orden del código es el compilador. Es él el indicado para insertar las ranuras de retardo si que se altere la semántica del programa.

CAPÍTULO 5

La jerarquía de memoria y caches

A cualquier programador le agradaría tener una cantidad ilimitada de memoria. Además le agradaría que esta fuera muy rápida, en buena medida la rapidez en la ejecución de un programa está determinada por la velocidad de acceso a la memoria, varios ordenes de magnitud más lenta que el procesador.

Memorias de diferentes velocidades y precios.

Hay diferentes tipos de memoria, algunos muy rápidos, otros muy lentos. A mayor velocidad mayor es también el costo de la memoria. Existen memorias mucho más rápidas que nuestras RAM usuales, pero son mucho más caras, Si pretendiéramos que toda nuestra RAM fuera del tipo rápido el costo sería excesivamente alto.

Analicemos la situación. Hay dos extremos: (a) poner pura memoria lenta, el desempeño del sistema es el peor posible, el costo es el más barato posible. (b) poner pura memoria rápida, el desempeño del sistema es el mejor

posible, el costo es el más alto posible. Es evidente el compromiso entre costo y velocidad, necesitamos lograr un equilibrio razonable, necesitamos encontrar un punto intermedio entre los extremos descritos.

De este problema se percataron los primeros en diseñar computadoras. Goldstine, Burks y Von Neumann escribieron: "Estamos forzados a reconocer la posibilidad de construir un jerarquía de memorias, cada una con una mejor capacidad que le precedente pero más lenta.

Jerarquía de memoria.

La solución fue, desde el principio, definir una jerarquía de memoria. Se colocan varios niveles de memoria. Unos muy rápidos, muy caros y muy cercanos al procesador, otros más lentos, más baratos y más lejanos. El tamaño de cada nivel es proporcional a su lejanía del procesador. Se pone poca memoria rápida y cara y mucha memoria lenta y barata. Cada nivel de memoria es, en términos de contenido, un subconjunto del nivel inmediato superior, más barato y grande. Es decir todos los datos contenidos en el nivel i están también en el nivel $i+1$.

Hay diferentes tecnologías para construir dispositivos de memoria:

- SRAM[1]. Tiene tiempos de acceso de 5 a 25 ns. y un costo de 100 a 250 dolares el Mb.
- DRAM[2]. De 60 a 120 ns. de tiempo de acceso y de 5 a 10 dolares por Mb.
- Disco magnético. De 10 a 20 ms. de tiempo de acceso y de 0.1 a 0.2 dolares el Mb.

Cuando uno va a la biblioteca a hacer un trabajo de algún tema en particular, junta varios libros y los lleva a su mesa. Si se eligen bien los libros es muy probable que con esos baste, pero eventualmente ocurrirá que uno no encuentra algo en esos libros o ha cambiado de tema y tiene que

[1]RAM estática, no requiere estarse refrescando periódicamente como la dinámica, el acceso a este tipo de memoria es más rápido que el acceso a DRAM porque no hay que esperar a que se termine el refresco.

[2]RAM dinámica, requiere de estarse rerfescando periodicamente a una cierta frecuencia propia de la memoria (tiene su propio reloj), esto limita la capacidad de respuesta de la memoria. Cuando el procesador solicíta un dato debe esperar a que termine el refresco y a que el dato sea extraido de la memoria. Para minimizar este periodo de latencia se inventó la memoria SDRAM (*Synchronized DRAM*) en la que la frecuencia de refresco es la misma que la frecuencia de operación del CPU y por tanto este no tiene que esperar tanto, ya que se puede sincronizar con la memoria.

regresar a la estantería a buscar nuevos libros. Algunos necesarios, otros porque se piensa que pueden ser útiles en algún momento. Usualmente los libros del mismo tema están cerca unos de otros en la estantería, así que por cada trabajo que hay que hacer, uno normalmente accede sólo a una pequeña sección de le estantería. Además suele suceder que, luego de que uno ha consultado un libro, lo continua utilizando durante un rato.

Principio de localidad.

A la situación ejemplificada en el párrafo anterior se le denomina *principio de localidad*. Como ya nos percatamos en el ejemplo, hay dos variantes: una vez que se ha accedido a un libro (dato) es muy probable que se accedan también los libros (datos) que se encuentran muy cerca de él (*localidad espacial*); una vez que se ha accedido un libro (dato), es muy probable que en un futuro cercano se vuelva a acceder nuevamente (*localidad temporal*). Al llevarnos los libros a nuestra mesa definimos una jerarquía de memoria. Nuestra mesa es una memoria de acceso rápido y es un subconjunto de la memoria principal, nuestra biblioteca.

La utilidad de la jerarquía de memoria radica en el principio de localidad. Si este no es cierto en ciertas circunstancias, la jerarquía de memoria tendrá un impacto menos favorable o desfavorable sobre el desempeño del sistema. Con una jerarquía de memoria es posible lograr ese buen compromiso que ya mencionamos en la relación costo-beneficio. En principio pueden existir muchos niveles de memoria. En general sólo ponemos 4 o cinco niveles: registros del procesador, cache L1, cache L2, RAM y disco (por cierto en Unix es muy evidente el uso de disco como miembro de la jerarquía, el área de *swap*).

En la jerarquía de memoria, tal como ocurre con la mesa de biblioteca, los niveles de memoria más rápidos, más caros y más cercanos al procesador son un subconjunto de (contienen información que también está contenida en) los niveles más lentos, más lejanos y baratos. Los datos son copiados únicamente entre niveles adyacentes. Las unidades de información mínima que se transfieren entre niveles se denominan *bloques*, el análogo de un libro en nuestro ejemplo.

Acierto (*hit*) y fallo (*miss*).

Cuando se solicita información a un nivel, si el bloque que la contiene está en dicho nivel se dice que ha tenido lugar un acierto (*hit*). Si la información no se encuentra en el nivel al que se solicito se dice qe ha ocurrido un fallo (*miss*). El administrador de este nivel debe solicitarla al nivel inmediato superior, más grande y más lento, hay que ir al estante por el libro solicitado.

Una medida del desempeño de una jerarquía es la tasa de acierto: la

fracción de accesos a memoria que son satisfechos con éxito por el nivel más cercano al procesador.

Debemos definir ahora un par de intervalos promedio de tiempo:

1. Tiempo de acierto (*hit-time*). Tiempo de acceso al nivel de memoria más cercano, mas el tiempo necesario para determinar si es acierto o fallo.

2. Penalización por fallo (*miss penalty*). Tiempo necesario para reemplazar un bloque de memoria de un nivel cercano al procesador por el que es requerido y que está en un nivel más lejano, mas el tiempo necesario para entregar el bloque al procesador.

Los caches de nuestras computadoras actuales están destinados a estar en los niveles más cercanos al CPU. Hay varias preguntas que debemos resolver para diseñar un cache: ¿donde debe colocarse un bloque del nivel de memoria superior en el cache? ¿como se encuentra un bloque de memoria en el cache? en caso de fallo, ¿cuál bloque del cache debe ser reemplazado por el bloque proveniente de memoria? ¿que se hace en caso de una escritura en memoria? ¿donde se escribe?

Hay varias opciones para responder cada pregunta, dependiendo de la respuesta, un cache pertenece a una categoría u otra.

5.1 Colocación de un bloque en el cache

Mapeo directo. Cada bloque de memoria principal puede ponerse en uno y sólo un sitio del cache. Generalmente el bloque de memoria en la dirección *dir* de memoria principal se coloca en el bloque de dirección:

$$dir \pmod{B_{cache}}$$

donde B_{cache} es el tamaño del cache en bloques.

Completamente asociativa. (*fully associative*) el bloque se puede poner en cualquier lugar del cache.

Asociativa de conjunto de n posibilidades. (*n-way set associative*) el bloque de memoria se mapea a un conjunto de bloques del cache, el bloque de memoria puede ponerse en cualquiera de los bloques del conjunto. Generalmente:

$$c = dir \pmod{\#C}$$

c es el número de conjunto del cache, dir la dirección del bloque en memoria principal, $\#C$ el número de conjuntos del cache.

Si hay n bloques en cada conjunto del cache entonces hay n maneras de elegir el bloque dentro del conjunto donde se habrá de colocar el bloque de memoria principal. Es por eso que se llama de n posibilidades: n es el número de bloques en cada conjunto.

Nótese que esta alternativa define una gama de posibilidades entre la primera y la segunda. El mapeo directo es asociativo de conjunto de una posibilidad, la asociatividad completa es asociativa de m posibilidades donde m es el número total de bloques en el cache.

Los caches actuales son generalmente de mapeo directo, asociativos de conjunto de dos y de cuatro posibilidades.

5.2 Localización de bloques en el cache

El tamaño de un cache es mucho menor que el tamaño de la memoria principal, ese es el objetivo, que sea pequeño porque es muy caro. Así que para direccionar un bloque en el cache se requieren de muchos menos bits que para direccionar uno en la memoria principal. Sin embargo el objetivo es que, tenga o no cache la máquina, el acceso a memoria sea transparente para el procesador. Este debe decir la dirección de memoria de la misma manera, independientemente de si tiene o no cache. Asi que se debe definir una manera de traducir direcciones de memoria principal a memoria cache. El encargado de hacer la traducción debe ser un dispositivo de interfaz del cache con el procesador.

Hay que hacer algunas otras cosas. Si un bloque en el cache nunca ha sido leído de memoria o es inválido hay que tener manera de determinarlo marcando el bloque.

Por cierto, dado que los caches son suficientemente pequeños es posible verificar en paralelo todos los bloques del cache para ver si un bloque

dado está o no. Para comprender como podría hacerse esto y como funciona la búsqueda de bloques en general debemos comprender cabalmente la traducción de direcciones.

Imaginemos que tenemos un cache de mapeo directo de 4 Kbytes, un bus de direcciones del procesador de 32 bits y un tamaño de bloque de 4 bytes. El procesador solicita entonces acceso a un dato en memoria diciendo 32 bits de dirección, imaginemos que el dato solicitado es un byte. ¿Cuantos bits necesitamos para...

- direccionar un byte dentro de un bloque? cada bloque tiene cuatro así que necesitamos dos bits.
- direccionar un bloque en el cache? el cache es de 4Kbytes, cada bloque es de 4 bytes, así que el cache tiene 1Kbloque (1024 bloques) y para direccionar un bloque se necesitan 10 bits.

Traducción de direcciones.

Así que de los 32 bits de dirección necesitamos 12 para direccionar en el cache. Los restantes 20 bits sobran... pero es necesario conservarlos de alguna manera, porque juntos los 32 bits direccionan un byte particular, si sólo nos quedamos con los 12 menos significativos hay un total de 2^{20} direcciones que coinciden en esos 12 bits, ¿como los distinguimos?. Lo que se hace es poner esos 20 bits más significativos de la dirección como etiqueta de cada bloque en el cache.

El proceso completo es entonces:

1. El procesador dice una dirección de 32 bits: $dir[0, \ldots, 31]$.
2. El cache usa los bits $dir[2, \ldots, 11]$ como índice del bloque en el cache. Ese bloque tiene asociada una etiqueta (tag) de 20 bits $tag[0, \ldots, 19]$
3. Se comparan $dir[12, \ldots, 31]$ y $tag[0, \ldots, 19]$ si coinciden y ese bloque está marcado como válido el cache entrega el byte indicado por los bits $dir[0, 1]$ al procesador.
4. Si no coinciden $dir[12, \ldots, 31]$ y $tag[0, \ldots, 19]$, o si coinciden pero el bloque está marcado como inválido o vacío, es un fallo. Se solicita a la memoria principal el bloque direccionado por $00 \uplus dir[2, \ldots, 31]$[3].

[3]Nótese que los bits de índices 0 y 1 han sido puestos a cero alineando por bloque. Se transfiere todo el bloque aunque sólo se haya solicitado un byte.

La memoria entrega el bloque al cache, este lo almacena y lo pasa al procesador.

5.3 Reemplazo de bloques en el cache

Imaginemos que se requiere un bloque de memoria y que no se encuentra en el cache y los lugares donde se puede poner están ocupados. Hay que traer el bloque de memoria principal y ponerlo en uno de los lugares que le corresponde quitando el bloque que actualmente este allí. ¿cuál lugar elegimos para poner el que necesitamos eliminar a su ocupante actual (regresarlo a memoria principal)?

- Si es un cache de mapeo directo sólo hay una posibilidad, si el bloque está ocupado, ni modo, debe salir el bloque alojado allí actualmente para colocar el nuevo.
- Si es asociativa de varias posibilidades hay más de una opción:
  - Aleatorio. Elegimos al azar el bloque a reemplazar.
  - El menos recientemente usado (*least recently used* o LRU). Si usamos un bloque hace poco entonces es muy probable que se vuelva a usar pronto, asi que, haciendo uso del principio de localidad, quitamos el bloque del cache, de aquellos donde el bloque de memoria puede residir, que haya sido usado menos recientemente. Por supuesto esto implica que cada bloque en el cache tiene un sello de tiempo *timestamp* con la "hora" de entrada.
  - Usar una cola para sacar el bloque que haya entrado primero.

5.4 Escritura

En general es más común (en promedio tres veces más común) hacer lecturas de memoria que escrituras, así que con nuestros caches tal como los hemos pensado hasta ahora, para lecturas, estamos optimizando el desempeño en el caso más común. Eso es bueno, pero a pesar de eso sería bueno mejorar las escrituras involucrando al cache.

Manejo de escrituras.

Cuando el procesador ordena una escritura en memoria, es muy probable

que en un futuro cercano requiera el mismo dato que escribió, así que, en principio, sería bueno tener el dato en el cache. Pero podemos pensar en dos opciones:

- Escribirlo en el cache y en memoria (*write through*) al mismo tiempo.
- Escribirlo sólo en el cache y regresarlo a memoria sólo cuando sea indispensable por reemplazo (*write back*).

La escritura simultánea en memoria y el cache tiene la ventaja de que la memoria principal siempre está actualizada, siempre es consistente con el cache. Esto tiene ventajas, por ejemplo, en sistemas paralelos con memoria compartida. Pero se incrementa notablemente el tráfico con la memoria principal. Generalmente cada vez que se escribe se debe esperar a que la memoria termine y eso genera retardos (*write stall*). A veces para evitar estos retardos se antecede la memoria de un buffer donde se almacena temporalmente el dato a escribir y de allí es tomado por la memoria principal sin generar retardos para el procesador, este hace de cuenta que ya escribió realmente.

Con el esquema de escritura *write back* se reduce el tráfico con la memoria, múltiples escrituras en el mismo bloque generan uns sola escritura final en memoria principal, mientras que con escritura simultánea cada escritura corresponde a una escritura en memoria principal. Generalmente el bloque modificado en el cache se marca para saber si hay que escribirlo o no cuando sale por reemplazo. Se le pone un bit "sucio" *dirty bit* para indicar que debe ser escrito.

Asi como hay fallos al momento de solicitar una lectura de memoria también podría haberlos al momento de solicitar una escritura. En esos casos hay nuevamente dos opciones:

- *write allocate* o *fetch on write*, se trae en bloque de memoria al cache y luego se escribe en el, ya sea con escritura simultánea o después, con *write back*.
- Se escribe directamente en la memoria, este esquema se llama simplemente *no-write allocate*.

Podría parecer que no es muy conveniente el primero de estos esquemas usando escritura simultánea, pero lo és gracias al principio de localidad, es

muy probable que el bloque se se acaba de escribir sea utilizado en un futuro cercano, así que lo traemos al cache previniendo futuros accesos.

5.5 Desempeño de caches

Podemos preguntarnos ahora ¿como organizamos el cache de una máquina?. Hay dos cosas que la computadora debe acceder de la memoria, código y datos. Así que podemos pensar en dos alternativas diferentes: (1) poner un solo cache en donde se almacenen tanto datos como código y (2) poner dos caches (o un cache dividido en dos secciones, formalmente hablando) uno para datos y otro para código. Como siempre, para saber que alternativa es mejor debemos hacer cuentas. En el libro de Hennessy (figura 5.7, página 384) se muestra una tabla con las tasas de fallo de diferentes tamaños de cache, tanto divididos como unificados. Lo que se observa es que la tasa de fallo para el cache de datos siempre es mayor que la tasa del unificado y la del cache de instrucciones siempre es menor. Por ejemplo para un cache de 64 Kb. la tasa de fallo para instrucciones es de 0.15% de los accesos, la tasa de fallo para datos es de 3.77% y la del unificado (de 128 Kb.) es de 0.95%. Dado que son tasas de fallo podemos pensar en sacar el promedio de tasa de fallos para datos e instrucciones y comparar con la tasa del unificado, si hacemos esto obtenemos un promedio de tasa fallo de 1.96% para el cache dividido (64Kb. datos + 64Kb. instrucciones), contra el 0.95% de tasa de fallo del unificado (128Kb.), por lo que pensaríamos que es mejor tener un cache unificado.

Evaluación correcta del desempeño.

Pero en realidad sólo tenemos una visión parcial de las cosas, medir el desempeño de caches considerando únicamente la tasa de fallo es un error. Estamos perdiendo de vista las situaciones en las que el cache es realmente útil, cuando no hay fallo. Para evaluar correctamente el desempeño de los caches debemos abrir nuestro campo visual. Usaremos el tiempo promedio de acceso a memoria como medida del desempeño:

$$\overline{T}_{acc} = T_{acierto} + T_{fallo} R_{fallo}$$

En esta expresión $\overline{T}_{acc}$ es el tiempo promedio de acceso, $T_{acierto}$ es el tiempo de acceso a la memoria cuando lo que se solicita está en el cache, T_{fallo} es el tiempo de acceso a la memoria cuando lo que se solicita no está en el cache y R_{fallo} es el porcentaje, o proporción, o tasa, de fallo (número de veces que el acceso genera un fallo entre el número total de accesos).

Con esto podemos proceder a evaluar cual alternativa es mejor, si un cache unificado o uno de datos y otro de código. Supongamos que en caso de acierto el tiempo de acceso a memoria es de 1 ciclo de reloj y en caso de fallo es de 50 ciclos. En una máquina con pipelining, en un solo ciclo de reloj se pretendería acceder dos veces al cache unificado, porque la etapa de MEM de una instrucción ocurriría al mismo tiempo que el IF de otra. Esto, como habíamos dicho es un hazard estructural y hace que se tenga que atorar el pipeline un ciclo, así que, en caso de acierto en un cache unificado, hay un retardo extra de un ciclo de reloj en el acceso a datos, porque siempre que se accede a un dato al mismo tiempo se pretende hacer el acceso por la siguiente instrucción.

También debemos tener en consideración, en al caso de un cache dividido, cuanto se usa cada sección, cuantos accesos a memoria son causados por acceso a instrucciones y cuantos por acceso a datos. Según medidas experimentales, un 75% de los accesos a memoria son por instrucciones y el 25% son por datos (SPEC). Así que nuestras tasas de fallo serán:

$$R_{fallo}^{dividido} = (0.75 \cdot 0.0015) + (0.25 \cdot 0.0377) = 0.01055 \ fallos/acceso \quad (5.5.1)$$

Esta es una medida más fina que el promedio que usamos arriba, se consideran los pesos reales de cada tipo de acceso (.75 para instrucciones y .25 para datos) que es más verosímil que usar el mismo peso para ambos tipos, como hicimos arriba.

por otra parte, según la tabla de Hennessy:

$$R_{fallo}^{unificado} = 0.0095 \ fallos/acceso \quad (5.5.2)$$

A pesar de que 5.5.1 es una medida más fina que el promedio burdo usado antes sigue ocurriendo que parece mejor un cache unificado. Pero las cosas cambian si usamos el tiempo promedio de acceso:

$$\overline{T}_{acc}^{unificado} = 0.75(1+0.0095 \cdot 50)+0.25(1+1+0.0095 \cdot 50) = 1.725 \ ciclos/acceso \quad (5.5.3)$$

Por otra parte:

$$\overline{T}_{acc}^{dividido} = 0.75(1+0.0002 \cdot 50)+0.25(1+0.0288 \cdot 50) = 1.3675 \ ciclos/acceso \quad (5.5.4)$$

Por supuesto, con la visión global que nos da el medir el tiempo esperado total de acceso a memoria, obtenemos el intuitivo resultado de que es mejor tener un cache dividido en vez de uno unificado que se vuelva un cuello de botella en una máquina con pipeline y, lo que es determinante, que tiene una tasa de fallo superior al promedio pesado de las tasas de cada división (sin pipeline, el tiempo promedio de acceso es 1.475 en el cache unificado, que sigue siendo mayor que el del dividido).

Así pues la medida correcta para evaluar el desempeño de un cache es el tiempo promedio de acceso a memoria. No sólo evaluamos lo que perdemos si las cosas salen mal, sino que evaluamos también cuanto ganamos si las cosas salen bien.

5.6 Mejoras al desempeño

Por lo que hemos visto el desempeño de un cache puede ser mejorado por los siguientes medios:

1. Reduciendo el número (proporción) de fallos
2. Reduciendo la penalización por fallo
3. Reduciendo el tiempo de acceso en caso de acierto

Para reducir la penalización por fallo lo que se suele hacer desde hace unos pocos años (a partir de 1995 se ha ido haciendo más usual) es poner dos niveles de cache. En el esquema con un solo cache, la penalización por fallo es grande dado que hay que solicitar la información directamente a la memoria principal, mucho más lenta que la memoria cache, pero si se coloca otro cache, uno de segundo nivel (L2), es este el que, con muy alta probabilidad, debe responder en caso de que el dato solicitado no se encuentre en el cache de primer nivel (L1). El cache de segundo nivel no será tan rápido como el de primer nivel, pero si es mucho más rápido que la memoria principal. Actualmente los caches de primer nivel se colocan en la misma pastilla del procesador, en la misma estampa, por lo que se les llama internos. Los de segundo nivel son externos.

Otra estrategia para reducir la penalización por fallo consiste en hacer lo indispensable para responder y luego terminar. Cuando se requiere un dato

de memoria, si es más barato traer sólo el dato requerido en vez de todo el bloque que lo contiene entonces se trae sólo el dato, se entrega y luego se termina la transferencia del bloque (*Critical Word First*).

Para reducir el tiempo de acceso en caso de un acierto se han hecho dos cosas: (1) hacer caches simples y pequeños y (2) evitar la traducción de direcciones. Esta segunda alternativa se denomina cache virtual y consiste en que el cache se hace pasar por la memoria principal sin traducir el espacio de direcciones de aquella al suyo propio. Básicamente el cache renombra sus direcciones para que correspondan a las de memoria y la desventaja de hacer esto es que al cambiar de proceso, en un sistema capaz de ejecutar varios de ellos concurrentemente (*process switch*), hay que renombrar otra vez el espacio de direcciones completo y por tanto hay que vaciar todo el cache para asegurar consistencia con la memoria principal (*cache flush*).

Para atacar el primero de los rubros mencionados, reducir la tasa de fallos, hay que lograr un delicado equilibrio entre el tamaño de la memoria, el tamaño del cache y el grado de asociatividad de este. En general la memoria principal es mucho más grande que el cache, tres ordenes de magnitud mayor, aproximadamente. Si el cache tiene un grado alto de asociatividad (muchos bloques en cada conjunto), entonces tendrá menos conjuntos. Esto significa que cada conjunto debe dar servicio, en promedio, a grandes áreas de memoria principal, es decir, grandes áreas de memoria se mapean al mismo conjunto del cache. Esto hace que tienda a saturarse un bloque del cache mientras que muchos otros están desaprovechados, la saturación de un bloque hace que se incremente la probabilidad de tener que reemplazar bloques todavía en uso, por nuevos bloques que luego hay que reemplazar por los originales otra vez.

Regla 2:1 de caches.

Si el grado de asociatividad es muy bajo, entonces hay pocas opciones para colocar un bloque de memoria y es fácil que se llegue al conflicto de tener que reemplazar un bloque útil por uno nuevo y, al igual que en el caso anterior, tener que volverlo a reemplazar pronto. En general se ha observado que la tasa de fallo en un cache de mapeo directo de tamaño N es casi la misma que en un cache asociativo de conjunto de 2 opciones de tamaño $N/2$. Esto se conoce como *la regla 2:1 de caches.*

Para tratar de reducir estos problemas de reemplazos poco deseables pero necesarios, una alternativa es hacer un *cache de víctimas*, un área de memoria en la que se colocan los bloques que salen del cache por reemplazo, esta cantidad de memoria es de acceso rápido, no tanto como el cache, pero

más rápida que la memoria principal, la idea es reducir la penalización por fallos debidos a reemplazos prematuros.

Cache exclusivo.

Hemos dicho que en la jerarquía de memoria cada nivel es un subconjunto del nivel inmediato superior, más grande, más barato y más lento. Así es porque la intención es que uno pueda acceder a los mismos datos que están en memoria pero más rápido. Sin embargo se esta ensayando la alternativa en la que en una máquina con dos niveles de cache, el cache L1 no es un subconjunto del L2. A este tipo de cache se le ha denominado exclusivo y es utilizado en el AMD *Thunderbird*, versión mejorada del *Athlon*. Aparentemente este cache exclusivo es punto clave (junto con el uso de cobre en vez de aluminio para las pistas) en la mejora de desempeño que ha puesto al Thunderbird ligeramente por encima de su rival, el *Copermine* de Intel, versión mejorada del Pentium III.

La ventaja evidente del cache exclusivo es que la totalidad del cache L2 es usada para almacenar datos que, por definición, no se encuentran en el de L1. En una situación normal, el cache L2 tiene una parte de su tamaño total ocupada por los mismos datos que contiene L1.

CAPÍTULO 6

Entrada/Salida y canales

6.1 Canales (*buses*)

Por supuesto debe haber un medio de comunicación entre el CPU y los dispositivos periféricos del sistema. La memoria y los dispositivos de entrada/salida, deben tener la posibilidad de enviar y recibir datos hacia y desde el CPU, el medio de comunicación está constituido, esencialmente, por un conjunto de cables al que todos los dispositivos y el CPU se conectan. A este medio compartido de comunicación se le llama canal (*bus*).

Ancho de banda.

El hecho de que todos los dispositivos compartan un solo canal de comunicación, convierte a este en un cuello de botella potencial, porque es compartido y porque posee una cierta capacidad de comunicación finita. Es como una via rápida con límite de velocidad estricto, de hecho imposible de violar. A la capacidad del canal se le llama *ancho de banda*, las unidades

en que se mide son unidades de información sobre unidades de tiempo (p.ej. Mbytes/segundo).

Generalmente los canales de comunicación en las computadoras modernas se dividen en dos categorías: canales de CPU-memoria y canales de entrada/salida. La diferencia es en los anchos de banda y la disponibilidad. Los canales que conectan el CPU con la memoria, y nada más, deben ser lo más rápidos posible dado que la memoria es el recurso más importante, este canal no es compartido con ningún otro dispositivo, siempre está completamente disponible para la memoria, es lo que se denomina un canal dedicado. Los canales de entrada/salida conectan el CPU con el resto de los dispositivos: puertos de comunicaciones seriales y paralelos, dispositivos de entrada (teclados, ratón, adaptadores de red, etc.) y dispositivos de salida (adaptadores de video, de sonido, de red, etc.). Normalmente estos dispositivos comparten un único canal al que todos se conectan y que todos "escuchan"[1].

Por supuesto el canal se utiliza para transferir información que se ha de escribir o que se ha leído de alguno de los dispositivos conectados a él, así que las operaciones elementales que debe poder efectuar cualquier dispositivo sobre el bus son también lecturas y escrituras de datos en él. Una secuencia de lecturas y escrituras en las que dos dispositivos se comunican para lograr un propósito particular se llama una *transacción*. Cuando el CPU solicita al controlador de disco una lectura envía, a través del bus, la localización de los datos a leer, el controlador obtiene los datos del disco y los escribe en el bus, de donde son leídos por el CPU o la memoria, esto es un ejemplo de transacción.

Bus masters y árbitros.

Los dispositivos conectados al canal y que tienen la capacidad de iniciar transacciones se denominan *bus masters*. El CPU es, por supuesto un *bus master*. Si hay muchos *bus masters* puede ocurrir que más de uno trate de iniciar una transacción en un instante de tiempo, así que debe haber algún mecanismo para legislar sobre el canal. Normalmente esta tarea la realiza un dispositivo especial llamado *árbitro del canal* que decide quien sigue. El mecanismo para decidir esto es una cola con prioridades: en principio el canal se le concede a quien primero lo pidió, a menos que haya alguien que lo requiera con urgencia. Normalmente los dispositivos electromecánicos de entrada tienen prioridad porque no pueden detenerse (una vez que las cabezas del disco están posicionadas el disco gira leyendo y si no tiene a

[1] Hay canales que no son así, por ejemplo el usado por Alpha y AMD (Athlon) es punto a punto.

donde enviar los datos se pierden).

Algunos canales fragmentan las transacciones de tal forma que, si el dispositivo al que se piden datos es lento, mientras se obtienen los datos se procesa otra transacción, de tal forma que no se tiene que esperar hasta que la primera transacción termine para iniciar la segunda. Se podría pensar, por ejemplo en enviar una petición al controlador de disco. Posicionar las cabezas y obtener la información es, comparativamente hablando, muy lento respecto a la frecuencia de operación del procesador, así que mientras se puede enviar un paquete de datos al adaptador de red, por ejemplo.

Las cosas que deben viajar por el canal son, en general, datos y direcciones. Se puede pensar en tener el canal divido en dos, unas líneas para transmitir sólo datos y otras para transmitir sólo direcciones, esto por supuesto hace que el canal sea más costoso, pero mejora el rendimiento.

6.1.1 Sincronización del canal

Canales síncronos y asíncronos.

Otra de las características sobresalientes que influyen en el desempeño es la sincronización del canal. Esto significa esencialmente, determinar cuando puede ser iniciada una transacción. Si se puede iniciar en cualquier momento, aún cuando algunos de los diversos dispositivos conectados no estén listos para responder y cada uno de ellos opere a frecuencias diferentes, se habla de un canal asíncrono. Por supuesto la otra alternativa es que todos los dispositivos operen a la misma frecuencia y entonces, si el canal es sincronizado a la misma frecuencia, se segura que siempre que un dispositivo esté listo lo estarán el canal y el dispositivo receptor de la transacción, en este caso se dice que el canal es síncrono. Si el canal debe ser muy largo, la propagación de los pulsos de reloj tiene retardos significativos a lo largo del canal y de las líneas que conectan el reloj con los dispositivos, así que en estos casos es mejor usar un canal asíncrono. En general tenemos el panorama mostrado en la tabla 6.1

Protocolo de sincronización (*handshake*).

Cuando el canal es síncrono, son los dispositivos los que deben lograr la sincronización entre ellos para asegurar que los datos transmitidos no se pierdan. Esto significa que deben ejecutar un protocolo de sincronización o de *handshake*. Un protocolo típico de este tipo es algo como esto: supongamos que un dispositivo A le solicita a otro B la transferencia de datos indicándole la localización de los mismos, es decir la dirección[2] donde se

[2] Aquí el término dirección se usa en un sentido más amplio que el usual en el que se

Característica	**Mejor desempeño**	**Menor costo**
Ancho del canal	datos+direcciones	unificado (multiplexado)
bus masters	múltiples	único
Transacciones	fragmentadas *petición-respuesta*	conexión continua
Sincronización	síncrono	asíncrono

Tabla 6.1: Alternativas de diseño de canales.

encuentran. Se procede como sigue:

1. A levanta una señal de petición de datos DataReq en las líneas de control del canal y coloca la dirección de los datos requeridos en las lineas de direcciones.
2. B se dá cuenta de que le es solicitada información al ver la señal DataReq para él y lee la dirección de las líneas de direcciones. Levanta entonces una señal de reconocimiento Ack.
3. A ve el Ack y libera la señal de petición y las lineas de direcciones.
4. B baja la señal Ack y busca los datos solicitados (de hecho esto lo empieza a hacer desde el paso 2).
5. B coloca los datos leídos en las lineas de datos del canal y envía una señal DataRdy a A.
6. A ve la señal DataRdy, lee los datos del canal y levanta Ack otra vez.
7. B se percata del Ack, baja la señal DataRdy y libera las lineas de datos.
8. A ve abajo DataRdy y baja Ack lo que completa la transacción.

Es la ejecución de este protocolo lo que hace más lentos los canales asíncronos.

En las computadoras de hace algunas décadas y en muchas de las estaciones de trabajo actuales se utilizan dispositivos periféricos fabricados por la misma compañía que produce la máquina, la posibilidad de conectarles dispositivos de e/s de otro fabricante es muy limitada (p.ej. en general uno siempre se encuentra con una Sun con discos, unidades de cinta, monitores, tarjetas de red, teclado, ratón (incluyendo el tapete), micrófono y bocinas

refiere a direcciones de memoria. Abusando del término, dirección significa cualquier cosa que permita localizar datos en el dispositivo

Sun) de esta manera la compañía se asegura de hacer todo perfectamente compatible, todo perfectamente sincronizable. En un entorno como este es fácil pensar en hacer canales síncronos, lo que no es sencillo de lograr si la arquitectura permite la conexión de una gran variedad de dispositivos de diferentes fabricantes, algo frecuente en el mundo de las computadoras personales sobre todo hace algunos años.

6.1.2 Otra clasificación

Los canales también pueden caracterizarse por su uso. En esta clasificación las categorías son las siguientes:

- Canales memoria-CPU. El ancho de banda del canal es igual al rendimiento (*throughput*) de la memoria. Ningun otro dispositivo se conecta al canal. El ancho del canal es también igual al tamaño de bloque entregado por la memoria.
- Canal de panel trasero. Se llaman asi porque tradicionalmente los conectores del canal estaban dispuestos en un panel trasero del chasís de la computadora. Pretenden ser un punto intermedio entre los canales de memoria y los de entrada/salida, poseen un ancho de banda suficientemente alto como para conectar en ellos la memoria sin que el desempeño sufra demasiado y al mismo tiempo permiten la coexistencia de el resto de los dispositivos compartiendo el canal. En general requieren de lógica adicional para admitir las conexiones de los distintos dispositivos.
- Canales de entrada/salida. Los canales anteriores son, en general, cortos, todos los dispositivos deben estar físicamente cerca, en los canales de entrada/salida esto no necesariemente es así, pueden ser bastante largos y casi no requieren de lógica adicional para la conexión de los diferentes dispositivos.

6.1.3 Esquemas de arbitraje

Cuando hay múltiples *bus masters* debe haber un árbitro que regule el acceso al canal. Hay diferentes esquemas para definir qué dispositivo accede al canal. En general hay dispositivos que tienen mayor prioridad de acceso, como ya hemos dicho.

- Arbitraje *Daisy chain*. Por el canal fluye, en una sola dirección, una señal de permiso de acceso, los dispositivos se conectan al canal en orden decreciente de prioridad, cuando un dispositivo ve la señal de permiso pasando frente a él, si tiene transacciones pendientes, atrapa la señal e inicia su transacción. Por supuesto esto hace que los dispositivos de prioridad más alta puedan hablar antes que los de prioridad más baja, dado que ven la señal antes que estos últimos. Lo malo es que si algun dispositivo deja de funcionar correctamente y se queda con la señal permanentemente, el canal completo se vuelve inútil.

- Centralizado. Hay un árbitro que recibe, ya sea por líneas de control específicas o por líneas del canal, peticiones de acceso de los diferentes dispositivos. El árbitro sabe cuáles dispositivos tienen mayor prioridad y responde especificamente al dispositivo seleccionado, que puede iniciar su transacción. El problema es que el árbitro puede volverse un cuello de botella. PCI es un canal centralizado.

- Arbitraje distribuido por autoselección. Cada dispositivo sabe su propia prioridad y se dá cuenta de las prioridades del resto de los dispositivos que hayan solicitado acceso simultáneo al canal, si es el de mayor prioridad sabe que tiene derecho a iniciar su transacción, si no, sabe que debe esperar. SCSI y el canal de las Apple Macintosh II son de autoselección.

- Arbitraje distribuido por detección de colisión. Es como el CSMA/CD de las redes Ethernet. Si un dispositivo requiere iniciar una transacción lo intenta, si hay más de un intento de acceso al canal todos se percatan de ello y aquellos que comenzaron sus transacciones al mismo tiempo esperan un tiempo antesde volver a intentar.

Hoy en día hay muchos y muy variados canales, algunos de ellos se han convertido en un estándar en la industria, primero en un estándar de facto y más tarde en estándar formal, por ejemplo SCSI. Algunos de los estándares más famosos han surgido del mundo de las computadoras personales más usuales, las de procesador de Intel 80X86.

Canales de PC.

El primer canal de una PC de IBM fue el llamado S-100 (100 lineas), 2.3 MHz de frecuencia, 8 bits de capacidad. Más tarde los 8 bits se convirtieron en 16 y en 1987 el IEEE promovió el diseño del bus como un estándar llamado ISA (*Industry Standard Architecture*). ISA era un canal de 16 bits a una frecuencia de 8 MHz, una tasa de transferencia de 8Mbytes por segundo. El

canal ISA fue usado por la IBM PC AT. A ISA se conectaban el procesador, los periféricos y la memoria inclusive, asi que resultaba bastante lento para nuestros estándares de hoy día. Cada dispositivo conectado a bus lo hace a través de un controlador, de tal forma que hay muchos *bus masters* (cada controlador lo és). Para acceder a la memoria todos debían pasar por procesador, algo que disminuye el desempeño notablemente, dado que el procesador debe mediar siempre entre el controlador del dispositivo que requiere la memoria y esta última.

Como a pesar de todo el bus era lento, IBM decidió diseñar un nuevo canal mucho más rápido, el resultado fué la arquitectura de microcanal (MCA o *MicroChannel Architecture*) bastante más rápida, mucho más costosa e incompatible con ISA. Normalmente los dueños de computadoras desean poder actualizar estas y poder reutilizar: el software (de allí la eterna compatibilidad hacia atrás de Intel) y los periféricos. El hecho de no poder reutilizar los periféricos hizo poco rentable usar máquinas con MCA y el resultado fue que IBM acabó haciendo máquinas que no eran *IBM compatibles*. MCA surgió en 1987 y fue usado sólo en las PS/2 de IBM, transfería 32 bits de una sola vez y su tasa de transferencia era de unos 40 Mb por segundo, era un canal asíncrono.

Hoy en día nuestros canales más usuales son PCI (*Peripheral Component Interface*), creado en 1992. PCI opera a 33MHz transfiere 32 bits de una sola vez y su máxima tasa de transferencia es de 132 Mbps, es un canal síncrono, *backplane*, con direcciones y datos multiplexados (ambas cosas por las mismas líneas), admite múltiples *bus masters*, de arbitraje centralizado y admite la conexión de hasta 32 dispositivos por segmento, aunque es posible conectar más de un segmento y colgar hasta 1024 dispositivos. Actualmente se trabaja (Compaq o mejor dicho HP) en una extensión de 64 bits para PCI llamada PCI-X que será capaz de transferir hasta 1Gbps con transferencias de 64 bits de una sola vez. La especificación actual más reciente es PCI 2.2, también de 64 bits a 66 MHz y con transferencia máxima de 533 Mbps. Detrás de PCI hay un organismo, el grupo de interés especial en PCI (*PCISig*) que se encarga de formular los estándares de PCI.

Otro canal famoso es el SCSI, del que hablamos en el contexto de discos. SCSI es un canal de entrada/salida, datos y direcciones se multiplexan, admite múltiples *bus masters*, es de arbitraje distribuido de autoselección, hay versiones síncronas y asíncronas de SCSI, también hay distintas versiones con diferentes límites de conexión: 7, 15 y 31 dispositivos, por ejemplo; la longitud puede ser de hasta 25 metros.

Otros buses exitosos usados hoy en día son los punto a punto usados por Alpha (21264) y el Athlon de AMD, el bus es el EV6 de 100 y 200 MHz nominales y 200 MHz y 400 MHz efectivos, 32 bits.

6.2 Discos

El dispositivo de almacenamiento por excelencia es, hasta hoy, el disco magnético fijo, al que comúnmente llamamos disco duro.

Estructura física de un disco duro.

Un disco duro consiste de una pila de uno o más platos[3] de aluminio recubiertos de oxido ferroso para hacerlos susceptibles a la magnetización. Los platos miden desde 1.2 hasta unas 5 pulgadas, lo más popular, por el momento, parecen ser los de 3 1/2 pulgadas, el diámetro de un disco flexible, en las *nootebooks* son usuales los de 1.2. La capacidad de almacenamiento, actualmente, va desde 4.3 Gb. hasta alrededor de 80 Gb[4]. Las labores de escritura y lectura en el disco están a cargo de un conjunto de cabezas capaces de desplazarse en una linea recta radial.

Para escribir en el disco la cabeza recibe corriente del exterior, positiva o negativa dependiendo del bit a escribir, esta corriente induce un campo magnético que alinea las moléculas del disco hacia un lado u otro (0 o 1, izquierda o derecha). Para leer la operación es inversa, el campo magnético de la superficie del disco induce una corriente en la cabeza que se traduce en 0 o 1 dependiendo del signo de la corriente inducida. La cabeza lectora/escritora no toca la superficie del disco[5] y lee o escribe la secuencia de bits conforme el plato gira. Las velocidades de giro actuales más comunes son de 5400 RPM o 7200 RPM, pero hay discos de 10,000 RPM SCSI y seguramente habrá algunos más veloces.

En general la densidad de almacenamiento de los discos magnéticos, durante los 1990 se triplico cada dos años (al menos) y el costo ha descendido de 100 USD el megabyte en 1983 a un rango de 6 a 9 USD el gigabyte actualmente (discos IDE).

Los datos en el disco se organizan en pistas (*tracks*) concéntricas, cada pista es dividida en tramos llamados sectores y el conjunto de sectores en

[3]Algunos llegan hasta 12 platos, p.ej. el Seagate Cheetah 18LP SCSI de 36Gb.

[4]El Quantum Atlas 10K II es de 73Gb (SCSI) y el IBM DTLA307075 es de 76 Gb (IDE)

[5]En los discos flexibles si la toca.

la misma posición en todos los platos se denomina cilindro. La información de cada sector va precedida de un preámbulo (generalmente una secuencia de bits con valores alternados) para sincronizar la cabeza lectora, luego el bloque de datos (512 bytes es un tamaño usual) y luego un código corrector de errores (ECC, generalmente Reed-Solomon). Para separar cada sector se deja entre ellos un pequeño espacio vacío (*gap*). Generalmente la capacidad reportada de los discos ya excluyen toda la que se utiliza en preámbulos, corrección de errores y *gaps*, en lo que se invierte un 15% de la capacidad absoluta.

Seek time y latencia rotacional.

El desempeño de los discos depende de cuanto tiempo tome colocar las cabezas en posición, y cuanto tiempo tome que los datos pasen por debajo de la cabeza. Así que hay dos periodos de tiempo a medir:

- El tiempo usado para poner las cabezas a la distancia adecuada del centro para leer/escribir la pista que se requiere. A esto se le llama *seek time*.
- El tiempo que hay que esperar para que el sector requerido pase debajo de la cabeza. A este se le llama *latencia rotacional*.

Además hay que considerar el tiempo requerido para transferir los datos de la cabeza al canal de donde serán leídos (lo que involucra su verificación de integridad), pero ciertamente lo más tardado es la operación mecánica del disco, los dos factores mencionados arriba son los que dominan el tiempo de acceso.

Densidad variable.

El lector observador se estará preguntando "y el disco ¿tiene más sectores por pista mientras más lejos del centro esté la pista?". Dado que un circulo de mayor radio tiene mayor circunferencia que uno de radio menor, esta pregunta es lógica y representa un problema, dado que los discos giran a una velocidad angular constante (evidente dado que se usan RPM para medir la velocidad). En principio cabrían más sectores en la pista más exterior que en la más interior, así que hay que decidir que hacer. Lo más sencillo resulta ver cuantos sectores caben en la más interior de las pistas y grabar siempre a esa densidad. Esto hace que las pistas más externas se desperdicien mucho, pero bueno, esa fue la solución adoptada por todos hasta hace poco. Actualmente, para minimizar el desperdicio se dividen las pistas en categorías (de 10 a 30 categorías diferentes) cada categoría caracterizada por el número de sectores que caben en la pista más interior del conjunto, luego todas las pistas en la categoría se graban a distinta densidad variando a saltos entre categorías.

Asi que resulta mejor para el desempeño tener discos de mayor velocidad de giro y de menor tiempo de posicionamiento de cabezas. La medida de desempeño debe ser entonces el tiempo promedio requerido para leer o escribir un sector en el disco. Por ejemplo: con un disco Seagate Barracuda IDE de 7200 RPM con un tiempo promedio de posicionamiento de cabezas de 8.2 ms. una capacidad de transferencia de 45.5 Mb/seg y tiempo de transferencia (*overhead*) de 1 ms.[6] obtenemos:

$$7200\ RPM = 120\ Rev/seg = 0.12\ Rev/ms \tag{6.2.1}$$

Si, como es usual, cada sector tiene 512 bytes (0.5 Kb) entonces, cada vez que se lee un sector, como se transfiere a una tasa de 45.5 Mb/seg = 46592 Kb/seg = 46.592 Kb/ms, el tiempo requerido para transferirlo es:

$$\frac{0.5\ Kb}{46.592\ Kb/ms} = 0.0107\ ms \tag{6.2.2}$$

dada la posición de las cabezas y la del sector a ser leído, habrá veces que el sector tendrá que dar toda una vuelta para llegar bajo la cabeza, habrá veces que no tenga que girar nada, en promedio tendrá que dar media vuelta. Así que el tiempo promedio de latencia rotacional, usando el resultado obtenido en 6.2.1 es[7]:

$$\frac{0.5\ Rev}{0.12\ Rev/ms} = 4.166\ ms \tag{6.2.3}$$

Ahora, usando los resultados de 6.2.2 y 6.2.3 y añadiendo el tiempo de posicionamiento promedio y el *overhead* obtenemos el tiempo promedio de acceso:

$$\overline{T}_{disco} = 8.2\ ms + 0.0107\ ms + 4.166\ ms + 1\ ms = 13.377\ ms \tag{6.2.4}$$

Esta es una estimación del promedio, el *seek time* promedio que reportan los fabricantes del disco es el promedio sobre todas las posibles posiciones de las cabezas, no es un promedio medido sobre el uso real del disco, este es de un 25 a un 35 % del *seek time* promedio dado por el fabricante, dependiendo del que tanto del disco haya sido usado y de la fragmentación de la información en el mismo (aquí el sistema operativo tiene un papel importante en el

[6]Este valor es el único inventado en este ejemplo, pero no es descabellado, generalmente esta entre 1 y 2 ms.

[7]Por cierto el valor obtenido coincide con el reportado por Seagate.

desempeño). Si usáramos el 30 % de los 8.2 ms (es decir 5.74 ms) dados por Seagate tendríamos:

$$\overline{T}_{disco} = 5.74\ ms + 0.0107\ ms + 4.166\ ms + 1\ ms = 10.9167\ ms \quad (6.2.5)$$

Recalibración térmica.

Ahora es evidente que, mientras más rápido gire un disco, mejor será su desempeño porque disminuye su latencia rotacional. Sin embargo, dado que los discos giran muy rápido hay fricción, esto significa calor, así que los discos tienden a calentarse y como son de metal (aluminio, que además es buen conductor de calor) se dilatan, ol que hace que las pistas del disco, luego de un tiempo, ya no están donde deberían estar, sino un poco más afuera (además la diferencia no es constante, es una función lineal del radio de la pista). Así que los discos duros deben autorecalibrarse con frecuencia, lo que hace que, de pronto algunos accesos del disco se retarden inusualmente. Para recalibrarse los discos avanzan sus cabezas a lo largo de todo su desplazamiento de ida y vuelta. Para evitar este inconveniente hay discos que se hacen de vidrio en vez de aluminio, como el vidrio es muy frágil se recubre de cerámica, ninguno de los dos materiales se dilata.

IDE Vs. SCSI.

Todos los discos funcionan igual, sin embargo se clasifican en dos categorías dependiendo de la interfaz del disco con el bus: IDE (*Integrated Drive Electronics*), también llamado ATA (*Attachment*) y SCSI (*Small Computer Systems Interface*). Generalmente los discos SCSI tienen tiempos de posicionamiento menor y pueden girar a velocidades mayores, esto no se debe a limitaciones mecánicas de los IDE sino a que estos, o mejor dicho sus controladores, requieren de la intervención del CPU del sistema mientras que los controladores SCSI son independientes y están, por entero dedicados al disco. Un disco IDE no puede entregar tan rápido porque su controlador es más simple. Los controladores SCSI son además suficientemente inteligentes como para manejar eficientemente solicitudes diversas, encolarlas y procesarlas con traslape de tiempo para aprovechar al máximo las posiciones relativas de los diversos accesos y no perder tanto tiempo de *seek*. Además SCSI contempla la posibilidad de tener muchos dispositivos conectados en cascada en la misma interfaz, 7 en algunas versiones de SCSI, en otras hasta 15 dispositivos, sin que se vuelva un cuello de botella la interfaz (en la medida de lo posible). Esto hace que SCSI sea la alternativa idónea para sistemas con muchos discos o varios dispositivos en general, que deben poder ser accedidos concurrentemente en un ambiente multitarea (como servidores, por ejemplo). En ambientes más sencillos y, en general, en máquinas con uno o pocos usuarios y una carga no muy pesada, los discos IDE son la

Versión	Bus (bits)	Transferencia (Mb/seg)
SCSI-1	8	5
Fast SCSI	8	10
Fast Wide SCSI	16	20
Ultra SCSI	8	20
Wide Ultra SCSI	16	40
Ultra 2 SCSI LVD	8	40
Wide Ultra 2 SCSI lvd	16	80
Wide Ultra SCSI 3 LVD	16	160

Tabla 6.2: Las diferentes versiones de SCSI. LVD significa *Low Voltage Differential* una estrategia para proveer de redundancia al bus de SCSI y así poder inmunizarlo a un amplio rango de errores, lo que hace posible tener un cable de hasta 25 metros confiable.

mejor opción, tienen tiempos de posicionamiento de dos o tres milisegundos más que los SCSI, pero cuestan la mitad.

SCSI sí es un estándar y ha pasado por varias versiones desde que se estableció en 1986. Las diversas versiones se encuentran listadas en la tabla 6.2.

6.3 El sistema operativo y la E/S

Así como el desempeño del sistema, en particular del CPU, depende en mucho de los compiladores, el desempeño del sistema de E/S depende en mucho del sistema operativo.

En general el sistema operativo debe:

- Garantizar que los procesos tengan acceso a los dispositivos si tienen permiso para ello. Es decir, el sistema debe poder poner el semáforo verde a los procesos.
- Asegurar que ningun proceso no autorizado para acceder a un dispositivo tenga acceso a él. Es decir, debe poder poner el semáforo rojo.
- Proveer de un conjunto de abstracciones que hagan manejables los dispositivos y oculten las complicaciones de hardware innecesarias (es más fácil leer un archivo que leer el sector X del cilindro Y, pista Z).

- Proveer de servicios a los dispositivos. Servicios indispensables en el momento preciso. Proveer por ejemplo, de las herramientas necesarias para el manejo de interrupciones y las rutinas de servicio asociadas o proveer los mecanismos necesarios para el monitoreo frecuente (*polling*).

- Optimizar el acceso a los recursos de E/S para mejorar, en lo posible, el desempeño. Por ejemplo que el diseño del sistema de archivos sea tal que minimice la fragmentación de datos en el disco y con ello los viajes largos de las cabezas de lectura/escritura.

- Proveer de servicios de alto nivel a los programadores. Por ejemplo las llamadas al sistema Unix que permiten hacer E/S de manera casi uniforme a dispositivos muy diversos.

- Garantizar la consistencia de la jerarquía de memoria cuando se tienen dispositivos de acceso directo a memoria (DMA). Garantizar que si el dato de la localidad X, que es escrito mediante acceso directo por un dispositivo, se encuentra alojado también en el cache el contenido de estos caches sea consistente con la memoria principal.

CAPÍTULO 7

Multiprocesadores

Por supuesto muchas cabezas piensan mejor que una, de igual manera muchos procesadores procesan, no mejor, pero si más rápido que uno. Así que se han diseñado y construido, a partir de la década de 1960, sistemas de cómputo con múltiples procesadores que operan en paralelo y colaboran para lograr un objetivo común.

Clasificación de Flynn.

En general las arquitecturas paralelas se han clasificado de acuerdo con la relación existente entre la entrada y los procesadores que la utilizan.

SISD (*Single Instruction Single Data*). Un sistema típico de un solo procesador.

SIMD (*Single Instruction Multiple Data*). La misma instrucción es ejecutada por múltiples procesadores sobre diferentes datos. Podemos pensar en muchos procesadores, cada uno con sus propios datos, que

ejecutan una misma instrucción.

MISD (*Multiple Instruction Single Data*). Varios procesadores ejecutan diferentes instrucciones sobre el mismo conjunto de datos. No hay máquinas de este tipo por el momento.

MIMD (*Multiple Instruction Multiple Data*). Los procesadores ejecutan sus propios programas con sus propios datos de entrada.

La clasificación fue propuesta por Flynn en 1966 y aun se usa extensamente para clasificar las arquitecturas paralelas.

Máquinas vectoriales.

Las supercomputadoras tradicionalmente han sido máquinas vectoriales, esto significa que son capaces de ejecutar unas cuantas instrucciones sobre grandes volúmenes de datos, lo que las coloca muy cerca del paradigma SIMD, aunque los puristas no lo consideren así (porque hay un solo procesador qua hace fetch y luego pasa la instrucción a unidades funcionales subordinadas). Las máquinas de Cray (ahora parte de Silicon Graphics) y las de la ahora desaparecida *Thinking Machines* son de este tipo.

Clusters.

Los cúmulos (*clusters*) que han cobrado popularidad hoy en día entran en el paradigma MIMD. Esta popularidad se debe fundamentalmente a la flexibilidad. Cada procesador en el sistema es, por sí mismo funcional, así que es posible pensar en usar cada uno por separado, todos coordinados en el sistema paralelo o una combinación de ambas cosas. Además la relación costo beneficio es muy ventajosa en el caso de estos sistemas, dado que es posible construirlos con procesadores comunes, no se requiere de diseños especiales. Sin embargo el desempeño obtenido puede ser muy aceptable, comparable con algunas supercomputadoras.

Organización de la memoria.

Por supuesto programas y datos deben estar en memoria. Si hay muchos procesadores cada uno de ellos necesita memoria, pero todos comparten algo, ya sea datos o instrucciones, así que podemos pensar en que compartan la memoria. Pero también podemos pensar en que no sea así. Cada unas de las alternativas es válida:

- Memoria compartida centralizada. Hay un solo espacio de memoria, un solo espacio de direcciones, todos los procesadores comparten la memoria.
- Memoria distribuida. Cada procesador posee su propio espacio de direcciones.

El común de nuestras máquinas paralelas de hoy en día son de memoria compartida. Como cada procesador accede a la memoria siguiendo las mismas normas y con el mismo derecho que cualquier otro procesador a veces a este esquema se le llama de acceso uniforme a la memoria (*Uniform Memory Access* o UMA). Cualquier máquina dual es de este tipo, posee dos procesadores y una sola memoria. En este esquema es conveniente dotar a los procesadores de caches individuales para no hacer de la memoria un cuello de botella, claro que esto genera problemas de consistencia que abordaremos después.

El esquema de memoria distribuida, como el que usan los *clusters*, tiene sus ventajas. Dado que, generalmente los procesadores accederán a un área restringida de memoria (principio de localidad), dotarlos de su propio espacio reduce el tiempo que cualquiera de ellos debe esperar por un dato, además hace que cada uno tenga para si mismo todo el ancho de banda disponible de la memoria. La desventaja es que cuando los procesadores deben comunicarse el tiempo de respuesta de comunicación es mayor, porque en general las lineas de comunicación entre ellos tienen menos ancho de banda que en los sistemas de memoria centralizada.

7.1 Comunicación entre procesadores

Memoria compartida Vs. intercambio de mensajes.

Por supuesto los distintos procesadores deben poder comunicarse de alguna manera. Si poseen acceso a un espacio de memoria compartido, ese puede ser el medio de comunicación, los resultados producidos por cada procesador son puestos allí, donde pueden ser accedidos y actualizados por otros procesadores.

La otra posibilidad es que los procesadores intercambien mensajes con los datos que requieren para trabajar y que no son producidos por ellos mismos.

Estos son los dos mecanismos de comunicación posibles. Cada uno posee ventajas. El mecanismo de memoria compartida, por ejemplo:

1. Facilidad en la programación. La comunicación entre los procesadores se basa en algoritmos y mecanismos bien estudiados, determinísticos, no influidos por eventualidades.

2. El tráfico generado por la comunicación es mínimo. Las ligas de comu-

nicación son muy confiables en general y eso simplifica enormemente los protocolos.

3. Eficiente, dado que casi todo se puede hacer en hardware.

Las principales ventajas del esquema de comunicación por paso de mensajes son: Hardware más simple, comunicación explícita, lo que permite tener control permanente por parte del programador.

Por supuesto hay diferentes factores que influyen en el desempeño de la comunicación:

1. El ancho de banda. El ancho de banda efectivo es el mínimo de todas los dispositivos involucrados en la comunicación. El procesador es muy rápido en general, las memorias son bastante más lentas, sobre todo si son accedidas por múltiples procesadores, los canales de comunicación son más lentos aún, pero son todavía mucho más rápidos que las lineas de comunicación externa (la red en un *cluster*), en general.

2. La latencia. Este es el tiempo necesario para que una comunicación sea recibida. El emisor decide enviar datos, su interfaz de comunicación procesa el envío y lo transmite, los datos viajan a través del medio, son recibidos y procesados en el receptor hasta que llegan al proceso que realmente los necesita.

7.2 Memoria centralizada compartida

El esquema general que usaremos en esta sección es el mostrado en la figura 7.1. Hay un solo banco de almacenamiento principal compartido por todos los procesadores (etiquetados P_i en la figura). Cada procesador además posee su propio cache para optimizar sus accesos a memoria. Cada cache, como es usual, contiene un subconjunto de la memoria principal compartida. En los caches hay, en general, variables compartidas por todos los procesadores, así como variables privadas de cada procesador.

Coherencia de caches.

En un esquema así pueden ocurrir cosas como esta: La dirección X de memoria contiene un valor v. La dirección X es compartida por todos los procesadores. Un par de procesadores, digamos A y B, tienen, ambos, X en sus respectivos caches. Supongamos que B escribe un nuevo valor para X,

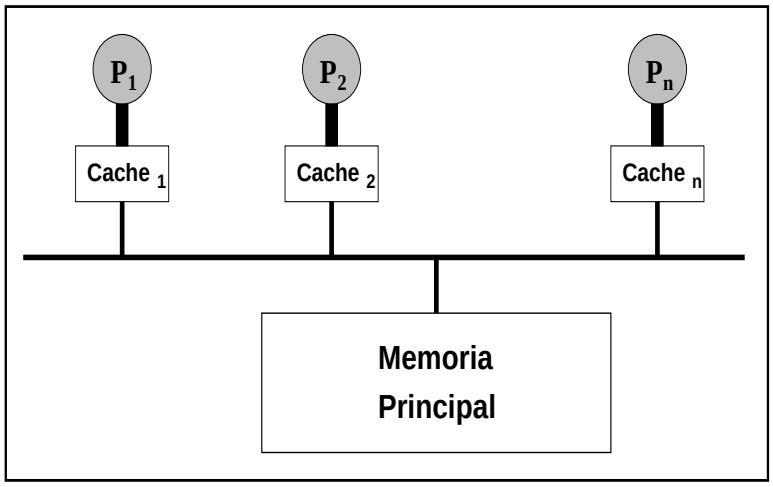

Figura 7.1: Esquema de memoria centralizada compartida.

v_B. Si A accede al valor de X, dado que está en su cache, no hay fallo de acceso y obtiene el valor v original, no el nuevo valor escrito por B. Aún si los caches fueran del tipo *write through* el valor de X sería actualizado por B en su cache y en la memoria principal, pero el valor leído por A seguiría siendo erróneo. A este problema se le llama de *coherencia de caches* y es uno de los más importantes en arquitecturas paralelas de memoria centralizada compartida.

Se dice que un sistema de cómputo paralelo de memoria centralizada compartida asegura la coherencia de caches si cumple con las siguientes tres condiciones:

- Si un procesador P escribe en X y nadie más escribe antes de que el mismo P lea X entonces debe ocurrir que P lee lo mismo que escribió.

- Si P escribe en X, transcurre un tiempo suficiente, durante el cuál ningún otro procesador escribe en X y luego otro procesador Q lee de X entonces debe ocurrir que Q lee el mismo valor que fue escrito por P.

- Si en la localidad X hay un valor v y algún procesador P escribe en X el valor v_P entonces debe ocurrir que ningún procesador puede leer el valor v de la localidad X luego de que P le haya escrito v_P.

Hay esencialmente dos maneras de lograr que se cumplan estas condiciones:

- Mediante un directorio en el que se registran todos los bloques de memoria compartida por los procesadores, en ese directorio se registra cada bloque y su estado actual. Si un procesador ha escrito en un bloque compartido en el directorio se marca que dicho bloque ha sido alterado. Si algún procesador necesita leer un dato contenido en el bloque compartido se dará cuenta de que debe leerlo de la memoria principal.

- Mediante la "escucha" del canal de comunicación con la memoria. Si cada cache posee un mecanismo de control que "escuche" el canal se percatará cuando haya escrituras de otros caches en la memoria principal y la dirección del dato a escribir (si los caches son *write through*). Cada cache puede actualizar sus valores con lo que "oye" en al canal. Como los caches deben estar de chismosos oyendo conversaciones ajenas a esta estrategia se le llama "fisgoneo" (*snooping*).

Estas dos soluciones se implementan en lo que se denomina un *protocolo de coherencia de caches*. Por supuesto una vez que un dato compartido ha sido alterado por un procesador, los demás procesadores deben actualizar la copia que tienen de ese dato en sus caches. Hay dos maneras de hacer esto en general:

- Difusión de escritura (*write broadcast*). Cuando se escribe una localidad de memoria compartida el escritor avisa al resto de los procesadores que deben actualizar el valor de su propia copia y proporciona el nuevo valor.

- Protocolo de invalidación. Cuando se escribe una localidad compartida se provee de un mecanismo para informar a todos los procesadores que, si tienen una copia de dicha localidad, esta ya no es válida.

En el esquema de difusión cada escritura requiere de una actualización, aún si hay varias escrituras en la misma localidad. En los protocolos de invalidación sólo hay actualización cuando se requiere el dato invalidado, lo malo es que esto implica un acceso a memoria principal. Por otra parte, los protocolos de actualización utilizan direccionamiento por palabras, porque

cada palabra puede ser escrita individualmente, en cambio en los protocolos de invalidación utilizan direccionamiento por bloques, así que si se han escrito varias palabras de un bloque, el protocolo de invalidación hace realmente una sola actualización de todo el bloque y no una por cada palabra alterada. Sin embargo los protocolos de actualización también tiene una ventaja: no se requiere de acceder a la memoria principal para hacer actualizaciones, porque toda la información requerida para actualizar es dada por el procesador que escribe.

7.3 Clusters

Las arquitecturas paralelas de las supercomputadoras de hace algún tiempo estaban diseñadas *ad hoc*, todos los componentes eran específicos para una máquina o línea de máquinas en particular. Desde siempre las supercomputadoras han tenido un uso limitado a ciertos sectores que requieren indispensablemente de cómputo de alto desempeño (computo científico, predicción climática, modelado de sistemas complejos, etc.), así que se venden poco. Asi que estamos hablando de una situación en la que se lleva a cabo todo un proceso de ingeniería, diseño y producción de componentes de la mejor calidad posible, con las mejores cualidades tecnológicas existentes y de los cuales se venden sólo unas decenas de ejemplares. El costo de producción se debe amortizar entre muy poca producción, como en un Ferrari. Por supuesto las supercomputadoras son muy caras, son los Ferraris del mundo de las cumputadoras.

¿Que se puede hacer para disminuir el costo? lo evidente es usar componentes que se produzcan en masa. De los mismos que se usan en computadoras más comunes, así se amortiza el costo de producción entre muchas más unidades. En una primera aproximación podríamos pensar en usar algunos componentes baratos y otros diseñados para obtener el mejor desempeño posible con esos componentes. Diseñar a la medida, pero no todo.

Esta es la alternativa explotada por las computadoras masivamente paralelas (MPP o *Massively Parallel Processors*). La Cray T3E por ejemplo usa procesadores normalmente usados para estaciones de trabajo, Alpha 21164 a 400 MHz. Para obtener un alto desempeño usa ¡2048 de ellos! conectados en un toro tridimensional (cada procesador tiene seis vecinos) esta maquinita puede tener un máximo de 4TB de memoria. Otro ejemplo menos impresionante después del anterior es la Origin 2000 con 128 procesadores

MIPS R10000 a 195 MHz.

Alternativa barata.

Una segunda alternativa es usar *sólo* componentes comunes. Máquinas ya construidas, ya sea estaciones de trabajo o PC's, interconectar muchas de ellas (hasta un millar o más) y darle cohesión a este cúmulo (*cluster*) de computadoras mediante hardware y principalmente software apropiado. Al no usar hardware especialmente diseñado para el sistema el costo de un *cluster* es mucho menor que el de un MPP y su desempeño puede llegar a ser competitivo con el de aquel. En general los *clusters* (por ahora) no tienen todo el desempeño que tienen las mejores máquinas masivamente paralelas, pero sin lugar a dudas la relación costo-beneficio les da ventaja. Actualmente (inicios de 2001) los MPP tienen un desempeño tres veces mejor que los mejores *clusters* pero cuestan cuatro veces más.

La desventaja de usar máquinas ya construidas es que para interconectarlas se requiere de tecnologías de red local, no es posible usar buses de sistema para conectarlas. De hecho el desempeño total del *cluster* está comprometido por el desempeño de la red que conecta los componentes, el eslabón más débil de la cadena. Por ejemplo los buses de la T3E son de 1200 MB/seg= 1.2GB/seg mientras que en Gigabit Ethernet son de 1 Gb/seg = 125 MB/seg = 0.122 GB/seg, un orden de magnitud de diferencia en ancho de banda.

Clasificación de *clusters*.

En general podemos clasificar los *clusters* en:

- Centralizados. Todas las máquinas de *cluster* están dedicadas exclusivamente a la cooperación dentro del *cluster*.
- Descentralizados. Las máquinas del *cluster* son usadas principalmente para labores individuales y poseen lo necesario para que, ocasionalmente, sea posible ponerlas en colaboración.

La distinción es, fundamentalmente de uso. En un *cluster* centralizado todas las maquinas están permanentemente colaborando, en un descentralizado las máquinas se dedican individualmente a labores varias determinadas por sus usuarios y solo en caso necesario se integran en un *cluster*.

Limitantes del desempeño.

Hemos dicho que la red se transforma en la mayor barrera de desempeño en un *cluster*. Para resolver este problema se requiere de una arquitectura de red:

- De alto ancho de banda.

- De baja latencia. La latencia es el tiempo promedio que transcurre entre que la aplicación en la máquina A envía un mensaje a la máquina B y este mensaje es recibido por la aplicación en B. Nótese que se habla de que la aplicación y no la máquina reciba el mensaje, es decir también se esta considerando el procesamiento que del mensaje hacen los protocolos de comunicación bajo la aplicación.

Las redes que se han usado a la fecha, a falta de algo mejor, son:

- *Fast Ethernet* con un ancho de banda de 100 Mb/seg = 12.5 MB/seg. Por supuesto usando *switches* que permitan evadir las colisiones y el trafico excesivo en las líneas de comunicación.
- ATM (*Asynchronous Transfer Mode*) con un ancho de banda de 622 Mb/seg = 77.75 MB/seg capaz de establecer circuitos virtuales de comunicación.
- Myrinet con un canal *full duplex* (comunicación simultánea en ambos sentidos del canal) de 1.28 Gb/seg = 160 MB/seg en cada sentido, tarjetas programables de acceso a la red.

7.3.1 Software

Quizás el componente más importante en un *cluster* es el software involucrado. El objetivo del *cluster* es lograr procesamiento paralelo en los distintos procesadores del sistema. Para ello es necesario que posea dos cosas fundamentales: un administrador de procesos capaz de distribuir tareas entre los procesadores y un sistema de comunicación entre procesadores.

Administrador de procesos.

El administrador de procesos, como en el caso de un sistema operativo (de hecho es parte del sistema operativo del *cluster*) se denomina *scheduler* o despachador. El despachador debe poder asignar procesador y tiempo a cada proceso procurando, en la medida de lo posible no dejar desocupados a los procesadores y coordinando la ejecución de los procesos de tal forma que los resultados necesarios para uno de ellos sean generados antes de que este se ejecute.

Sistema de comunicación.

El sistema de comunicación debe proveer de funciones elementales de envío y recepción de mensajes entre procesadores para permitir su sincronización. El paso de mensajes, es decir las operaciones de envío *send* y recepción (*receive*) pueden ser síncronas, es decir un envío o recepción bloquean

el proceso que las invoca hasta que el dato esperado sea recibido (acuse de recibo) o enviado, respectivamente, por la contraparte de la operación. Otra posibilidad es que el proceso envie sin esperar a que el receptor responda que ha recibido el dato, en ese caso las primitivas de comunicación son de no-bloqueo (*non-blocking*). También es posible que los datos enviados sean almacenados en un lugar, temporalmente, mientras el receptor está listo para procesarlos (*buffered messages*). Los protocolos de no-bloqueo son más rápidos pero menos confiables que los de bloqueo.

PVM Vs. MPI.

Hay principalmente dos sistemas de paso de mensajes en uso: PVM (*Parallel Virtual Machine*) y MPI (*Message Passing Interface*). PVM fue diseñado para *clusters* Unix en 1994, sus primitivas de comunicación son de bloqueo pero con opción de tiempo máximo de espera (*time-out*), consta de una biblioteca de servicios de comunicación y un *daemon* que se ejecuta en cada máquina del *cluster*. MPI es un estándar, no crea procesos como PVM, sólo agrupa procesos creados independientemente, les define un contexto y les asocia un administrador de comunicación (*communicator* en el contexto de MPI). En MPI es posible definir una topología virtual entre los procesadores y posee también tipos de datos pre-definidos extensibles para definir el contenido de los mensajes.

Protocolos ligeros de paso de mensajes.

En general PVM parece ser más fácil de aprender pero menos versátil y poderoso que MPI, aunque la comunidad de PVM considera que MPI ha incluido demasiadas cosas lo que lo hace pesado para el objetivo para el que fue hecho. En principio, dado que la red de comunicación es el eslabón más lento del *cluster*, el objetivo es optimizar su uso. Con el ancho de banda no se puede hacer mucho, es una cualidad intrínseca del canal, pero la latencia si tiene que ver con los protocolos de comunicación, así que hacerlos más ligeros, disminuir su complejidad, es fundamental para optimizar el uso de la red. Es por eso que se han creado protocolos ligeros de paso de mensajes específicos para *clusters* y que los dispositivos de acceso (tarjetas de red) se han hecho más inteligentes, por ejemplo acceso directo a la memoria de la aplicación para evitar el paso por el sistema operativo y por tanto el procesador, que debe ejecutar cosas esenciales. A esta estrategia se le conoce como "evitar al sistema operativo" (*bypass operating system*).

Bibliografía

[1] Centurión, José, "Operación Silicio", *Muy Interesante*, XVII(5), mayo 2000, pp. 75-77.

[2] Diefendorff, Keith, "Benchmarks are Bunk", *Microprocessor Report*, 14(26), junio 26, 2000. Disponible en: `http://www.mdronline.com/mpr_public/editorials/edit14_26.html`

[3] Hennessy, John y David A. Patterson, *Computer Architecture A Quantitative Approach*, 2a edición, Morgan Kaufmann, 1996.

[4] Jain, Raj, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, 1991.

[5] SPEC, *Systems Performance Evaluation Corporation*, `http://www.spec.org`.