Arquitectura de Computadores

Por PASCAL

UNIVERSIDAD COMPLUTENSE MADRID

Grado en Ingeniería Informática
Facultad de informática

Arquitectura de Computadores

Madrid, Curso 2018–2019

Sobre TEFLON

Teflon(cc0 1.0(documentación) MIT(código))es una plantilla de LaTeX creada por David Pacios Izquierdo con fecha de Enero de 2018. Con atribuciones de uso CC0.

Esta plantilla fue desarrollada para facilitar la creación de documentación profesional para Trabajos de Fin de Grado o Trabajos de Fin de Máster. La versión usada es la 1.3.

V:1.3 Overleaf V2 with pdfLaTeX, margin 1in, NO-bib

Contacto
Autor: David Pacios Izquiero
Correo: dpacios@ucm.es
ASCII: asciifdi@gmail.com
Despacho 110 - Facultad de Informática

Índice general

Página

1. Tema 1: Introducción: Tendencias Tecnológicas/ Coste/ Rendimiento/ Consumo **3**

1.1. Arquitectura del computador 4
1.1.1. Visión clásica 4
1.1.2. Visión actual 4
1.1.3. Propiedades de ISA 5
1.1.4. Paralelismo 5
1.2. Evolución y tendencias 6
1.2.1. Rendimiento del procesador 6
1.2.2. Supercomputadores 7
1.2.3. Tecnología 7
1.3. Rendimiento 10
1.3.1. ¿Qué se entiende por rendimiento de un sistema? 10
1.3.2. Medidas del rendimiento 10
1.3.3. Un principio simple: Ley de Amdahl 13
1.3.4. Concepto de Eficiencia 18
1.4. Consumo: Potencia y energía 18
1.5. Coste 19
1.6. Explicación de ejercicios resueltos 20

2. Tema 2: ILP, Planificación dinámica, Predicción de saltos, Especulación **31**

2.1. Introducción: ILP 32
2.1.1. ¿Qué técnicas conocemos? 32
2.1.2. Mecanismos para explotar ILP 32
2.1.3. Paralelismo a nivel de instrucción ILP 32
2.1.4. Técnicas para explotar ILP 33
2.1.5. Dependencias 33
2.1.6. Técnicas SW para explotar ILP 36
2.2. Tratamientode dependencias de datos en ejecución 42
2.2.1. Planificación dinámica: Procesador 42
2.2.2. Planificación Dinámica: Tomasulo 45
2.2.3. Tres estados para una instrucción en el algoritmo 47
2.2.4. Ejemplo: Ejecución de la secuencia 48
2.2.5. Ventajas e incovenientes 62
2.2.6. Reduciendo la penalización de los saltos 63
2.2.7. Tratamiento de Saltos: Otras alternativas 78

2.3. Especulación 79
2.3.1. Algoritmo de Tomasulo con especulación 81
2.3.2. Saltos e interpretaciones 88
2.3.3. Riesgo a través de memoria 89
2.4. Explicación Ejercicios Resueltos 89

3. Tema 3: Lanzamiento múltiple, Límites de ILP, Multithreading 109
3.1. Introducción: CPI <1 110
3.2. Lanzamiento múltiple de instrucciones: Superescalar 110
3.2.1. Procesador Superescalar con planificación estática 110
3.2.2. Procesador Superescalar con planificación dinámica 111
3.3. Límites del ILP 130
3.3.1. Comparaciones necesarias en lanzamiento 130
3.3.2. Comparaciones necesarias para paso de operandos 130
3.3.3. Modelo versus procesador real 131
3.3.4. ¿Cómo superar los limites de este estudio? 131
3.3.5. Buscar paralelismo de más de un thread 132
3.3.6. Thread Level Parallelism vs ILP 132
3.4. Multithreading 132
3.4.1. Ejecución Multithreaded 133
3.5. Explicación de ejercicios resueltos 134

4. Tema 4: Paralelismo a nivel de datos: arquitectura vectorial, instrucciones SIMD para multimedia, GPUs 153
4.1. Introducción 154
4.2. Arquitectura vectorial 154
4.2.1. Operaciones vectoriales 155
4.2.2. Repertorio VMIPS 156
4.2.3. Código escalar vs Código vectorial 156
4.2.4. Tiempo de ejecución 157
4.2.5. Ejecución de operaciones aritméticas 159
4.2.6. Encadenamiento 160
4.2.7. Vectores de longitud arbitraria 161
4.3. Instrucciones SIMD para procesamiento multimedia 165
4.3.1. Extensiones SIMD 165
4.4. Unidades para Procesamiento Gráfico (GPUs) 166
4.4.1. Arquitectura de NVIDIA GPU 167
4.4.2. Instrucciones PTX 168
4.4.3. Thread Scheduler 169
4.4.4. Saltos condicionales en GPUs 170
4.5. Paralelismo a nivel de bucle: vectorización 171
4.5.1. Ejemplos 171
4.5.2. Antidependencias y dependencias de salida 172
4.6. Explicación de ejercicios resueltos 172

5. Tema 5: Multiprocesadores y redes de interconexión 189
5.1. Introducción . . . 190
5.2. Arquitecturas Paralelas . . . 190
5.2.1. Arquitecturas de memoria compartida . . . 190
5.2.2. Arquitecturas de Paso de Mensajes . . . 193
5.3. Redes de interconexión . . . 194
5.3.1. Introducción . . . 194
5.3.2. Descripción estructural . . . 194
5.3.3. Topología de red . . . 197
5.3.4. Técnicas de conmutación . . . 200
5.3.5. Encaminamiento . . . 202
5.4. Explicación de ejercicios resueltos . . . 205

6. Tema 6: Multiprocesadores: Coherencia, sincronización y consistencia 211
6.1. Jerarquía de memoria extendida . . . 212
6.1.1. Alternativas de implementación . . . 212
6.1.2. Modelo intituivo de memoria . . . 213
6.2. Problemas de la coherencia cache . . . 214
6.2.1. Soluciones de “Grano Grueso” . . . 214
6.2.2. Coherencia . . . 215
6.2.3. Protocolo Snoopy de 2 Estados . . . 217
6.2.4. Protocolo MSI: Invalidación de 3 estados (multiprocesador) . . . 218
6.2.5. Implementación de protocolos Snoopy . . . 220
6.2.6. Protocolo MESI: Invalidación de 4 estados . . . 220
6.3. Coherencia basada en directorio . . . 221
6.3.1. Directorio plano basado en memoria . . . 223
6.3.2. Directorio plano basado en cache . . . 224
6.4. Sincronización . . . 225
6.4.1. Introducción . . . 225
6.4.2. Primitivas hw básicas . . . 226
6.5. Consistencia de memoria . . . 228
6.5.1. Sincronización mediante Flag . . . 228
6.5.2. Consistencia de memoria . . . 229
6.5.3. Consistencia secuencial . . . 229

Capítulo 1

Tema 1: Introducción: Tendencias Tecnológicas/ Coste/ Rendimiento/ Consumo

En este capítulo vamos a ver los siguientes conceptos.

- Arquitectura del Computador.
- Evolución y tendencias:
  - Procesador.
  - Supercomputadores.
  - Tecnología: Ley de Moore, binomio μ-electrónica-μ-arquitectura.
- Rendimiento:
  - Métricas de evolución.
  - Un principio simple: Ley de Amdahl.
- Consumo.
- Coste.
- Explicación de ejercicios resueltos.

1.1. Arquitectura del computador

1.1.1. Visión clásica

- Las decisiones de diseño se ocupan básicamente del repertorio de instrucciones (Instruction Set Architecture, ISA).
- Decisiones relacionadas:
  - Registros.
  - Modos de direccionamiento.
  - Tipos de operandos: Enteros, PF.
  - Tipos de instrucciones: Aritméticas, transferencia, control de flujo.
  - Codificación de instrucciones.

Instruction Set Architecture

El conjunto de instrucciones de un procesador (o computador) se refiere a todas las instrucciones que el procesador reconoce y, por lo tanto, puede ejecutar. La arquitectura de un conjunto de instrucciones se refiere a todos los aspectos relacionados al diseño del conjunto de instrucciones, tales como registros a usar, tamaño de operandos, tamaño del código de instrucciones (formato de instrucciones), tamaño del opcode, tamaño y tipo de operandos, forma en que los datos e instrucciones son almacenados en memoria, etc. Distintos procesadores tienen distintos tipos de instrucciones.

1.1.2. Visión actual

- Punto de partida: Optimizar el rendimiento. Restricciones:
  - Coste.
  - Consumo de energía.
  - Tamaño.
  - Disponibilidad.
- Además incluye:
  - **Microarquitectura** (u organización): Estructura de la CPU, tipos de paralelismo soportado, modelo de ejecución de instrucciones, soporte de jerarquía de la memoria.
  - **Hardware**: Diseño lógico detallado, tecnología, ciclo de reloj.
- **Papel del arquitecto**: Compiladores y SO, hasta diseño VLSI y encapsulado de chips, pasando por todo tipo de técnicas para lograr alto rendimiento sin disparar consumo.

1.1.3. Propiedades de ISA

- Permanencia con el tiempo/ tecnología (portabilidad).
- Proporciona funcionalidad eficaz a los niveles superiores.
- Permite implementación eficiente en los niveles inferiores.

1.1.4. Paralelismo

Es una propiedad inherente a las aplicaciones:

- No todos los cálculos necesarios hay que hacerlos en orden estricto: Hay ordenaciones alternativas.
- No todos los cálculos deben hacerse en secuencia: Algunos cálculos pueden hacerse simultáneamente.

Se traduce a nivel de:

- Instrucciones (**Instruction-Level Parallelism**):
  - Ejecución de varias instrucciones a la vez, potencialmente en orden diferente al del programa.
  - Básicamente transparente al programador.
- Datos (**Data-Level Parrallelism**):
  - Arquitectura vectorial, instrucciones multimedia, GPUs.
  - Aplicación de una misma instrucción de LM a un conjunto de datos.
- Hebra (**Thread-Level Parallelism**):
  - Hebras de cálculos independientes que se ejecutan concurrentemente sobre los recursos hw disponible (en uno o varios procesadores).

1.2. Evolución y tendencias

1.2.1. Rendimiento del procesador

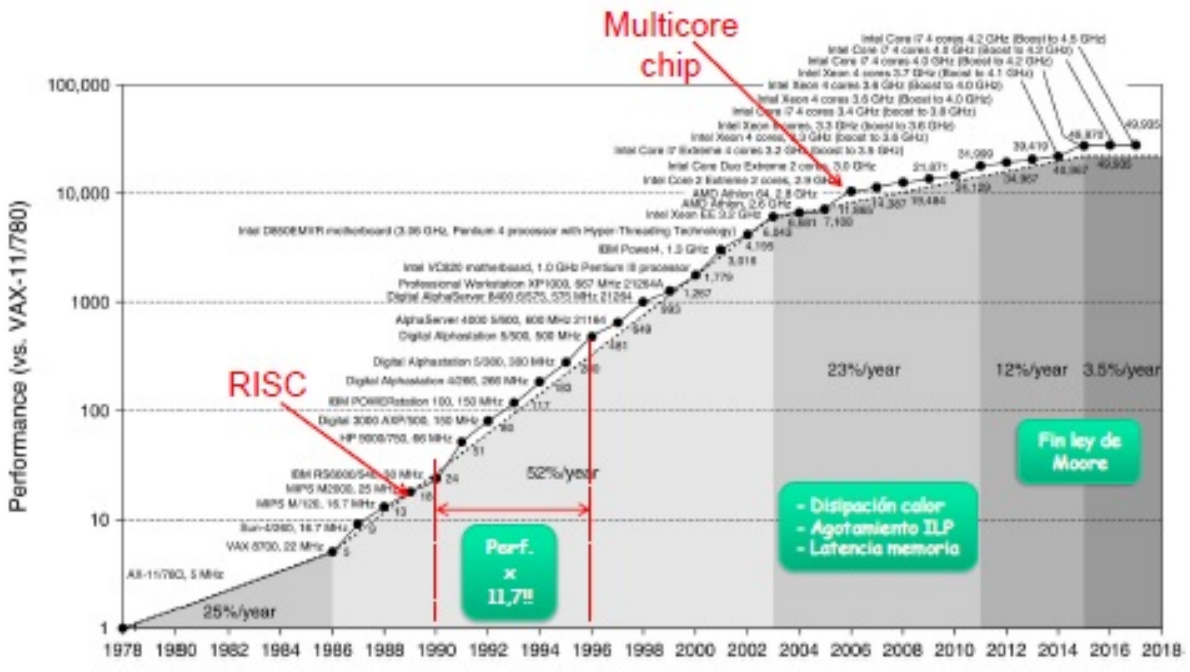


Figura 1.1: Rendimiento del procesador

Procesadores
En este apartado vamos a destacar los CISC y RISC.

RISC (**Reduced Instruction Set Computer**) propone una estrategia de diseño de CPU con instrucciones sencillas. La simplicidad hace que las instrucciones se ejecuten muy rápido.
CISC (**Complex Instruction Set Computer**) por el contrario fue la estrategia primitiva donde las instrucciones (y el hardware) eran más complejas.
Los procesadores RISC en general necesitan más instrucciones para ejecutar la misma tarea que un CISC.
Características de las Arquitecturas RISC:

- Juego de instrucciones reducido.
- Acceso a memoria limitado a instrucciones de carga/ almacenamiento.
- Muchos registros de uso general.
- Pocos modos de direccionamiento (inmediato, directo, indexado).
- Formato de instrucciones homogéneo.
- Todas las instrucciones se ejecutan en un ciclo de reloj.
- Utiliza unidades de proceso segmentadas y memorias cache.

1.2.2. Supercomputadores

Evolución del rendimiento de los computadores más potentes.
TOP500: Proporciona una lista ordenada de sistemas de propósito general que se están usando para aplicaciones de alta demanda computacional (High Perfomance Computing, HPC). Se utiliza el test LINPACK para resolver un sistema denso de n ecuaciones lineales, n es seleccionado por el sistema y cualquier sistema específicamente diseñado para la ejecución de LINPACK queda descalificado en TOP500.
GREEN500: Publicada desde 2007, ordena los sistemas en sentido decreciente de eficiencia energética.

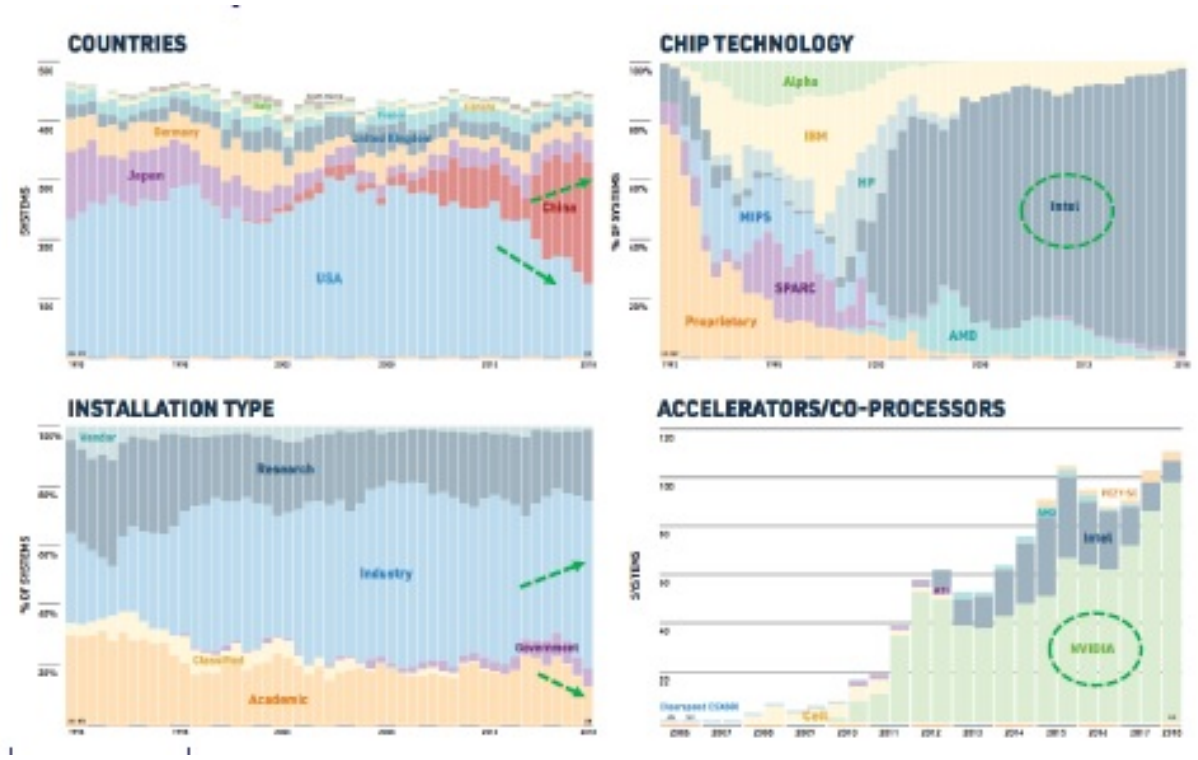


Figura 1.2: TOP500: Actores principales

1.2.3. Tecnología

- 1949: EDSAC 10^2 op/seg.
- 1957: Transistor de 10^3 a 10^4 op/s.
  - DEC PDP-I (1957).
  - IBM 7090 (1960).
- 1971: Microprocesador.
  - Intel 4004.
- 2003: Más de 3×10^{13} op/s.
- 2008: $> 10^{15}$ FLOP/s= (1 PFLOPS).

- Junio 2018: TOP500:
  - 1º SUMMIT: 2282544 cores: 122.3 PFLOPS (IBM Power 9, NVIDIA).
  - 22º Mare Nostrum 4: 153216 cores: 6.47 PFLOPS (Intel Skylake).

Ley de Moore

La Ley de Moore se refiere a una observación hecha por el cofundador de Intel, Gordon Moore, en 1965, en la que descubrió que el número de transistores por pulgada cuadrada en los circuitos integrados se había estado duplicando año tras año desde su invención.
La Ley de Moore vaticina que esta tendencia se mantendrá intacta en los próximos años. Aunque el ritmo ha disminuido, el número por pulgada cuadrada se duplicó aproximadamente cada año y medio. Esto se usa como la definición actual de la Ley de Moore.
La versión simplificada de esta ley establece que las velocidades del procesador o la potencia de procesamiento general para ordenadores se **duplicarán cada dos años**. Un rápido chequeo entre técnicos de diferentes empresas informáticas muestra que el término no es muy popular, pero la regla sigue siendo aceptada.
Si examináramos las velocidades de los procesadores desde 1970 hasta 2018 y luego de nuevo en 2019, podríamos pensar que la ley ha alcanzado su **límite** o se está acercando. En la década de 1970, las velocidades del procesador oscilaban entre 740 KHz y 8 MHz. Sin embargo, en realidad es más preciso aplicar la ley a los transistores que a la velocidad. La cantidad de potencia informática que ahora podemos utilizar en los dispositivos más pequeños, es algo más notable en comparación con lo que se podía lograr, digamos, hace una década.
Mirando hacia atrás, incluso cinco años más o menos, un PC que era el mejor en su momento, sería considerado anticuado si se comparara con un PC actual.
Esto es posible simplemente porque los fabricantes de chips son capaces de **aumentar el número de transistores** en un chip significativamente cada año, a medida que mejoran los avances en la investigación de chips.
La extensión de la Ley de Moore es que los ordenadores, los componentes que funcionan en ordenadores y la potencia informática se vuelven más pequeños y rápidos con el tiempo, a medida que los transistores de los **circuitos integrados** se vuelven más eficientes. Los **transistores** son simples interruptores electrónicos de encendido y apagado integrados en microchips, procesadores y pequeños circuitos eléctricos. Cuanto más rápido procesan las señales eléctricas, más eficiente se vuelve un ordenador.

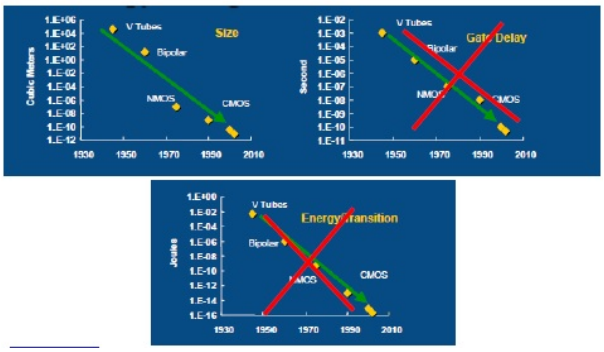


Figura 1.3: Escalado de la Ley de Moore

Cincuenta años después de la Ley de Moore, la sociedad contemporánea ve docenas de

beneficios expuestos por esta ley. Los dispositivos móviles, como los teléfonos inteligentes y los ordenadores de sobremesa, no funcionarían sin **procesadores muy pequeños**. Los ordenadores más pequeños y rápidos mejoran el transporte, la atención médica, la educación y la producción de energía. Casi todas las facetas de una sociedad de alta tecnología se benefician del concepto de la Ley de Moore puesta en práctica.
Hoy en día, todos los procesadores de consumo están hechos de **silicio**, el segundo elemento más abundante en la corteza terrestre, después del oxígeno. Pero si el silicio no es un conductor perfecto, y los límites a la movilidad de los electrones que lleva imponen un límite duro en cómo puedes densamente embalar los transistores de silicio.
Pero no solo el **consumo de energía** representa un problema enorme, sino también un efecto llamado túnel cuántico puede causar problemas para mantener los electrones contenidos más allá de un cierto umbral de espesor.
Los transistores de silicio alcanzan actualmente los 14 nanómetros, y aunque algunos diseños de chips de 10 nanómetros llegarán pronto al mercado, se ha llegado a la conclusión de que para cumplir con la Ley de Moore durante un largo período de tiempo, las empresas tendrán que crear materiales más nuevos y mejores para que sean la base de los ordenadores de la próxima generación.

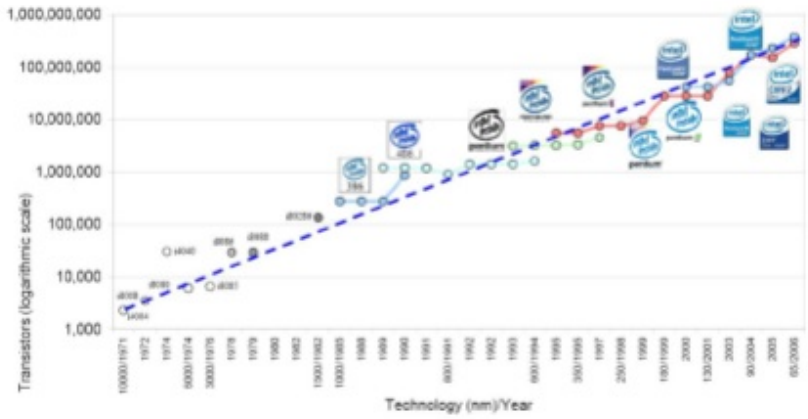


Figura 1.4: Predicción Moore Intel

Aunque la Ley de Moore había dicho cada dos años, este rápido aumento en la producción tecnológica ha disminuido el período en las mentes de los técnicos y usuarios por igual.
Debido a la Ley de Moore sugiere un crecimiento exponencial, es poco probable que continúe indefinidamente. La mayoría de los expertos esperan que la Ley de Moore dure otras dos décadas. Algunos estudios han demostrado que las limitaciones físicas podrían alcanzarse en 2018.

Ancho de banda vs Latencia

Ancho de banda (o rendimiento):

- Cantidad de trabajo realizado en un tiempo dado.
- Mejora en procesadores: 32000X-40000X.
- Mejora en memoria y discos: 3000X-1200X.

Latencia (o tiempo de respuesta):

- Tiempo entre el comienzo y la conclusión de una tarea.
- Mejora en procesadores: 50X-90X.
- Mejora en memoria y discos: 6X-8X.

1.3. Rendimiento

1.3.1. ¿Qué se entiende por rendimiento de un sistema?

Importante

Conviene elegir la unidad adecuada para medir el rendimiento y a veces hace falta considerar más de un parámetro para hacer más correcta la decisión a tomar.

Comparación de rendimientos

Rapidez de una computadora:

- Tiempo de respuesta: Una computadora es más rápida si ejecuta un determinado programa en menor tiempo.
- Incremento de productividad: Cantidad de tareas realizadas por la máquina en una determinada unidad de tiempo.

Al comparar rendimientos, lo que realmente queremos hacer es comparar velocidades relativas entre dos máquinas A y B.
Decir que "A es más rápida que B" significa que el tiempo de respuesta o el tiempo de ejecución de A es menor que el de B para una tarea dada.

$$n = \frac{\text{Tiempo de ejecución de } \quad B}{\text{Tiempo de ejecución de } \quad A}$$

1.3.2. Medidas del rendimiento

Importante

La única medida fiable es el tiempo de ejecución programas reales. Dos aspectos:

- Rendimiento del procesador.
- Rendimiento del computador.

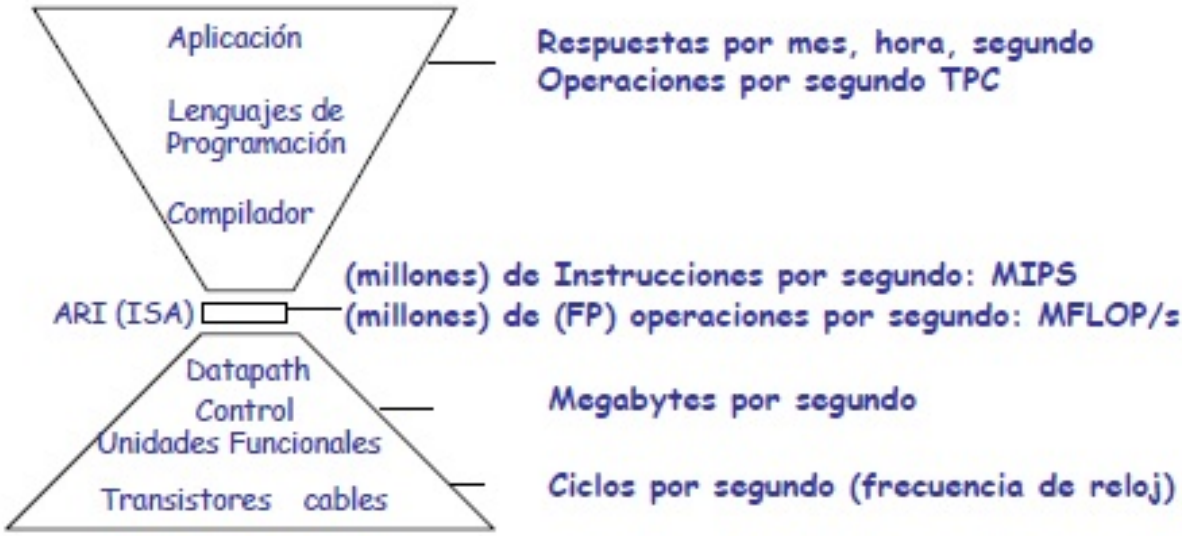


Figura 1.5: Esquema rendimiento

Si definimos al rendimiento como el inverso del tiempo de ejecución:

$$\text{Rendimiento} = \frac{1}{\text{Tiempo de ejecución}}$$

$$\frac{\text{Tiempo de ejecución de}B}{\text{Tiempo de ejecución de}A} = n = \frac{\text{Rendimiento deA}}{\text{Rendimiento de}B}$$

El tiempo de CPU para un programa puede expresarse de dos maneras:

$$\text{Tiempo de CPU} = \text{Ciclos de reloj de CPU por programa} \times \text{Ciclos de reloj}$$

$$\text{Tiempo de CPU} = \frac{\text{Ciclos de reloj de CPU por programa}}{\text{Frecuencia de reloj}}$$

En lugar de los ciclos de reloj se puede contar el número de instrucciones ejecutadas-recuento de instrucciones-.
Si se conocen los ciclos de reloj y el recuento de instrucciones se puede calcular el número medio de ciclos de reloj por instrucción (CPI):

$$\text{CPI} = \frac{\text{Ciclos de reloj de CPU por programa}}{\text{Recuento de instrucciones}}$$

$$\text{Tiempo de CPU} = \text{RI} \times \text{CPI} \times \text{Ciclo de reloj}$$

Cada parámetro tiene las siguientes unidades:

$$\frac{\text{Segundos}}{\text{Programa}} = \frac{\text{Instrucciones}}{\text{Programa}} \times \frac{\text{Ciclos de reloj}}{\text{Instrucciones}} \times \frac{\text{Segundos}}{\text{Ciclo de reloj}}$$

El CPI indica el número medio de ciclos de reloj que necesita cada instrucción para ejecutarse.

$$T_{\text{CPU}} = T_{\text{CICLO}} \times \sum_{i=1}^{n} \text{CPI}_i \times C_i$$

Ejemplo 1:

TIPO	LD/ST	$\text{ARITH}_{\text{INT}}$	ARITH_{FP}	CONTROL
$\text{CPI}_{\text{PROMEDIO}}$	6	2	8	4
$\text{N}^{\text{o}}\text{INS}_{\text{CLASE}}$	15	50	20	15

Cuadro 1.1: Para un cierto programa

$$T_{\text{CPU}} = T_{\text{CICLO}} \times \{6 \times 15 + 2 \times 50 + 8 \times 20 + 4 \times 15\} = 410 \times T_{\text{CICLO}}$$

Ejemplo 2: ALU 1 ciclo (50 %), Ld 2 ciclos (20 %), St 2 ciclos (10 %), saltos 2 ciclos (20 %).

$$\text{CPI} = 1 * 0{,}5 + 2 * 0{,}2 + 2 * 0{,}1 + 2 * 0{,}2 = 1{,}5$$

SPEC

¿Cómo se calculan las métricas SPECspeed?
Para cada bechmark, i, se mide el tiempo de ejecución en el sistema a evaluar (SAE) y se calcula la razón:

$$r_i = \frac{\text{Tiempo de ejecución en la “máquina de referencia”}}{\text{Tiempo de ejecución en la SAE}}$$

¿Y en SPECrate?
Para cada benchmark, i, se lanza a ejecutar un nº de copias determinado por el usuario, nc, y se calcula:

$$r_i = \frac{\text{Tiempo de ejecución de 1 copia en la “máquina de referencia”}}{\text{Tiempo de ejecución de las nc copias el SAE}} \times nc$$

¿Qué es la máquina de referencia?

- Un computador elegido para comparar con los demás.
- ¿Dónde encuentro los tiempos de ejecución en la máquina de referencia? En la web de SPARC.
- ¿Es crucial la elección de la máquina de referencia? Si comparamos un par de computadores, el resultado no depende de la máquina de referencia.

¿Cómo se calculan los resultados globales en SPEC?

- Para cada suite, se calculan las razones $r_i (i = 1..n)$ correspondientes a los n benchmarks que la integran.
- Para cada suite se reporta un único valor, dado por la media geométrica (MG) de las n razones, es decir:
$$\sqrt[n]{r_1 \times r_2 \times ... \times r_n}$$

Ejemplo 3: Calcular "SPECspeed_int_base" de un cierto SAE:

A. Obtener los tiempos de ejecución de cada uno de los 10 benchmarks enteros en el SAE, con las opciones de compilación "base".

B. Calcular las razones $r_i(i = 1..,10)$ entre los t de ejecución en la máquina de referencia y los t de ejecución en el SAE.

C. Resultados (cuanto mayor rendimiento cuanto mayor).

$$\text{SPECspeed2017_int_base} = \sqrt[10]{r_1 \times r_2 \times ... \times r_{10}}$$

¿Qué ventaja aporta el uso de la MG?

Importante

El uso de la MG garantiza la neutralidad de la máquina de referencia.

	Máquina referencia	**Computador A**	**Razón**	**Computador B**	**Razón**
P1	R1	A1	R1/A1	B1	R1/B1
P2	R2	A2	R2/A2	B2	R2/B2

Cuadro 1.2: Ejemplo para calcular rendimiento con máquina de referencia

$$\text{Rendimiento A} = \sqrt[2]{\frac{R1}{A1} \times \frac{R2}{A2}}; \text{Rendimiento B} = \sqrt[2]{\frac{R1}{B1} \times \frac{R2}{B2}}$$

La relación de rendimientos entre A y B es independiente de la máquina de referencia elegida. Se demuestra operando:

$$\frac{\text{Rendimiento A}}{\text{Rendimiento B}} = \sqrt[2]{\frac{B1 \times B2}{A1 \times A2}} = \frac{\text{MG tiempo ejecución en B}}{\text{MG tiempo ejecución en A}}$$

1.3.3. Un principio simple: Ley de Amdahl

Aceleración del caso común

Optimizar el rendimiento de las ocurrencias más frecuentes sobre la menos frecuente. Este principio tiene varios puntos a favor sobre una optimización generalizada del rendimiento de la computadora.
Un diseño que trate de minimizar el tiempo de ejecución de todas las instrucciones de una CPU determinada seguramente requerirá agregados de lógica adicional que acrecenterá los costes de producción, por un lado y desmerecerá el rendimiento total del equipo en funcionamiento normal.

Ley de Amdahl: "La mejora obtenida al utilizar algún modo de ejecución más rápido está limitada por la fracción de tiempo que se pueda utilizar en ese modo más rápido".

$$\text{Aceleración} = \frac{\text{Rendimiento de la tarea completada usando la mejora cuando sea posible}}{\text{Rendimiento de la tarea completa sin utilizar la mejora}}$$

$$\text{Aceleración} = \frac{\text{Tiempo de ejecución de la tarea sin utilizar la mejora}}{\text{Tiempo de ejecución de la tarea utilizando la mejora cuando sea posible}}$$

La aceleración depende de dos factores:

- La fracción del tiempo de cálculo de la máquina original que puede utilizarse para aprovecharla mejor. Esta fracción (que llamaremos $\text{fracción}_{\text{mejorada}}$) es siempre menor o igual que 1.
- La optimización lograda por el modo de ejecución mejorado; es decir, cuanto más rápido se ejecutaría la tarea si solamente se utilizase el modo mejorado. Este valor es el cociente entre el tiempo del modo mejorado y el tiempo modo original y es siempre mayor que 1. Llamaremos a este valor $\text{aceleración}_{\text{mejorada}}$.

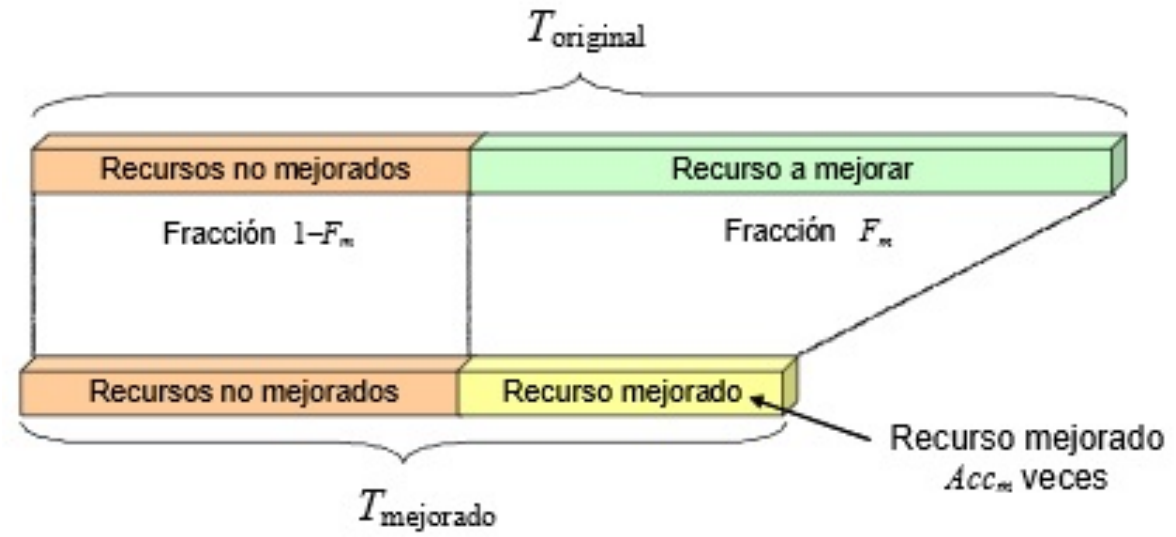


Figura 1.6: Esquema Ley de Amdahl

El tiempo de ejecución utilizando la máquina original con el modo mejorado será el tiempo empleado utilizando la parte no mejorada de la máquina más el tiempo empleado utilizando la parte mejorada.

$$\text{Tiempo de ejecución}_{\text{mejorado}} = \text{Tiempo de ejecución}_{\text{original}} *$$

$$* \left((1 - \text{Fracción}_{\text{mejorada}}) + \frac{\text{Fracción}_{\text{mejorada}}}{\text{Aceleración}_{\text{mejorada}}} \right)$$

$$\text{Aceleración}_{\text{global}} = \frac{\text{Tiempo de ejecución}_{\text{original}}}{\text{Tiempo de ejecución}_{\text{mejorado}}} = \frac{1}{(1 - \text{Fracción}_{\text{mejorada}}) + \frac{\text{Fracción}_{\text{mejorada}}}{\text{Aceleración}_{\text{mejorada}}}}$$

$$\text{Aceleración}_{\text{máxima}} = \frac{1}{1 - \text{Fracción}_{\text{mejorada}}}$$

Lo que expresa que lo máximo que podemos esperar es la fracción de la tarea mejorada se realice en un tiempo tendiente a cero.

En resumen

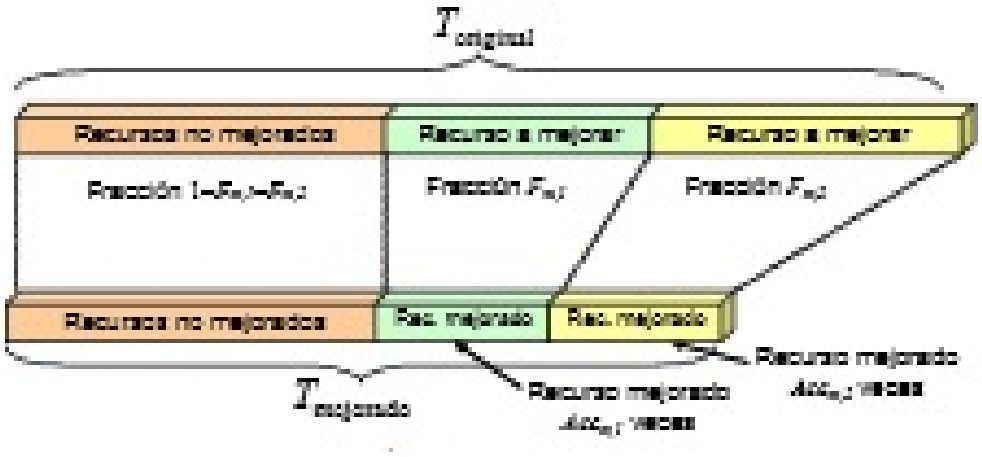


Figura 1.7: En resumen

Con una mejora:

$$\mathrm{Acc}_{\mathrm{global}} = \frac{1}{(1 - F_m) + \dfrac{F_m}{\mathrm{Acc}_m}}$$

Caso general con n mejoras:

$$\mathrm{Acc}_{\mathrm{global}} = \frac{1}{1 - \sum_{i=1}^{n} F_{m,j} + \sum_{i=1}^{n} \dfrac{F_{m,j}}{\mathrm{Acc}_{m,j}}} = \frac{1}{\sum_{i=0}^{n} \dfrac{F_i}{\mathrm{Acc}_i}}$$

Siendo F_i la fracción no mejorada y Acc_o (=1) la aceleración no mejorada.

Ejemplo 4: Un circuito realiza la operación de multiplicar en 2.3 microsegundos, de los cuales 0.5 μseg. Se necesitan para el cálculo de rebalse del resultado. Se realiza una versión optimizada de ese circuito que reduce a la mitad el tiempo de análisis de rebalse.

A. Hallar la fracción mejorada.

B. Hallar la aceleración mejorada.

C. ¿Cuál es la aceleración global del nuevo circuito respecto del anterior?

Se puede hallar la solución por dos vías:

- Solución convencional.
- Solución vía ley de Amdahl.

Solución convencional:

$$T_{\text{original}} = 2{,}3\mu\text{seg}$$

$$T_{\text{original-rebalse}} = 0{,}5\mu\text{seg}$$

$$T_{\text{nuevo-rebalse}} = 0{,}25\mu\text{seg}$$

$$T_{\text{mejorado}} = 2{,}3 - 0{,}5 + 0{,}25 = 2{,}05\mu\text{seg}$$

$$\text{Acc}_{\text{global}} = \frac{2{,}3}{2{,}05} = 1{,}12$$

Solución vía ley de Amdahl:

$$F_{\text{mejorada}} = \frac{0{,}5\mu\text{seg}}{2{,}3\mu\text{seg}}$$

$$\text{Acc}_{\text{mejorada}} = \frac{0{,}5}{0{,}25} = 2$$

$$\text{Acc}_{\text{global}} = \frac{1}{(1 - F_{\text{mejorada}}) + \dfrac{F_{\text{mejorada}}}{\text{Acc}_{\text{mejorada}}}} = \frac{1}{\left(1 - \dfrac{0{,}5}{2{,}3}\right) + \dfrac{\dfrac{0{,}5}{2{,}3}}{2}} = 1{,}12$$

Ejemplo 5: Un procesador A realiza una operación en N segundos. Se optimiza dicha operación en un procesador B que mejora 2/3 de la misma en un factor 2.5. Un tercer procesador C (basado en B) optimiza la operación duplicando la velocidad de la parte que mejora.
¿Cuál es la aceleración global si la mejora de C se realiza sobre la fracción que mejoró B (2/3)?

$$A : \begin{cases} f_{nm} = \dfrac{1}{3} \\ a_{nm} = 1 \end{cases} \rightarrow f_m = \frac{2}{3}; B : \begin{cases} f_{nm} = \dfrac{1}{3} \\ a_{nm} = 1 \end{cases} \rightarrow a_m = 2{,}5; C : \begin{cases} f_{nm} = \dfrac{1}{3} \\ a_{nm} = 1 \end{cases} \rightarrow a_m = 2$$

$$\text{Acc}_{B-A} = \frac{1}{1 - \dfrac{2}{3} + \dfrac{\dfrac{2}{3}}{2{,}5}} = 1{,}66; \text{Acc}_{C-A} = \frac{1}{1 - \dfrac{2}{3} + \dfrac{\dfrac{2}{3}}{2 * 2{,}5}} = 2{,}14$$

Una vez obtenido B, la proporción de las fracciones ya no es la misma.

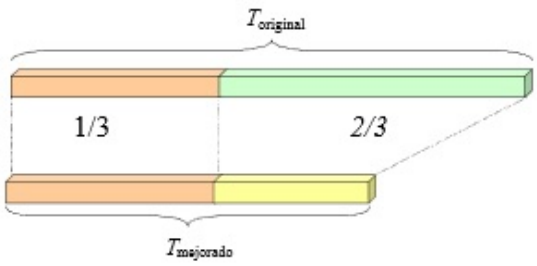


Figura 1.8: Proporción de las fracciones tras obtener B

Como $T_{\text{original}} > T_{\text{mejorado}}$ entonces las fracciones respecto de T_{mejorado} ya no son 1/3 y 2/3.
Ahora tenemos un problema, ¿cómo calculamos las nuevas fracciones de B para C?

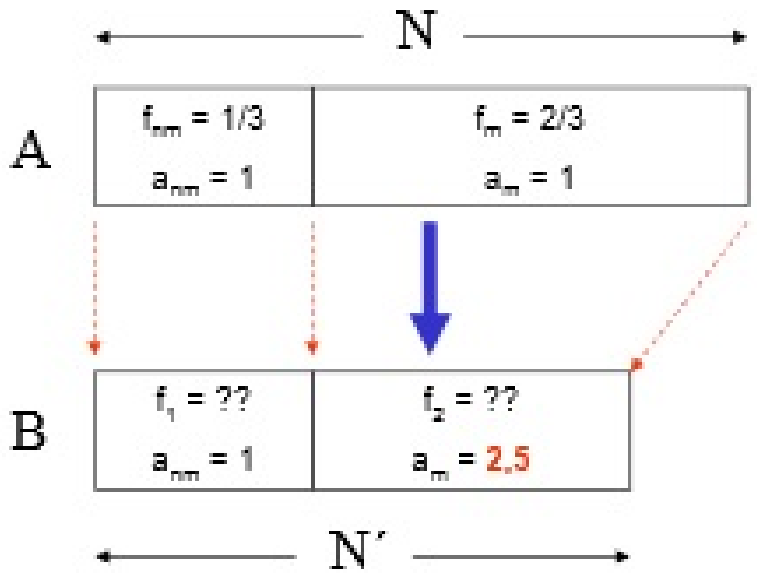


Figura 1.9: Nueva fracción

$$f_1 = \frac{5}{9}; f_2 = \frac{4}{9}$$

$$N' = \frac{\frac{1}{3}}{1}N + \frac{\frac{2}{3}}{2{,}5}N$$

$$\frac{N}{N'} = \frac{\frac{1}{3}}{1}\frac{N}{N'} + \frac{\frac{2}{3}}{2{,}5}\frac{N}{N'}$$

$$\text{Acc}_{B-A} = \frac{5}{3} = 1{,}66$$

$$\frac{N}{N'} = 1 = \frac{1}{3} * \frac{5}{3} + \frac{\frac{2}{3}}{\frac{10}{4}}\frac{5}{3} = \frac{5}{9} + \frac{4}{9}$$

$$\text{Acc}_{C-B} = \frac{1}{1 - \frac{4}{9} + \frac{\frac{4}{9}}{2}} = 1{,}29$$

$$\text{Acc}_{C-A} = \text{Acc}_{B-A} * \text{Acc}_{C-B} = 1{,}29 * 1{,}66 = 2{,}14$$

1.3.4. Concepto de Eficiencia

$$E = \frac{\text{Speedup}}{x} = \frac{\dfrac{1}{(1-F)+\dfrac{F}{x}}}{x} = \frac{1}{x(1-F)+F} = \frac{1}{x+F(1-x)}$$

Ejemplo 6: Para mejorar la velocidad de una aplicación, se ejecuta una parte que consumía el 90 % del tiempo sobre 100 procesadores en paralelo. El 10 % restante no admite ejecución en paralelo.

$$T_M = T_{SM} \times (0{,}1 + 0{,}9/100) = 0{,}109 \times T_{SM}$$

$$\text{Speedup} = \frac{1}{0{,}109} = 9{,}17$$

El uso de los procesadores sólo multiplica la velocidad por 9.17.
Ahora calculamos la eficiencia del ejemplo:

Procesadores (x)	**F**	**Speedup**	**Eficiencia**
10	0.9	5.26	0.526 (52.6 %)
100	0.9	9.17	0.0917 (9.17 %)
1000	0.9	9.91	0.00991 (0.99 %)

Ejemplo 7: El 10 % del tiempo de ejecución de mi programa es consumido por operaciones en PF. Se mejora la implementación de las operaciones PF reduciendo su tiempo a la mitad.

$$T_M = T_{SM} \times (0{,}9 + 0{,}1/2) = 0{,}95 \times T_{SM}$$

$$\text{Speedup} = \frac{1}{0{,}95} = 1{,}053$$

Mejora de sólo un 5.3 %.

1.4. Consumo: Potencia y energía

El consumo de potencia eléctrica es uno de los principales retos en el diseño de computadores.
El problema es que el consumo se transforma en calor a disipar.
Debemos considerar:

- Consumo máximo de potencia posible (pico).
- Consumo de potencia sostenida. Thermal Design Power (TDP):
  - Determina la potencia del sistema de refrigeración y de la fuente de alimentación.
  - Está por debajo de la potencia “pico” y por encima de la “media”.

- Energía y eficiencia energética: Comparar el rendimiento de los procesadores.

Objetivos:

- Autonomía.
- Tamaño compacto.
- Reducir coste y consumo del sistema de refigeración.

Energía dinámica: Consumida para la conmutación de los transistores.

- Transistores más pequeños, rápidos y de menor consumo (menor energía por conmutación).
- Evitar conmutaciones innecesarias.

Energía estática: Se consume aunque los transistores no estén conmutando.

- Reducir leakage current.
- Evitar leakage en módulos que no se están usando. Suprimir alimentación.

Factores que influyen en la $E_{\text{dinámica}}$:

$$E_{\text{dinámica}} = VDD^2 * C_{\text{load}}$$

Reducir VDD. Impacto: Influye en el consumo de energía, lineal sobre la frecuencia máxima y rendimiento.
Reducir C_{load}: Microarquitectura.
Potencia$_{\text{dinámica}}$=$VDD^2 * C_{\text{load}}$ * Frecuencia.
Reducir la frecuencia de reloj reduce la potencia, pero no la energía.

1.5. Coste

$$\text{Coste de CI} = \frac{\text{Die coste} + \text{Testing coste} + \text{Packaging coste}}{\text{Final test yield}}$$

$$\text{Die coste} = \frac{\text{Coste del Wafer}}{\text{Dies por Wafer} * \text{Die yield}}$$

$$\text{Die yield} = \text{Wafer yield} \times 1/(1 + \text{Defects per unit area X Die arca})^N$$

Factores que influyen en el binomio coste-precio:

- Mejora de aprendizaje:
  - Mejora del "yield": Fracción de unidades fabricadas que pasan los test de calidad.
  - Abarata el coste por unidad, permite disminuir el precio.
- Volumen de producción:
  - Estimación para coste μ-procesadores: 10 % menos al duplicar el volumen.
  - Curva de aprendizaje más rápida.
- Paso de equipos especializados a "commodities": Sector muy competitivo, mayor margen de beneficio, menor precio.

1.6. Explicación de ejercicios resueltos

Ejercicio 1. Ejecutamos los programas P1 y P2 en los computadores A, B y C y cronometramos los tiempos de ejecución, para evaluar las prestaciones. Las medidas anotadas son:

Programa	Computador A	Computador B	Computador C
P1	1	10	20
P2	1000	100	20

Se pide:

A. Hacer las seis comparaciones posibles entre computadores cuando ejecutan P1 o P2, de la manera "X es N veces más rápido que Y ejecutando el programa P".

B. Hacer las tres comparaciones globales posibles para los tres computadores, utilizando la media aritmética de los tiempos de ejecución normalizados al computador A (el tiempo de ejecución en A se toma como la unidad). Repetir usando los tiempos de ejecución normalizados al computador C.

C. Hacer las tres comparaciones globales posibles para los tres computadores, utilizando la media geométrica de los tiempos de ejecución normalizados al computador A. Repetir usando los tiempos de ejecución normalizados al computador C.

Solución ejercicio 1:
Aquí solo tenemos que comparar las cifras entre los computadores, cuanto menor sea, mejor velocidad tendrá.
Apartado A:
Computador A es 10 veces más rápido que el computador B ejecutando el programa P1.
Computador A es 20 veces más rápido que el computador B ejecutando el programa P1.
Computador B es 2 veces más rápido que el computador C ejecutando el programa P1.
Computador B es 10 veces más rápido que el computador A ejecutando el programa P2.
Computador C es 50 veces más rápido que el computador A ejecutando el programa P2.
Computador C es 5 veces más rápido que el computador B ejecutando el programa P2.

$$\begin{array}{l} \text{Ranking ejecutando P1: A, B, C.} \\ \text{Ranking ejecutando P2: C, B, A.} \end{array} \rightarrow$$

→Confuso, no es posible determinar cuál es el mejor computador.
Apartado B:
Damos valor 1 al computador que tiene mayor velocidad para realizar la media aritmética. Esto se realiza para normalizar la media.

$$M.A = \frac{1}{N}\sum_{j=1}^{N} x_j$$

A	B	C
1	5.05	10.01

Ranking: A, B, C.
Normalizando a C:

A	B	C
25.025	2.75	1

Ranking: C, B, A.
Apartado C:

$$M.G = \sqrt[n]{\prod_{j=1}^{n} x_j}$$

Normalizando a A:

A	B	C
1	1	0.63

C es 1.58 (1/0.63) veces más rápido que A y B.
Normalizando a C:

A	B	C
1.58	1.58	1

A y B son 1.58 veces más rápidos que C.

Ejercicio 2. En un computador con un procesador a 1600 MHz se compila y ejecuta el programa P que realiza 200000 operaciones en coma flotante. Existen dos versiones del procesador: una simple (Ent) sin hw de punto flotante y otra (HWPF) que sí incluye punto flotante. El compilador traduce las operaciones de punto flotante en instrucciones de PF, o en rutinas con instrucciones de enteros, según las opciones de compilación indicadas. Variando estas opciones se ha compilado P dos veces, generando dos códigos ejecutables: Phwpf (código para el procesador con hw de punto flotante) y Pent (código sin hw de punto flotante). Se han ejecutado los dos programas en los respectivos procesadores y se han obtenido las siguientes medidas:

Programa	Procesador	Tiempo de ejecución	CPI medio medido
Phwpf	HWPF	0.01 segundos	10 CPI
Pent	Ent	0.12 segundos	6 CPI

Se pide:

A. Los MIPS de las ejecuciones.

B. El número total de instrucciones ejecutadas en cada caso.

C. El número medio de instrucciones de enteros que sustituye cada operación en coma flotante en Pent.

D. Los MFLOPS obtenidos en la ejecución de Phwpf.

Solución ejercicio 2:
Primero nos dan la $f = 1600MHz$, seguidamente el problema nos dice que realiza 200000 operaciones y finalmente, nos indica que realiza dos procesos, uno en Ent que realiza las rutinas con instrucciones de enteros, y HWPF que realiza las instrucciones en PF.
Solución apartado A:
Para calcular los MIPS de las ejecuciones, necesitamos calcular el tiempo de ciclo ($t = 1/f$) y los MIPS:

$$\text{MIPS} = \frac{\text{n}^{\text{o}}\ \text{instr}}{T_{\text{CPU}}(s)} \times 10^{-6} = \frac{f \times 10^{-6}}{\text{CPI}} = \frac{f(\text{MHz})}{\text{CPI}}$$

Ahora sustituimos los datos del problema en la fórmula para calcular los MIPS de los dos procesadores:

$$\text{Phwpf} \rightarrow \text{MIPS} = \frac{1600}{10} = 160$$

$$\text{Pent} \rightarrow \text{MIPS} = \frac{1600}{6} = 266{,}7$$

Solución apartado B:
Para calcular el número de instrucciones para los dos procesadores vamos a utilizar la siguiente fórmula:

$$\text{N}^{\text{o}}\ \text{instr} = \text{MIPS} \times 10^6 \times T_{\text{CPU}}$$

$$\text{Phwpf} \rightarrow \text{N}^{\text{o}}\ \text{instr} = 160 \times 10^6 \times 0{,}01s = 1{,}6 \times 10^6 \quad \text{instrucciones}$$

$$\text{Pent} \rightarrow \text{N}^{\text{o}}\ \text{instr} = 266{,}7 \times 10^6 \times 0{,}12s = 32 \times 10^6 \quad \text{instrucciones}$$

Solución apartado C:
Para calcular el número medio de instrucciones enteras vamos a ir restando la frecuencia que tiene en un principio, en Hz, con las operaciones que realiza en un principio, seguidamente vamos a ir restando las operaciones con el resultado anterior, y finalmente dividimos lo que nos ha dado en la segunda operación con los procesos.

$$\text{Instrucciones enteras en Phwpf} \rightarrow 16000000 - 200000 = 14000000$$

$$\text{Operaciones en rutinas SW en Pent} \rightarrow 32000000 - 1400000 = 30600000$$

$$\text{Cada operación en PF es sustituida, en promedio} \rightarrow \frac{30600000}{200000} = 153$$

En total, se dan 153 instrucciones enteras.
Solución apartado D:
Para este apartado vamos a utilizar la definición de MFLOPS, que es la siguiente:

$$\text{MFPLOS} = \frac{\text{Nº op en PF}}{T_{\text{CPU}}} \times 10^{-6} = \frac{200000}{0{,}01} \times 10^{-6} = 20$$

Ejercicio 3. Se pretende comparar dos computadores utilizando cierto programa. Las velocidades que alcanzan son de 100 y 150 MIPS, respectivamente. Indica, si tienes suficientes datos, cual de los dos es el mejor:

A. Si los dos computadores tienen la misma arquitectura del juego de instrucciones, pero distinta frecuencia de reloj.

B. Si los dos computadores tienen la misma arquitectura del juego de instrucciones y frecuencia de reloj.

C. Si los dos computadores tienen distinto juego de instrucciones y la misma frecuencia de reloj.

Solución ejercicio 3:
Solución apartado A:
Con la misma arquitectura, el computador 2 ejecuta 1.5 veces más instrucciones que el 1, por tanto para este apartado es mejor el 2.
Solución apartado B:
Pasa lo mismo que en el apartado anterior, solo que a diferencia del otro, tienen la misma frecuencia. Pero aún así el computador 2 ejecuta más instrucciones que el 1, por tanto, sigue siendo mejor el 2.
Solución apartado C:
No tenemos datos suficientes porque, al tratarse de distintos repertorios de instrucciones no podemos concluir qué computador ejecuta más trabajo por unidad de tiempo.

Ejercicio 4. En cierta CPU, todas las instrucciones enteras se ejecutan en un ciclo de reloj, mientras que las de coma flotante necesitan 5 ciclos de reloj para completarse. La mayoría de los programas a ejecutar incluyen un 20 % de operaciones en coma flotante concentradas en la misma zona del código. Desde el punto de vista coste-prestaciones, ¿es interesante rediseñar la parte de punto flotante del procesador para que sea 5 veces más rápida a costa de duplicar el coste total de la CPU? Justifica la respuesta.

Solución ejercicio 4:
IMPORTANTE: CPI_{ent}=1; $\text{CPI}_{\text{FP}} = 5$ y también sabemos que realiza el 20 % de las operaciones en PF.

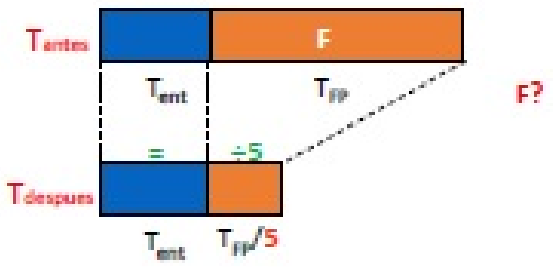


Figura 1.10: Esquema problema 4

Con este esquema en mente vamos a calcular la F, pero para ello vamos a calcular primero el T_{ent} y el T_{FP}:

$$T_{\text{ent}} = N_{\text{ent}} \times CPI_{\text{ent}} \times t \rightarrow T_{\text{ent}} = 0{,}8N \times 1 \times t \rightarrow T_{\text{ent}} = 0{,}8N \times t$$

$$T_{\text{FP}} = N_{\text{FP}} \times CPI_{\text{FP}} \times t \rightarrow T_{\text{FP}} = 0{,}2N \times 5 \times t \rightarrow T_{\text{FP}} = N \times t$$

Una vez lo hemos calculado, vamos a calcular la F con la siguiente fórmula:

$$F = \frac{T_{\text{FP}}}{T_{\text{FP}+T_{\text{ent}}}} = \frac{N \times t}{N \times t + 0{,}8 \times t} = \frac{N \times t}{N \times t(1 + 0{,}8)} = \frac{1}{1{,}8}$$

Ya tenemos calculada la F, ahora sólo tenemos que aplicarla a la fórmula de Speedup para ver si es interesante un rediseño de la computadora.

$$\text{Speedup} = \frac{1}{1 - F + \frac{F}{x}} = \frac{1}{1 - \left(\frac{1}{1{,}8}\right) + \frac{\left(\frac{1}{1{,}8}\right)}{5}} = 1{,}8$$

El Speedup es más pequeño que el coste de las operaciones (2), por lo que no es interesante el rediseño.
Hay otra manera de realizar el problema sin calcular la F y es aplicando la fórmula siguiente:

$$\text{Speedup} = \frac{T_{\text{antes}}}{T_{\text{desp}}} = \frac{T_{\text{emt}} + T_{\text{FP}}}{T_{\text{emt}} + \frac{T_{\text{FP}}}{5}}$$

Aplicando esta fórmula tenemos que llegar a la misma solución de arriba, sino llegas a esa solución revisa todas las cuentas para que estén en orden.

Ejercicio 5. El hw de punto flotante de un computador mejora en un factor de 5 el procesamiento de números en punto flotante. El tiempo de ejecución de cierto programa es de 1 minuto si se tiene hw de punto flotante, y de 2.5 minutos sin éste. Calcular el porcentaje del tiempo de ejecución que el programa está realizando operaciones en punto flotante is tiene que ejecutar dichas operaciones como llamadas a rutinas que implementan los cálculos en punto flotante mediante secuencias de operaciones enteras.

Solución ejercicio 5:
Primero, vamos a ordenar las ideas, sabemos el tiempo que tarda en procesar con y sin punteros, los cuales vamos a nombrar de la siguiente forma:

$$T(\text{con HW de PF}) = 60s; T(\text{sin HW de PF}) = 150s$$

Una vez que hemos puesto los tiempos de procesamiento, vamos a hacer un dibujo para calcular el Speedup, y posteriormente la F.

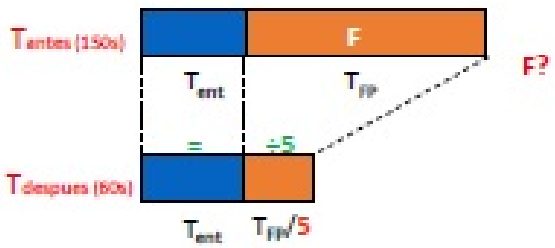


Figura 1.11: Esquema problema 5

Ahora solo tenemos que aplicar la fórmula de Speedup que hemos utilizado arriba y despejar la F, y ya tenemos la solución a nuestro problema.

$$\text{Speedup} = \frac{T_{\text{antes}}}{T_{\text{despues}}} = \frac{150}{60} = 2{,}5 = \frac{1}{1 - F + F/x} = \frac{1}{1 - F + F/5} \rightarrow F = 0{,}75 \rightarrow 75\,\%$$

Ejercicio 7. Un computador valorado en 1200€ reparte su tiempo entre dos aplicaciones A1 y A2 (20 % y 80 % del tiempo, respectivamente). Por su parte, la aplicación A1 dedica un 10 % de su tiempo a operaciones de E/S, un 20 % a la ejecución de instrucciones en punto flotante y el resto a la ejecución de operaciones enteras. La aplicación A2 dedica el 40 % de su tiempo a operaciones E/S, y el resto a la ejecución de instrucciones enteras. Se planea sustituir un elemento del subsistema E/S para mejorar en un 150 % sus prestaciones, así como cambiar el hw de punto flotante por otro el doble de rápido. Desde el punto de vista de la relación coste/prestaciones, ¿cuánto pagarías por el conjunto de estas dos mejoras?

Solución ejercicio 7:
Sabemos que tenemos dos aplicaciones distintas, con las que se reparte el tiempo de una manera distinta. Y distribuimos ese tiempo, entre el sistema E/S, el punto flotante y el entero. Y lo veríamos así:
A1(20 %):

- 10 % E/S.
- 20 % FP.
- 70 % ENT.

A2(80 %):

- 40 % E/S.
- 60 % ENT.

Distribuimos el 100 % de la aplicación entre distintos subsistemas. Una vez lo tenemos claro, vamos a ver cómo se definiría el Speedup en el caso de dos aplicaciones:

$$\text{Speedup} = \frac{T_{\text{antes}}}{T_{\text{despues}}} = \frac{1}{1 - (F_1 + F_2) + \frac{F_1}{x_1} + \frac{F_2}{x_2}}$$

Ahora vamos a calcular F_1, x_1, F_2 y x_2. Los dos primeros se mejoran con E/S y los dos últimos con FP. Y lo calculamos de la siguiente forma:

$$\begin{aligned} F_1 &= 0{,}2 \times 0{,}1 + 0{,}8 \times 0{,}4 = 0{,}34 \\ x_1 &= 150\,\% \rightarrow x_1 = 2{,}5 \\ F_2 &= 0{,}2 \times 0{,}2 = 0{,}04 \\ x_2 &= 2 \end{aligned}$$

$$\text{Speedup} = \frac{1}{1 - (0{,}34 + 0{,}04) + \frac{0{,}34}{2{,}5} + \frac{0{,}04}{2}} = 1{,}29$$

$$\text{Speedup} = 1{,}29 = \frac{\text{Coste}_{\text{despues}}}{\text{Coste}_{\text{antes}}} \rightarrow \text{Coste}_{\text{despues}} = 1{,}29 \times 1200 = 1548e$$

Se pagarían 1548-1200=348€ por las mejoras.

Ejercicio 8. En cierto computador se plantea mejorar tres componentes del mismo que no se utilizan simultáneamente en el tiempo. Los factores de mejora son 5, 7 y 10, respectivamente. Los dos primeros componentes se usan durante el 20 % y el 25 % del tiempo, respectivamente. ¿Qué fracción del tiempo debe utilizarse el tercer componente para alcanzar una aceleración global de 5?

Solución ejercicio 8:

Para calcular la fracción F_3 debemos utilizar la definición de Speedup, para ello, vamos a definir los parámetros del Speedup aplicados al problema:

$$\begin{aligned} x_1 &= 5 \rightarrow F_1 = 20\,\% \\ x_2 &= 7 \rightarrow F_2 = 25\,\% \\ x_3 &= 10 \end{aligned}$$

$$\text{Speedup} = \frac{T_{\text{antes}}}{T_{\text{despues}}} = \frac{1}{1 - (F_1 + F_2 + F_3) + \frac{F_1}{x_1} + \frac{F_2}{x_2} + \frac{F_3}{x_3}}$$

$$5 = \frac{1}{1 - (0{,}2 + 0{,}25 + F_3) + \frac{0{,}2}{5} + \frac{0{,}25}{7} + \frac{F_3}{10}}$$

Despejamos F_3 de la fracción del Speedup y nos da lo siguiente:

$$F_3 = 0{,}473 \rightarrow F_3 = 47{,}3\,\%$$

Ejercicio 9. El propietario de un PC utiliza un conjunto de aplicaciones que trabajan bajo MS-Windows y se encuentra insatisfecho con la velocidad del entorno gráfico. Antes de tomar ninguna decisión, monitoriza el uso que el procesador hace de la controladora de vídeo, y descubre que el procesador ocupa el 20 % de su tiempo en acceder a la RAM de vídeo, y el 2 % en operaciones de entrada salida sobre los puertos de su controladora. En una tienda de ordenadores le proporcionan documentación sobre una nueva tarjeta controladora que asegura que con ella el acceso a los puertos. Si el PC costó 1000 euros y su propietario es partidario del análisis de coste/prestaciones, ¿cuánto dinero podría gastarse como máximo en sustituir la controladora sin degradar esta relación?

Solución ejercicio 9:
Identificamos la $F_1 = 0{,}2$ y $F_2 = 0{,}02$, que estos son obtenidos de los porcentajes en el tiempo. Y seguidamente, nos dice que el tiempo se reduce a la mitad, que es $x_1 = 2$, y posteriormente, que el tiempo se duplica, por lo que empeora respecto al anterior y sería $x_2 = \frac{1}{2}$. Una vez los tenemos, los pasamos a la fórmula de Speedup:

$$\text{Speedup} = \frac{T_{\text{antes}}}{T_{\text{despues}}} = \frac{1}{1-(F_1+F_2)+\frac{F_1}{x_1}+\frac{F_2}{x_2}} = \frac{1}{1-(0{,}2+0{,}02)+\frac{0{,}2}{2}+\frac{0{,}02}{\frac{1}{2}}} =$$

Relacionamos el Speedup con el coste posterior y sería lo siguiente:

$$\text{Speedup} = 1{,}087 = \frac{\text{Coste}_{\text{despues}}}{\text{Coste}_{\text{antes}}} \rightarrow \text{Coste}_{\text{despues}} = 1{,}087 \times 1000 = 1087e$$

Se pagarían 1087-1000=87€ por la nueva tarjeta.

Ejercicio 10. La compañía FILIPS, que fabrica equipos de sonido de altas prestaciones, dispone de un laboratorio de diseño de altavoces equipado con un computador, valorado en 1000 euros, y la aplicación PAM versión 1.0 valorada en 1000 euros. La estructura de la aplicación es la siguiente:

```
Repeat
        Generar_modelo;
        Simular_comportamiento;
        Analizar_resultados;
Until false;
```

Los tres subprogramas hacen un uso intensivo de la aritmética en coma flotante. Se ha monitorizado el sistema y se ha medido, para cada subprograma, la fracción de tiempo en que se está ejecutando y la productividad del procesador.

Módulo	**Fracción de tiempo**	**Mflops**
Generar modelo	30 %	80
Simular comportamiento	50 %	100
Analizar resultados	20 %	70

El laboratorio necesita aumentar la productividad, como mínimo, un 100 % y dispone de 1500 euros de presupuesto. Existen los siguientes productos en el mercado:

- El acelerador DSPX (400 euros), que es un dispositivo acelerador hardware que ejecuta Simular comportamiento a 200 MFLOPS y puede ser utilizado por PAM 1.0. Durante la ejecución del subprograma, la aplicación queda esperando mientras DSPX realiza todos los cálculos.
- La actualización de PAM a la versión 2.0 (300 euros), que es capaz de solapar la ejecución de Generar modelo y Analizar resultados con la ejecución de Simular comportamiento sobre el acelerador DSPX. Sin DSPX se comporta exactamente igual que la versión 1.0.
- La licencia adicional (200 euros) para cualquiera de las dos versiones de la aplicación PAM, para que se ejecute en cualquier ordenador de las mismas prestaciones (y al mismo precio) que el actual. Con ello se pueden crear nuevos puestos de trabajo y repartir los datos a procesar.

Nótese que hay varias combinaciones de mejora posible. Se pide:

A. Decidir cuál es la inversión que maximiza la relación productividad/coste, siempre dentro de las restricciones.

B. Calcular el número de operaciones de coma flotante ejecutadas (en una iteración del programa) para crear un modelo simulable sabiendo que la operación Generar modelo requiere 2.5 segundos.

Solución ejercicio 10:
Solución apartado A:
Primero valoramos el Speedup y si es aprovechable en un computador.
Del 50 % de la aplicación vemos que duplica la velocidad (100-200 MFLOPS). De aquí sacamos que $F = 0{,}5$ y $x = 2$:

$$\text{Speedup} = \frac{1}{0{,}5 + \frac{0{,}5}{2}} = 1{,}33$$

Como se solapan los procesos Generar modelo y Analizar resultados, el tiempo se reduce a la mitad. Con esto procedemos a calcular el Speedup:

$$\text{Speedup} = \frac{T_{\text{sin mejora}}}{T_{\text{mejora}}} = 2$$

Ahora pasamos todos estos datos a la tabla para tenerlo más claro:

Configuración	**Coste**	**Speedup**	**Rel Speedup/Coste**	**Aprovechable**
Base (PAM1)	2000	1	$1/2000 = 5 \times 10^{-4}$	No (Speedup=1)
PAM1+DSPX	2400	1.33	$1{,}33/2400 = 5{,}4 \times 10^{-4}$	No (Speedup=1.33)
PAM2+DSPX	2700	2	$2/2700 = 7{,}4 \times 10^{-4}$	Si (Speedup=2)

Ahora realizamos el mismo proceso aplicado a dos computadores:

Configuración	Coste	Speedup	Rel Speedup/Coste	Aprovechable
2 Comp+PAM1+ Lic adicional	3200	2	$2/3200 = 6{,}3 \times 10^{-4}$	Si (Speedup=2)
2 Comp+PAM1+ 2xDSPX+ Lic adicional	4000	2.66	$2{,}66/4000 = 6{,}7 \times 10^{-4}$	No (gasto>1500)
2 Comp+PAM2+ 2xDSPX+ Lic adicional	4300	4	$4/4300 = 9 \times 10^{-4}$	No (gasto>1500)

Los precios se han obtenido sumando los precios de los distintos componentes de la computadora.
Concluimos que con tres computadores se nos queda fuera de presupuesto, y concluimos que con dos computadores la mejor opción es la 1 por su coste y la relación speedup/coste.
Solución apartado B:
Para calcular el tiempo total, se dividen los segundos entre el tiempo que consume:

$$T = \frac{2{,}5}{0{,}3} = 8{,}33s$$

Módulo	Tiempo (s)	MFLOPS	Operaciones
Gen Modelo (30 %)	2.5	80	$2{,}5 \times 80 \times 10^6 = 200 \times 10^6$
Sin Comportam (50 %)	$8{,}33 \times 0{,}5 = 4{,}17$	100	$4{,}17 \times 100 \times 10^6 = 417 \times 10^6$
Analizar resul (20 %)	$8{,}33 \times 0{,}2 = 1{,}67$	70	$1{,}67 \times 70 \times 10^6 = 117 \times 10^6$
TOTAL			734×10^6 operaciones

Ejercicio 11. Un computador tiene un CPI de 4 para las instrucciones de coma flotante, excepto la raíz cuadrada que tiene 20, y de 1.33 para el resto. Cierto benchmark tiene un 25 % de operaciones en coma flotante (sin incluir la raíz cuadrada) y un 2 % de raíces cuadradas. Con el propósito de mejorar las prestaciones del computador, el equipo de diseño maneja dos alternativas que suponen el mismo coste:

A. Modificar la organización para aumentar la frecuencia de reloj en un 15 % a coste de aumentar a 23 los CPI de las instrucciones de raíz cuadrada.

B. Mejorar la organización del hardware que realiza las operaciones en coma flotante (sin incluir la raíz cuadrada) para reducir a 2 los CPI. ¿Qué opción es la más acertada?

Solución ejercicio 11:
Los datos que obtenemos del problema son CPI_{FP}=4; $CPI_{RC} = 20$; $CPI_{resto} = 1{,}33$. El 25 % de operaciones es en PF y 2 % de raíces cuadradas. Primero calculamos el $CPI_{original}$:

$$CPI_{\text{original}} = 0{,}25 \times 4 + 0{,}02 \times 20 + 0{,}73 \times 1{,}33 = 2{,}37$$

Solución apartado A:

$$CPI_a = 0{,}25 \times 4 + 0{,}02 \times 23 + 0{,}73 \times 1{,}33 = 2{,}43; f_a = 1{,}15 \times f_{\text{original}}$$

$$\text{Speedup}_a = \frac{T_{\text{antes}}}{T_{\text{despues}}} = \frac{N_{\text{original}\times\text{CPI}_{\text{original}}\times t_{\text{original}}}}{N_a \times \text{CPI}_a \times t_a} = \frac{\text{CPI}_{\text{original}\times t_{\text{original}}}}{\text{CPI}_a \times t_a} =$$

$$= \frac{2{,}37 \times 1/f_{\text{original}}}{2{,}43 \times \left(\dfrac{1}{1{,}15 \times f_{\text{original}}}\right)} = 1{,}12$$

Solución apartado B:

$$\text{CPI}_b = 0{,}25 \times 2 + 0{,}02 \times 20 + 0{,}73 \times 1{,}33 = 1{,}87; t_b = t_{\text{original}}$$

$$\text{Speedup}_b = \frac{T_{\text{antes}}}{T_{\text{despues}}} = \frac{N_{\text{original}\times\text{CPI}_{\text{original}}\times t_{\text{original}}}}{N_b \times \text{CPI}_b \times t_b} = \frac{\text{CPI}_{\text{original}}}{\text{CPI}_b} = \frac{2{,}37}{1{,}87} = 1{,}26$$

$\text{Speedup}_b > \text{Speedup}_a \rightarrow$ La opción b es la mejor.

Capítulo 2

Tema 2: ILP, Planificación dinámica, Predicción de saltos, Especulación

En este capítulo vamos a ver los siguientes conceptos.

- Introducción: ILP.
- Técnicas SW: Compilador.
- Planificación dinámica. Algoritmo de Tomasulo.
- Tratamiento de dependencias de control: Predicción de saltos.
  - Tratamiento de saltos.
  - Técnicas de predicción: Estáticas y dinámicas.
  - Explicación de ejercicios resueltos.
- Especulación.

2.1. Introducción: ILP

El objetivo es ejecutar el mayor número de instrucciones por ciclo.

IMPORTANTE

CPI=CPIideal+Penaliz. Media por Instr. (paradas "pipe").
Obtener el máximo numero de instrucciones independientes.

2.1.1. ¿Qué técnicas conocemos?

- Conflictos de recursos-Riesgos estructurales→ Replicación/segmentación.
- Dependencias de datos → Cortocircuitos.
- Dependencias de control (un salto cada 4-7 instrucciones)→ Saltos retardados.

2.1.2. Mecanismos para explotar ILP

Basados en HW en tiempo de ejecución (dinámicos). Ej: Intel Core, AMD, IBM.

- Toda la información disponible en ejecución.
- Código independiente de la implementación.

Basados en SW en tiempo de compilación (estáticos). Ej: Itanium.
Dependencias de memoria muy díficiles de determinar.

2.1.3. Paralelismo a nivel de instrucción ILP

Es la técnica consistente en explotar paralelismo entre instrucciones próximas en la secuencia.
El bloque básico:

- Es una secuencia de código sin saltos. Un solo punto de entrada y salida.
- Típicamente muy pequeño: solo de 4 a 7 instrucciones.
- Con fuertes dependencias entre ellos.

El camino es explotar ILP entre varios BB.
El caso más simple es el paralelismo a nivel de bucle. Todas las iteraciones son independientes (saltos).

2.1.4. Técnicas para explotar ILP

	Técnica	Reduce
Dinámicas	Planificación Dinámica	Paradas por riesgos de datos
	Predicción dinámica de saltos	Paradas por riesgos de control
	Lanzamiento múltiple Varias instrucciones por ciclo	CPI Ideal
	Especulación	Riesgos de datos y control
	Dynamic memory disambiguation	Paradas por riesgos de datos en memoria
Estáticas	Desenrollado de bucles	Paradas por riesgos de control
	Planificación por el compilador	Paradas por riesgos de datos
	Software pipelining	CPI Ideal y Paradas por riesgos de datos
	Predicción estática y Especulación por el Compilador	CPI Ideal, paradas por riesgos de datos y control

2.1.5. Dependencias

IMPORTANTE

Determinar las dependencias es crítico para obtener el máximo paralelismo.

Las dependencias son propias de los programas:

- La presencia de una dependencia indica la posibilidad de aparición de un riesgo, pero la aparición de éste y la posible parada depende de las características "pipe".
- Indican la posibilidad de un riesgo.
- Determinan el orden de cálculo de los resultados.
- Imponen un límite al paralelismo que es posible obtener.

Tipos de Dependencias

- Dependencias de datos:
  - Dependencia verdadera (LDE).
  - Dependencias de nombre:
    - Antidependencia (EDL).
    - Dependencia de salida (EDE).
- Dependencias de control.

Dependencia verdadera (LDE)

La instrucción j depende de i.

- i produce un resultado que usa j.
- j depende de k y k depende de i.

```
i:    L.D     F0,0(R1)
j:    ADD.D   F4,F0,F2
```

Figura 2.1: Ejemplo de LDE

Dependencias de nombre (Reutilización de los registros)

Dos instrucciones i y j donde i precede a j presentan dependencias de nombre en las siguientes instrucciones:

- Antidependencia WAR (EDL): La instrucción j escribe (Reg o memoria) antes de que i lea.
- Dependencia de salida WAW (EDE): Las instrucciones i y j escriben el mismo reg. o memoria.

```
ADD.D   F4,F0,F2
L.D     F0,-8(R1)
```

Figura 2.2: Antidependencia WAR

```
ADD.D   F4,F0,F2
SUB.D   F4,F3,F2
```

Figura 2.3: Dependencia de salida WAW

ILP y Dependencias de datos

- Los mecanismos de ejecución deben perservar el comportamiento del programa. **Mismo resultado que en ejecución secuencial**.
- Explotar todo el paralelismo posible sin afectar al resultado de la ejecución.
- Para las dependencias de nombre eliminar la dependencia usando otros "nombres".

Dependencias de control

Cada instrucción depende de un conjunto de saltos y en general esta dependencia debe preservarse para perservar el comportamiento del programa.

```
If P1 (
        S1;
);
if P2 (
        S2;
)
```

Figura 2.4: S1 depende de P1: S2 depende de S2

Ejecutar instrucciones no debidas **si esto no afecta** al resultado correcto del programa.

> IMPORTANTE
>
> El comportamiento de las excepciones y el flujo de datos deben preservase.

Dependencias de control y Excepciones

Comportamiento de excepciones se debe preservar. Cualquier cambio en el orden de ejecución no debe cambiar cómo las excepciones son atendidas en la ejecución.

```
        DADDU   R2,R3,R4
        BEQZ    R2,L1
        LW      R1,0(R2)
L1:     ---     ---
```

LW no se puede mover antes de BEQZ (posible fallo de página).

Dependencias de control y flujo de datos

Se debe mantener el flujo de datos entre instrucciones productoras y consumidoras de datos.

```
        DADDU   R1,R2,R3
        BEQZ    R4,L1
        DSUBU   R1,R5,R6
L1:     ---     ---
        OR      R7,R1,R8
```

OR usa el resultado de DADDU o DSUBU dependiendo del comportamiento del salto. El flujo de datos se debe preservar.

Dependencias de datos

Fáciles de determinar para registros.
Difíciles para direcciones de memoria.
Implica:

- Más registros para evitar dependencias de nombre.
- Identificar dependencias entre load y store para permitir su reordenación.

Dependencias de control

En general:

- Una instrucción dependiente de un salto no puede moverse antes del salto.
- Una instrucción no dependiente de un salto no puede moverse después del salto.

Efecto de las dependencias de control sobre el orden de las excepciones y el flujo de datos.

SOLUCIÓN
HW+SW (PROCESADOR+ COMPILADOR)

2.1.6. Técnicas SW para explotar ILP

Modelo de computador para el estudio:

- Lanzamiento a ejecución en orden. Detección de riesgos en ID. Si riesgo, congelar instrucción en ID.
- Saltos resueltos en ID.
- Etapa de ejecución con varios UFs de diferente duración. Posibilidad de finalización en desorden.
    - EX: load, store, int.
    - FP: cálculos en FP (segmentación en 4 etapas).
- Bancos de registros separados para INT y FP.

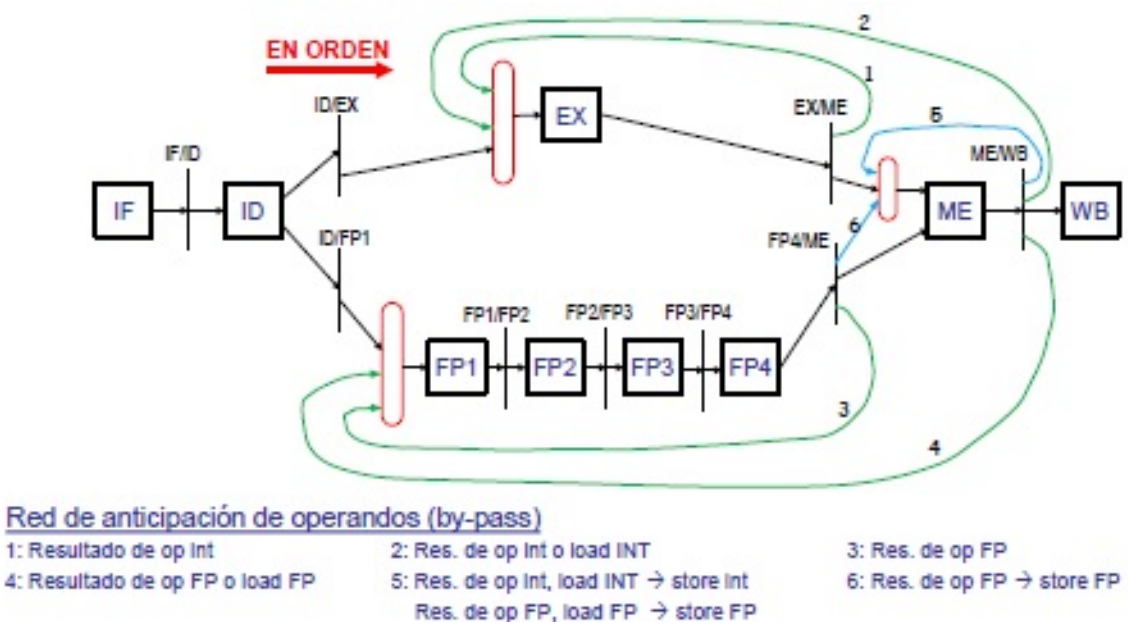


Figura 2.5: Modelo computador

Latencia de uso: número mínimo de ciclos entre una instrucción que produce un resultado y otra que lo consume.

Instrucción que produce un resultado	**Instrucción que usa el resultado**	**Latencia de uso**
FP aritmética	FP aritmética	3
FP aritmética	STORE FP	2
LOAD FP	FP aritmética	1
LOAD FP	STORE FP	0
LOAD Int	Int aritmética	1
Int aritmética	Int aritmética	0

Un programa se da en un bucle simple como:

```
for ( i =1000; i > 0; i--)
    x (i) = x (i) + s ;
```

Código maquina MIPS

```
Loop: L.D      F0,0(R1)     ; F0← elemento array
      ADD.D    F4,F0,F2     ; sumar escalar en F2
      S.D      F4,0(R1)     ; almacenar resultado
      DADDIU   R1,R1,#-8    ; decrementar puntero
      BNE      R1,R2,Loop   ; repetir si R1 ≠ R2
```

Ciclo lanzamiento			1	2	3	4	5	6	7	8	9	10
L.D	F0,0(R1)	IF	ID	EX(1)	ME	WB						
ADD.D	F4,F0,F2		IF	ID	ID	FP1	FP2	FP3	FP4	ME(2)	WB	
S.D	F4,0(R1)			IF	IF	ID	ID	ID(3)	EX(1)	ME		
DADDIU	R1,R1,#-8					IF	IF	IF	ID	EX	ME	WB
BNE	R1,R2,Loop								IF	ID	ID	DS(4)

(1) Cálculo de la dirección de memoria efectiva
(2) Pasa por la etapa ME, pero no hay un acceso real a memoria
(3) El SD no avanza mientras su R. fuente coincida con alguno de los R. destino en FP1 o FP2. Si la coincidencia es con FP3, el SD ya puede avanzar a EX, cuando llegue a ME recibirá el operando por forwarding desde FP4 (camino 6).
(4) Delay Slot

Figura 2.6: Ejemplo de programa

Rendimiento en una iteración

			Ciclo de lanzamiento	
Loop:	L.D	F0,0(R1)	1	
	Espera		2	; ADD.D bloqueado en ID
	ADD.D	F4,F0,F2	3	
	Espera		4	; S.D bloqueado en ID
	Espera		5	; S.D bloqueado en ID
	S.D	F4,0(R1)	6	
	DADDIU	R1,R1,#-8	7	
	Espera		8	; BNE bloqueado en ID
	BNE	R1,R2,Loop	9	
	Espera		10	; Salto retardado

- El rendimiento se degrada por las dependencias, a pesar del by-pass de operandos.
- El CPI ideal de 1 se duplica (10 ciclos para lanzar 5 instrucciones): CPI=2.

Planificación de instrucciones: Reordenar para ocultar latencias.

```
Loop:   L.D      F0,0(R1)        Ciclo 1
        DADDIU   R1,R1,#-8            2
        ADD.D    F4,F0,F2             3
        Espera                        4
        BNE      R1,R2,Loop           5
        S.D      F4,8(R1)             6
```

- Reordenamiento para ocultar latencias.
- 6 ciclos por iteración CPI $\sim$ 1.
- 2 ciclos de overhead por el salto.

Desenrollado 4 veces para más paralelismo (elimina saltos).

```
Loop:   L.D      F0,0(R1)
        ADD.D    F4,F0,F2
        S.D      F4,0(R1)
        L.D      F6,-8(R1)
        ADD.D    F8,F6,F2
        S.D      F8,-8(R1)
        L.D      F10,-16(R1)
        ADD.D    F12,F10,F2
        S.D      F12,-16(R1)
        L.D      F14,-24(R1)
        ADD.D    F16,F14,F2
        S.D      F16,-24(R1)
        DADDIU   R1,R1,#-32
        BNE      R1,R2,Loop
```

- Expone más paralelismo y elimina saltos.
- Se elimina 3 saltos y 3 decrementos.
- Permanecen dependencias y paradas: Hallar ciclos por iteración y CPI.
- Más registros: Renombrado por el Compilador.

Desenrollado+ Planificación.

```
Loop:   L.D      F0,0(R1)
        L.D      F6,-8(R1)
        L.D      F10,-16(R1)
        L.D      F14,-24(R1)
        ADD.D    F4,F0,F2
        ADD.D    F8,F6,F2
        ADD.D    F12,F10,F2
        ADD.D    F16,F14,F2
        S.D      F4,0(R1)
        S.D      F8,-8(R1)
        DADDIU   R1,R1,#-32
        S.D      F12,16(R1)        ; 16-32= -16
        BNE      R1,R2,Loop
        S.D      F16,8(R1)         ;  8-32 = -24
```

- Mover S.D después de DADDIU: ojo al valor de R1.
- 14 instrucciones lanzadas en 14 ciclos: 3.5 ciclos por iteración.
- CPI=1.
- Más registros.

IMPORTANTE

El compilador planifica para minimizar los riesgos y eliminar las paradas "pipe".

Software "pipelining"

Si las diferentes iteraciones de un bucle son independientes, tomemos instrucciones de diferentes iteraciones para aumentar el ILP.
Reorganiza los bucles de manera que cada instrucción pertenece a una iteración diferente.
Ejemplo de pseudo-código de alto nivel: Procesar un vector de 126 componentes.

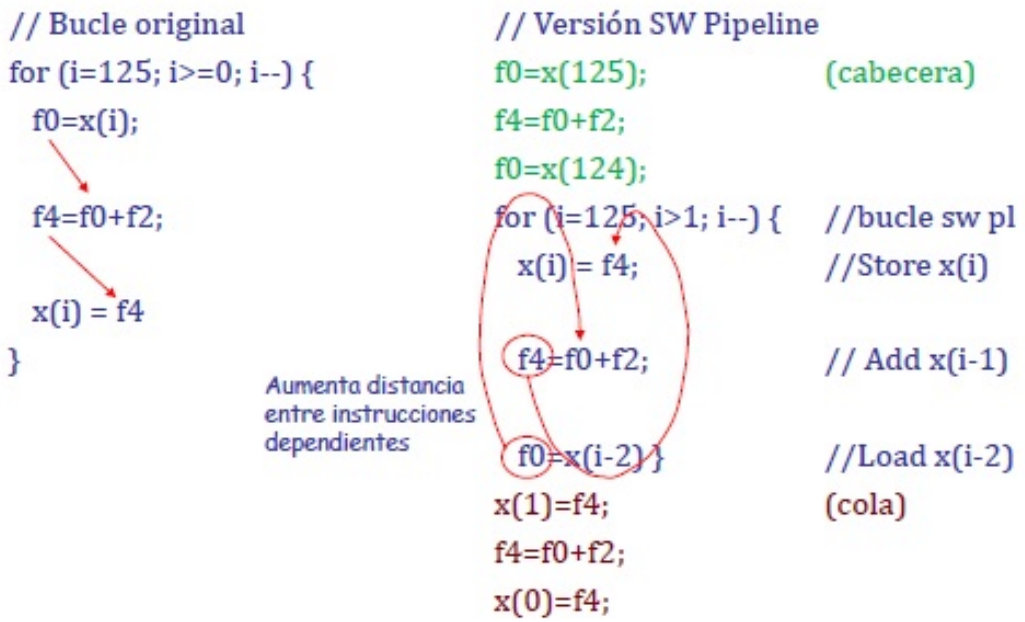


Figura 2.7: Ejemplo

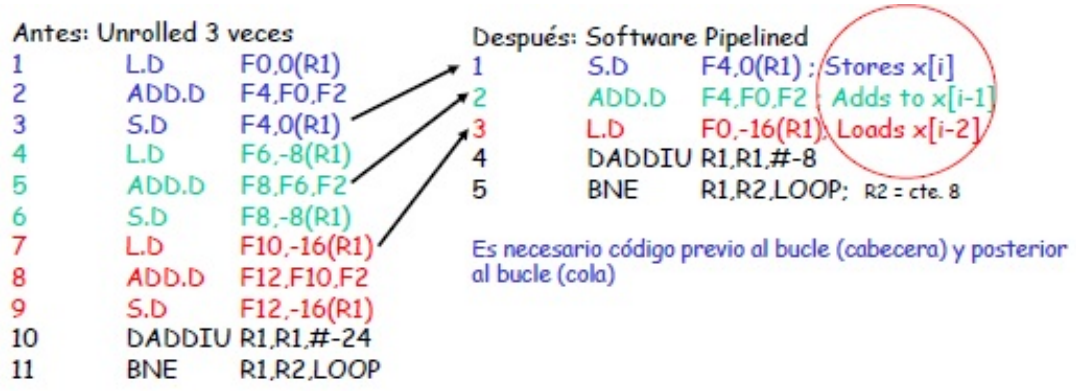


Figura 2.8: Ejemplo

Loop unrolling simbólico:

- Maximiza la distancia resultado-uso.
- Menor tamaño del código.

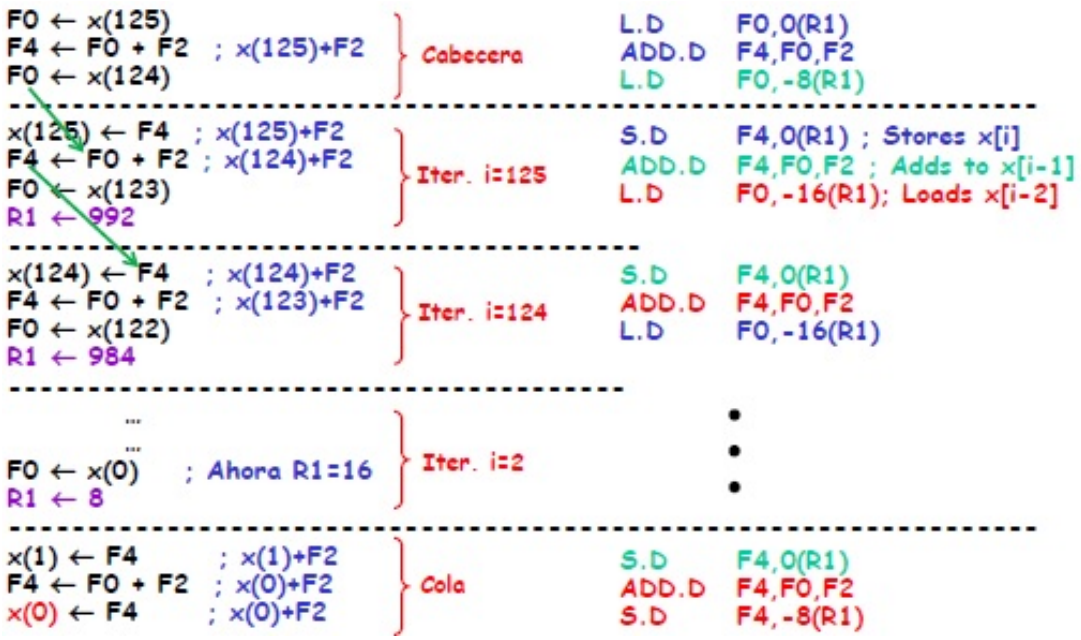


Figura 2.9: Ejecución SW pipelined: R1=1000

Comparación:
Loop Unrolling:

- Bloque grande para planificar.
- Reduce el numero de saltos.
- Incrementa el tamaño del código.
- Tiene que incluir iteraciones extra.
- Presión sobre el uso de registros.

Software Pipelining:

- No hay dependencias en el cuerpo del bucle.
- No reduce el numero de saltos.
- Necesita inicio y finalización especial.

2.2. Tratamientode dependencias de datos en ejecución

2.2.1. Planificación dinámica: Procesador

Modifica la secuencia de instrucciones resolviendo las dependencias en tiempo de ejecución. Disponibilidad de más unidades funcionales. Código válido para diferentes implementaciones.

Problema

Lanzamiento de instrucciones en orden.

```
DIV.D   F0,F2,F4        S1
ADD.D   F10,F0,F8       S2  S2 depende de S1
SUB.D   F12,F8,F14      S3  S3 es independiente de las demás
```

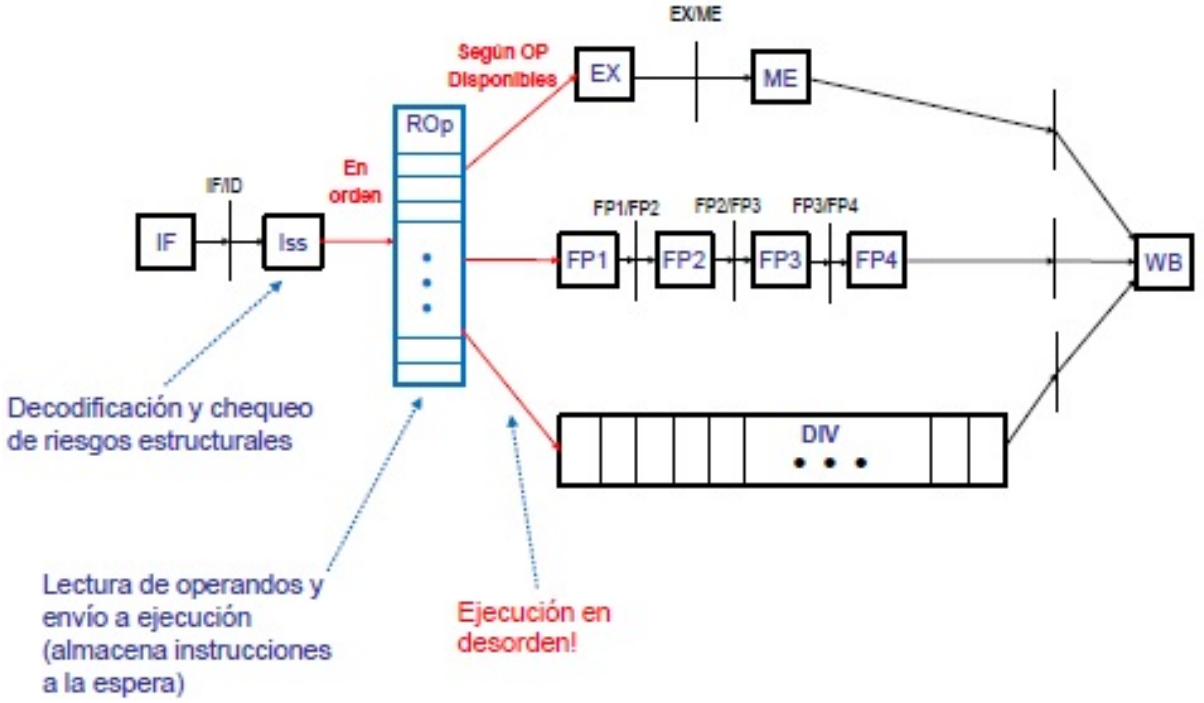
Solución

Dividir la etapa ID en dos etapas diferenciadas.

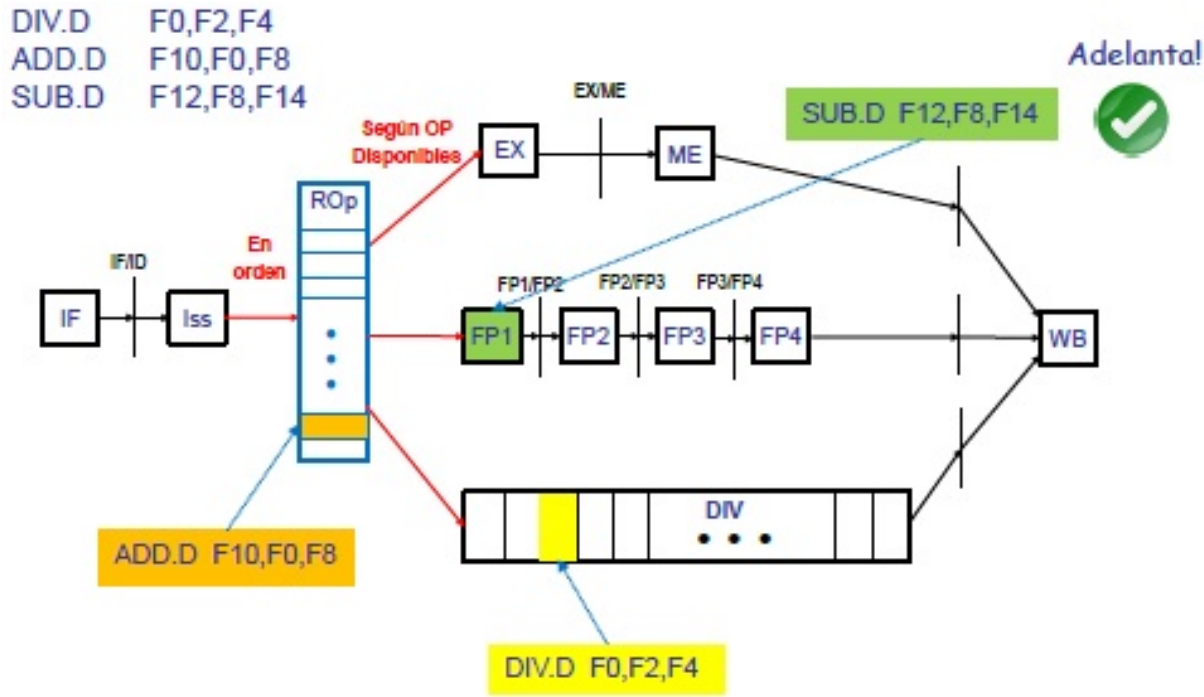
- **Issue**: Decodifica y chequea riesgos estructurales.
- **Lectura de operandos**: Chequea disponibilidad de operandos. Debe implementarse para permitir el flujo de instrucciones.

Ejecución fuera de orden → Finalización fuera de orden.

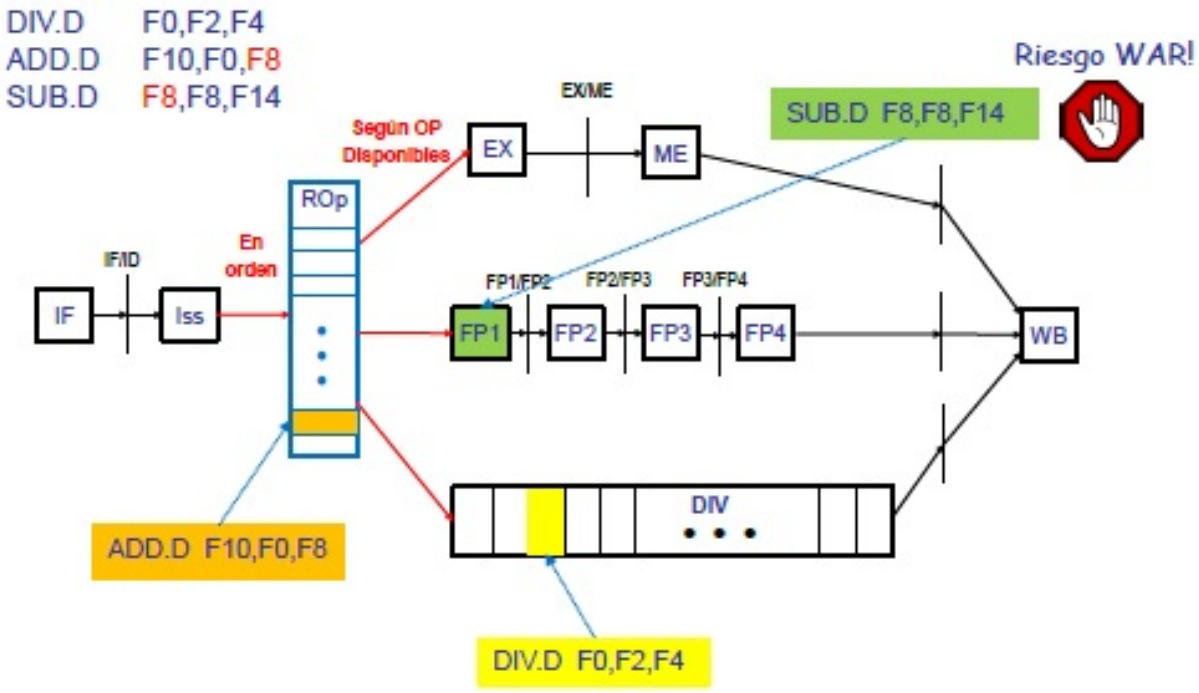
Etapas Issue (Iss) y Lectura de Operandos (ROp) separadas



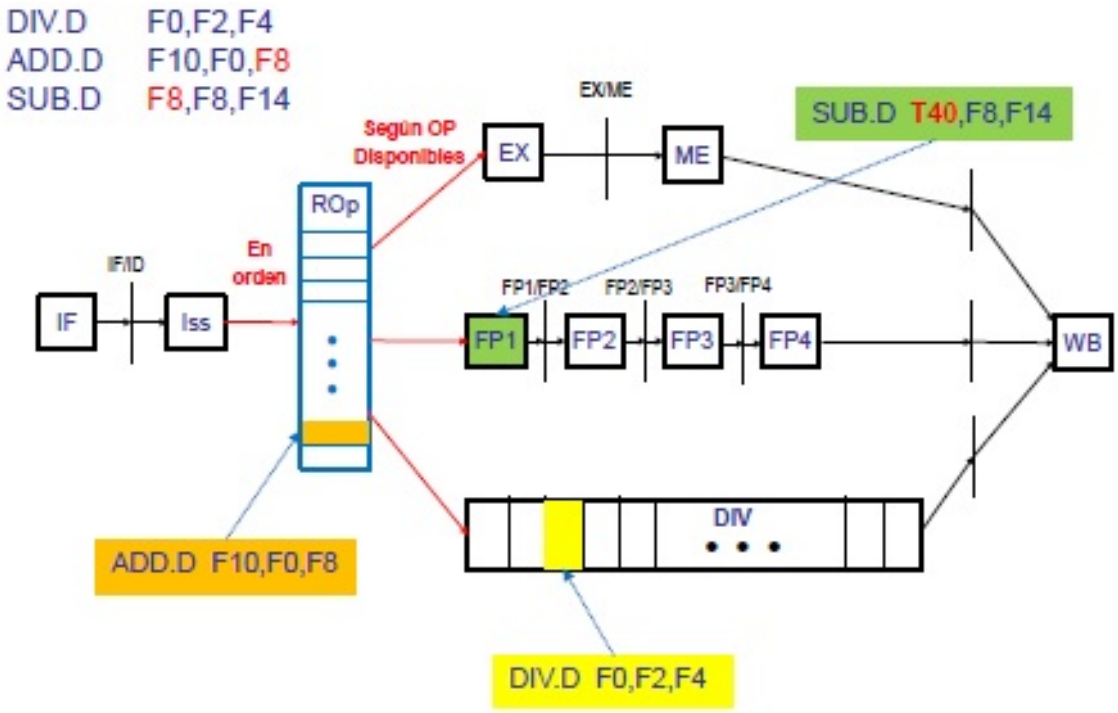
Ejecución de la secuencia



SUB.D acabará antes que DIV.D y que ADD.D: Se evita el bloqueo.
Pero si la secuencia cambia pasará lo siguiente:

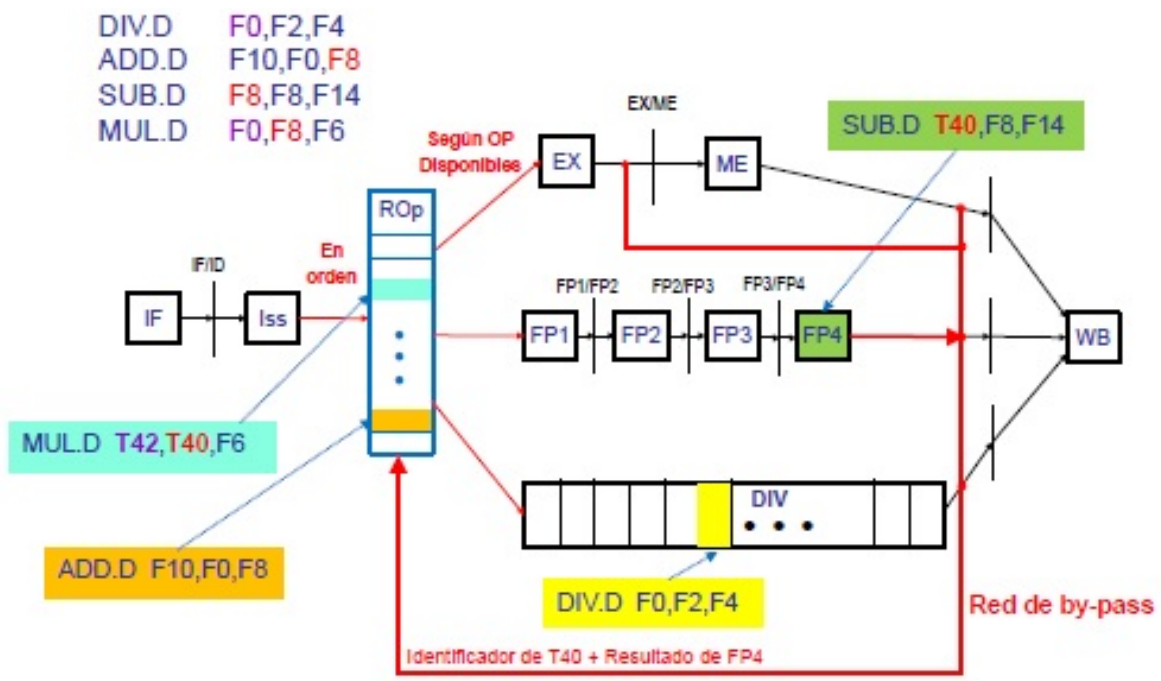


Si SUB.D acaba antes de que ADD.D lea sus operandos, destruirá el valor de F8.



¿Podría el Hw suplantar el reg destino de SUB.D?
Renombramiento Dinámico.

Las instrucciones posteriores al SUB.D que tengan F8 como operando, por supuesto deben usar el valor almacenado en T40.
Volvemos a cambiar la secuencia para ver qué ocurre:



¿Podrían las instrucciones en etapa ROp leer los operandos en cuanto estén calculados?
By-pass de operandos.

Las instrucciones en la etapa ROp pueden obtener sus operandos a través de la red de by-pass. En el ejemplo, MULD está a la espera de T40.
Una instrucción posterior a MULD que referencia FO, leerá el resultado de MULD, incluso aunque DIV.D acabe más tarde.
Esfuerzos desde los años 60:

- Scoreboard (CDC 6600):
  - Limita el efecto de dependencias RAW (no bloqueo en lanzamiento).
- Algoritmo de Tomasulo:
  - También limita el efecto de dependencias RAW.
  - Además, introduce el concepto de "renombramientodinámica de registros"→ Elimina el efecto de dependencias WAR y WAW.
  - Complejidad hardware considerable para la época: éxito modesto.
  - Idea implementada (con diversos matices): en casi todos los μ-procesadores modernos.

2.2.2. Planificación Dinámica: Tomasulo

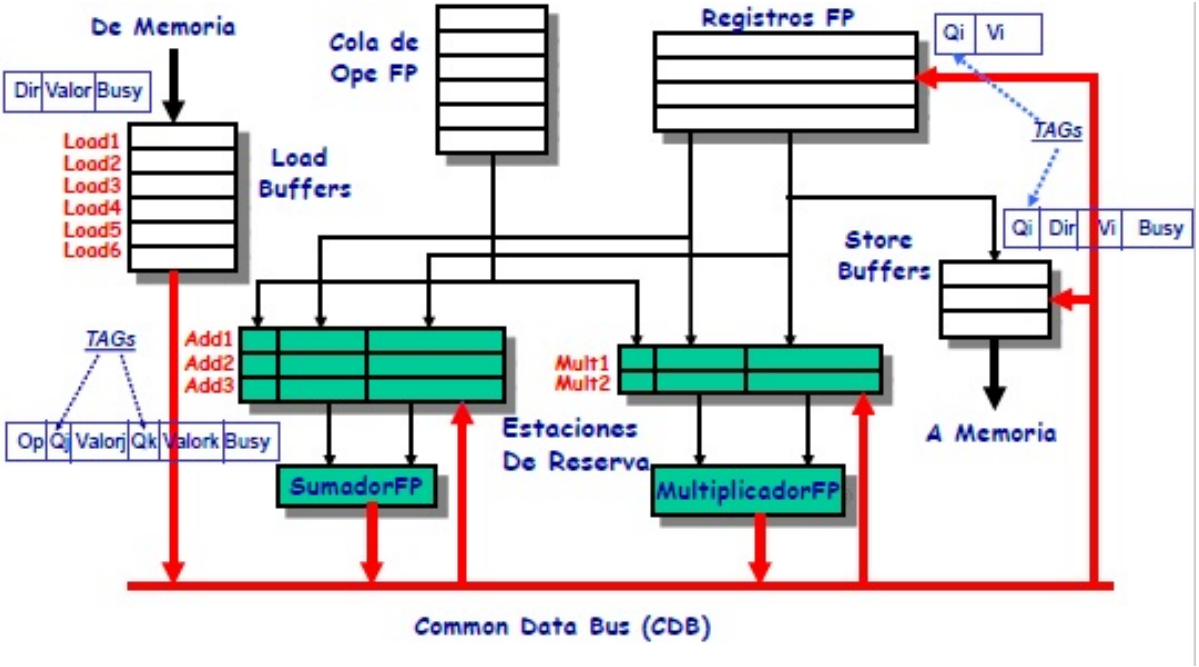
Elimina dinámicamente los riesgos WAW y WAR mediante el renombrado de registros.
Motivación:

- Arquitectura 360, 4 registros FP (pocos registros → muchas dependencias de nombre).

- Arquitectura RM.
- UF segmentadas Add(3), Mul(2), Load(6), Stores(3).
- Requerimientos de diseño: Arquitectura segmentada del alto rendimiento, pero compatible hacia abajo con el resto de máquinas de la familia 360 → no se pueden añadir más registros de FP.

Procesadores que usan mecanismos basados en este algoritmo:

- Alpha 21264.
- Pentium III-4-Core generations.
- AMD Opteron-Athlon.
- Zen.IBM Power 4-5-7-8...



Excepciones imprecisas.
Issue en orden– Ejecución fuera de orden– Finalización fuera de orden.
Componentes de las ER:

- Op: Operación a realizar.
- Valorj, Valork: Valores de los operandos fuente (si están calculados).
- Qj, Qk: Nº de la ER que está produciendo los operandos fuente. Notar: Qj, Qk=0 → operando calculado.
- Busy: Indica ER ocupada.

Store Buffers:

- Vi: Valor que debe ser almacenado en memoria (si está calculado).

- Dir: Dirección de almacenamiento.
- Qi: Nº de la ER que está produciendo el resultado. Si Qi=0→ valor calculado.

Registros FP:

- Vi: Valor almacenado en el registro (puede estar obsoleto).
- Qi: Nº de la ER que está produciendo el valor a almacenar en el registro. Si Qi=0 → no hay ninguna ER que vaya a almacenar en el registro. Si Qi=0 → no hay ninguna ER que vaya a almacenar un nuevo valor en el reg (el valor del registro no está obsoleto).

Load buffers:

- Valor: Valor leído en la memoria (si ya se ha completado la lectura).
- Dir: Dirección de lectura.

2.2.3. Tres estados para una instrucción en el algoritmo

Issue:

- Toma la instrucción (COD Fi, Fj, Fk) de la cola de instrucciones.
- Envía la instrucción a la Estación Reserva (ER) correspondiente si hay entradas disponible (caso contrario → bloqueo).
- Envía los operandos, si están disponibles, o identificación (tag) de la(s) ER(s) donde están las instrucciones que los van a generar.
- En load/store: envía la instrucción si hay buffer libre.
- En suma: Copiar "Tag+Valor" de registros fuente sobre campos "Tag+Valor" de ER.
- Marca tag de registros destino (Fi) con ID de la ER donde se aloja la instrucción.

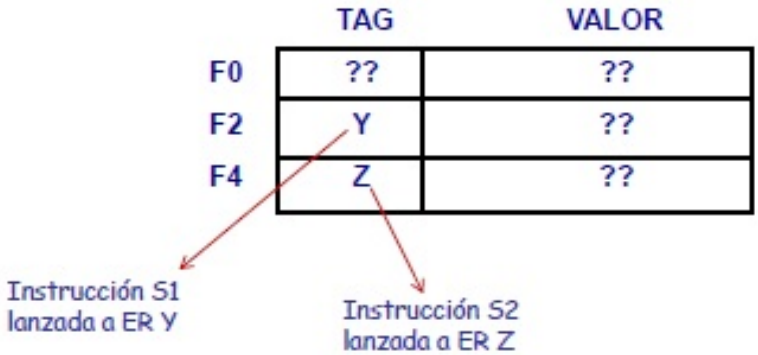
Ejecución:
Monitoriza CDB para disponibilidad de operandos. Cuandos los operandos están listos manda ejecutar. Resuelve riesgos LDE.
Escritura de resultados:

- Vía CDB en registros y ER(s) que esperan en el resultado: Comparación de tags.
- Marca ER como libre.
    - No chequea riesgos EDE ni EDL (eliminados por el renombrado dinámico).
    - En un bus normal: dato+destino.
    - En CDB: dato+fuente.

2.2.4. Ejemplo: Ejecución de la secuencia

```
(S1):  COD1   F2, ---, ---
(S2):  COD2   F4, ---, ---
(S3):  ADD.D  F0, F2, F4
```

Suposiciones: S1 y S2 ya lanzadas a ejecución, S3 se va a lanzar.
Estado inicial de registros:



Paso 1: Lanzamiento de ADD.D a la ER X (X es una ER libre de la UF de Suma/Resta de Punto Flotante).

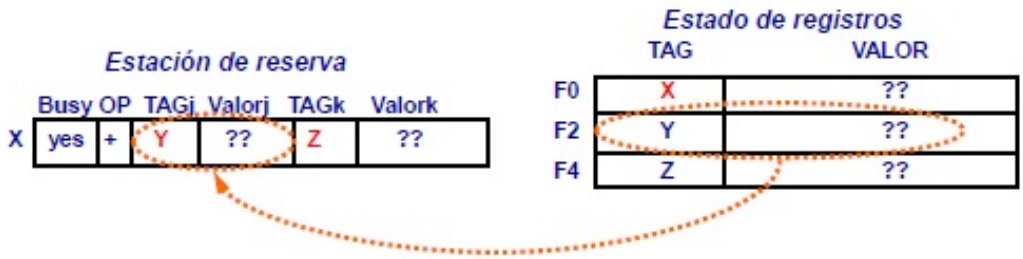


Figura 2.10: Paso 1

Paso 2: Esperar a que se generen los operandos.
Escritura del resultado de S1 sobre el CDB: (Y, 22.57).

Figura 2.11: Paso 2A

Escritura del resultad de S2 sobre el CDB: (Z, 3.2).

Estación de reserva

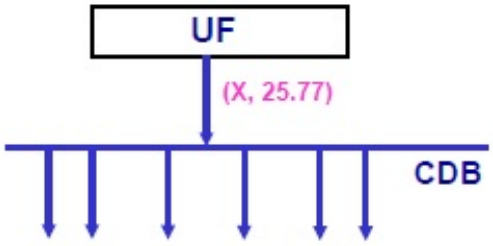
	Busy	OP	TAGj	Valorj	TAGk	Valork
X	yes	+	0	22.57	0	3.2

Estado de registros

	TAG	VALOR
F0	X	??
F2	0	22.57
F4	0	3.2

Figura 2.12: Paso 2B

Paso 3: Ejecutar operación y escribir resultado sobre el CDB.



Estado de registros

	TAG	VALOR
F0	0	25.77
F2	0	22.57
F4	0	3.2

Figura 2.13: Paso 3

A todas las ER, Store Buffers y Registros que tengan la marca X.
Tiempo de ejecución: LD 2 ciclos, ADD.D y SUB.D 2 ciclos, MULD 10 ciclos, DIV.D 40 ciclos.
Memoria segmentada: acepta comenzar 1 acceso/ciclo.

Instr		J	K	Issue	Ejecución	escritura
L.D	F6	34+	R2			
L.D	F2	45+	R3			
MUL.D	F0	F2	F4			
SUB.D	F8	F6	F2			
DIV.D	F10	F0	F6			
ADD.D	F6	F8	F2			

	Ocupada	Dirección
Load1		
Load2		
Load3		

Estado de ER
Operación
Qj y Qk: ER produciendo operandos
Vj y Vk: valores de los operandos

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
	Mul1						
	Mul2						

Estado de REG

	F0	F2	F4	F6	F8	F10	F12	...
Qi								
Vi								

Ciclo 1:

Instr		J	K	Issue	Ejecución	escritura
L.D	F6	34+	R2	1		
L.D	F2	45+	R3			
MUL.D	F0	F2	F4			
SUB.D	F8	F6	F2			
DIV.D	F10	F0	F6			
ADD.D	F6	F8	F2			

	Ocupada	Dirección
Load1	SI	34+R2
Load2	NO	
Load3	NO	

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1	NO					
	Add2	NO					
	Add3	NO					
	Mul1	NO					
	Mul2	NO					

Estado de REG

	F0	F2	F4	F6	F8	F10	F12	...
Qi				Load1				
Vi								

Ciclo 2:

Instr		J	K	Issue	Ejecución	escritura
L.D	F6	34+	R2	1		
L.D	F2	45+	R3	2		
MUL.D	F0	F2	F4			
SUB.D	F8	F6	F2			
DIV.D	F10	F0	F6			
ADD.D	F6	F8	F2			

	Ocupada	Dirección
Load1	SI	34+R2
Load2	SI	45+R3
Load3	NO	

No hay bloqueo

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1	NO					
	Add2	NO					
	Add3	NO					
	Mul1	NO					
	Mul2	NO					

Estado de REG

	F0	F2	F4	F6	F8	F10	F12	...
Qi		Load2		Load1				
Vi								

Ciclo 3:

Instr		J	K	Issue	Ejecución	escritura
L.D	F6	34+	R2	1	2-3	
L.D	F2	45+	R3	2		
MUL.D	F0	F2	F4	3		
SUB.D	F8	F6	F2			
DIV.D	F10	F0	F6			
ADD.D	F6	F8	F2			

	Ocupada	Dirección
Load1	SI	34+R2
Load2	SI	45+R3
Load3	NO	

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1	NO					
	Add2	NO					
	Add3	NO					
	Mul1	SI	Mult		R(F4)	Load2	
	Mul2	NO					

Estado de REG

	F0	F2	F4	F6	F8	F10	F12	...
Qi	Mul1	Load2		Load1				
Vi								

Ciclo 4:

Instr		J	K	Issue	Ejecución	escritura
L.D	F6	34+	R2	1	2-3	4
L.D	F2	45+	R3	2	3-4	
MUL.D	F0	F2	F4	3		
SUB.D	F8	F6	F2	4		
DIV.D	F10	F0	F6			
ADD.D	F6	F8	F2			

	Ocupada	Dirección
Load1	NO	
Load2	SI	45+R3
Load3	NO	

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1	SI	Subd	M(34+R2)			Load2
	Add2	NO					
	Add3	NO					
	Mul1	SI	Mult		R(F4)	Load2	
	Mul2	NO					

Estado de REG

	F0	F2	F4	F6	F8	F10	F12	...
Qi	Mul1	Load2			Add1			
Vi				M(34+R2)				

Ciclo 6:

Instr		J	K	Issue	Ejecución	escritura
L.D	F6	34+	R2	1	2-3	4
L.D	F2	45+	R3	2	3-4	5
MUL.D	F0	F2	F4	3		
SUB.D	F8	F6	F2	4		
DIV.D	F10	F0	F6	5		
ADD.D	F6	F8	F2	6		

	Ocupada	Dirección
Load1	NO	
Load2	NO	
Load3	NO	

Ningún bloqueo

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
2	Add1	SI	Subd	M(34+R2)	M(45+R3)		
	Add2	SI	Addd		M(45+R3)	Add1	
	Add3	NO					
10	Mul1	SI	Mult	M(45+R3)	R(F4)		
	Mul2	SI	Divd		M(34+R2)	Mul1	

Estado de REG

	F0	F2	F4	F6	F8	F10	F12	...
Qi	Mul1			Add2	Add1	Mul2		
Vi		M(45+R3)						

F6 reasignado

Ciclo 8:

Instur		J	K	Issue	Ejecución	escritura
L.D	F6	34+	R2	1	2-3	4
L.D	F2	45+	R3	2	3-4	5
MUL.D	F0	F2	F4	3		
SUB.D	F8	F6	F2	4	6-7	8
DIV.D	F10	F0	F6	5		
ADD.D	F6	F8	F2	6		

	Ocupada	Dirección
Load1	NO	
Load2	NO	
Load3	NO	

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1	NO					
2	Add2	SI	Addd	M()-M()	M(45+R3)		
	Add3	NO					
7	Mul1	SI	Mult	M(45+R3)	R(F4)		
	Mul2	SI	Divd		M(34+R2)	Mul1	

Estado de REG

	F0	F2	F4	F6	F8	F10	F12	...
Qi	Mul1			Add2		Mul2		
Vi		M(45)+R3			M()-M()			

Ciclo 13:

Instr		J	K	Issue	Ejecución	escritura
L.D	F6	34+	R2	1	2-3	4
L.D	F2	45+	R3	2	3-4	5
MUL.D	F0	F2	F4	3		
SUB.D	F8	F6	F2	4	6-7	8
DIV.D	F10	F0	F6	5		
ADD.D	F6	F8	F2	6	9-10	11

	Ocupada	Dirección
Load1	NO	
Load2	NO	
Load3	NO	

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1	NO					
	Add2	NO					
	Add3	NO					
3	Mul1	SI	Mult	M(45+R3)	R(F4)		
	Mul2	SI	Divd		M(34+R2)	Mul1	

Estado de REG

	F0	F2	F4	F6	F8	F10	F12	...
Qi	Mul1					Mul2		
Vi		M(45)+R3		F8+M()	M()-M()			

Ciclo 16:

Instr		J	K	Issue	Ejecución	escritura
L.D	F6	34+	R2	1	2-3	4
L.D	F2	45+	R3	2	3-4	5
MUL.D	F0	F2	F4	3	6-15	16
SUB.D	F8	F6	F2	4	6-7	8
DIV.D	F10	F0	F6	5		
ADD.D	F6	F8	F2	6	9-10	11

	Ocupada	Dirección
Load1	NO	
Load2	NO	
Load3	NO	

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1	NO					
	Add2	NO					
	Add3	NO					
	Mul1	NO					
40	Mul2	SI	Divd	M()*F4	M(34+R2)		

Estado de REG

	F0	F2	F4	F6	F8	F10	F12	...
Qi						Mul2		
Vi	M()*F4	M(45)+R3		F8+M()	M()-M()			

Ciclo 57: Finalización en desorden. Excepciones.

Instr		J	K	Issue	Ejecución.	escritura
L.D	F6	34+	R2	1	2-3	4
L.D	F2	45+	R3	2	3-4	5
MUL.D	F0	F2	F4	3	6-15	16
SUB.D	F8	F6	F2	4	6-7	8
DIV.D	F10	F0	F6	5	17-56	57
ADD.D	F6	F8	F2	6	9-10	11

	Ocupada	Dirección
Load1	NO	
Load2	NO	
Load3	NO	

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1	NO					
	Add2	NO					
	Add3	NO					
	Mul1	NO					
	Mul2	NO					

Estado de REG

	F0	F2	F4	F6	F8	F10	F12	...
Qi								
Vi	M()*F4	M(45)+R3		F8+M()	M()-M()	F0/F6		

Renombrado dinámico de unrolling

```
Loop: L.D      F0,0(R1)
      MUL.D    F4,F0,F2
      S.D      F4,0(R1)
      DADDIU   R1,R1,#-8
      BNE      R1,R2,Loop
```

Operación: vector F0* escalar F2.
Suposiciones:

- Predicción: el salto se toma.
- MUL.D tarda 4 ciclos.
- Memoria no segmentada, T acceso 1 ciclo.
  - L.D en 1º iteración tarda 8 ciclos (fallo). En 2º iteración: tarda 1 ciclo (acierto).
  - Mostraremos dos iteraciones.

Bucle:

Instr		J	K	Issue	Ejecución	escritura
L.D	F0	0	R1			
MUL.D	F4	F0	F2			
S.D	F4	0	R1			
L.D	F0	0	R1			
MUL.D	F4	F0	F2			
S.D	F4	0	R1			

	Ocupada	Direc	Vi	Qi
Load1	NO			
Load2	NO			
Load3	NO			
Store1	NO			
Store2	NO			
Store3	NO			

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
	Mul1						
	Mul2						

```
Loop: L.D     F0,0(R1)
      MUL.D   F4,F0,F2
      S.D     F4,0(R1)
      DADDIU  R1,R1,#-8
      BNE     R1,R2,Loop
```

Estado de REG

	F0	F2	F4	F6	F8	F10	F12	...
Qi								
Vi								

Ciclo 1:

Instr		J	K	Issue	ejecución	escritura
L.D	F0	0	R1	1		
MUL.D	F4	F0	F2			
S.D	F4	0	R1			
L.D	F0	0	R1			
MUL.D	F4	F0	F2			
S.D	F4	0	R1			

	Ocupada	Direc	Vi	Qi
Load1	SI	80		
Load2	NO			
Load3	NO			
Store1	NO			
Store2	NO			
Store3	NO			

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
	Mul1						
	Mul2						

```
Loop: L.D     F0,0(R1)
      MUL.D   F4,F0,F2
      S.D     F4,0(R1)
      DADDIU  R1,R1,#-8
      BNE     R1,R2,Loop
```

Estado de REG
R1=80

	F0	F2	F4	F6	F8	F10	F12	...
Qi	Load1							
Vi								

Ojo latencia del primer load

Ciclo 2:

Instr		J	K	Issue	Ejecución	escritura
L.D	F0	0	R1	1		
MUL.D	F4	F0	F2	2		
S.D	F4	0	R1			
L.D	F0	0	R1			
MUL.D	F4	F0	F2			
S.D	F4	0	R1			

	Ocupada	Direc	Vi	Qi
Load1	SI	80		
Load2	NO			
Load3	NO			
Store1	NO			
Store2	NO			
Store3	NO			

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
	Mul1	SI	MULT		R(F2)	Load1	
	Mul2						

```
Loop: L.D      F0,0(R1)
      MUL.D    F4,F0,F2
      S.D      F4,0(R1)
      DADDIU   R1,R1,#-8
      BNE      R1,R2,Loop
```

Estado de REG
R1=80

	F0	F2	F4	F6	F8	F10	F12	...
Qi	Load1		Mul1					
Vi								

Ojo latencia del 1er load

Ciclo 3:

Instr		J	K	Issue	Ejecución	escritura
L.D	F0	0	R1	1		
MUL.D	F4	F0	F2	2		
S.D	F4	0	R1	3		
L.D	F0	0	R1			
MUL.D	F4	F0	F2			
S.D	F4	0	R1			

	Ocupada	Direc	Vi	Qi
Load1	SI	80		
Load2	NO			
Load3	NO			
Store1	SI	80		Mul1
Store2	NO			
Store3	NO			

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
	Mul1	SI	MULT		R(F2)	Load1	
	Mul2						

```
Loop: L.D      F0,0(R1)
      MUL.D    F4,F0,F2
      S.D      F4,0(R1)
      DADDIU   R1,R1,#-8
      BNE      R1,R2,Loop
```

Estado de REG
R1=80

	F0	F2	F4	F6	F8	F10	F12	...
Qi	Load1		Mul1					
Vi								

Ojo latencia del 1er load

Ciclo 6: se lanza LD de 2º iteración. IMPORTANTE: Dependencia EDE.

Cierre del bucle

Instr		J	K	Issue	Ejecución	escritura
L.D	F0	0	R1	1		
MUL.D	F4	F0	F2	2		
S.D	F4	0	R1	3		
L.D	F0	0	R1	6		
MUL.D	F4	F0	F2			
S.D	F4	0	R1			

	Ocupada	Direc	Vi	Qi
Load1	SI	80		
Load2	SI	72		
Load3	NO			
Store1	SI	80		Mul1
Store2	NO			
Store3	NO			

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
	Mul1	SI	MULT		R(F2)	Load1	
	Mul2						

```
Loop: L.D     F0,0(R1)
      MUL.D   F4,F0,F2
      S.D     F4,0(R1)
      DADDIU  R1,R1,#-8
      BNE     R1,R2,Loop
```

Estado de REG
R1=72

Ojo latencia del 1er load

	F0	F2	F4	F6	F8	F10	F12	...
Qi	Load2		Mul1					
Vi								

Renombrado

Ciclo 7:

Instr		J	K	Issue	Ejecución	escritura
L.D	F0	0	R1	1		
MUL.D	F4	F0	F2	2		
S.D	F4	0	R1	3		
L.D	F0	0	R1	6		
MUL.D	F4	F0	F2	7		
S.D	F4	0	R1			

	Ocupada	Direc	Vi	Qi
Load1	SI	80		
Load2	SI	72		
Load3	NO			
Store1	SI	80		Mul1
Store2	NO			
Store3	NO			

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
	Mul1	SI	MULT		R(F2)	Load1	
	Mul2	SI	MULT		R(F2)	Load2	

```
Loop: L.D     F0,0(R1)
      MUL.D   F4,F0,F2
      S.D     F4,0(R1)
      DADDIU  R1,R1,#-8
      BNE     R1,R2,Loop
```

Estado de REG
R1=72

Ojo latencia del 1er load

	F0	F2	F4	F6	F8	F10	F12	...
Qi	Load2		Mul2					
Vi								

Renombrado

Ciclo 10: finaliza primer LD.

Instr		J	K	Issue	Ejecución	escritura
L.D	F0	0	R1	1	9	10
MUL.D	F4	F0	F2	2		
S.D	F4	0	R1	3		
L.D	F0	0	R1	6	10	
MUL.D	F4	F0	F2	7		
S.D	F4	0	R1	8		

	Ocupada	Direc	Vi	Qi
Load1	NO			
Load2	SI	72		
Load3	NO			
Store1	SI	80		Mul1
Store2	SI	72		Mul2
Store3	NO			

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
4	Mul1	SI	MULT	M(80)	R(F2)		
	Mul2	SI	MULT		R(F2)	Load2	

```
Loop: L.D     F0,0(R1)
      MUL.D   F4,F0,F2
      S.D     F4,0(R1)
      DADDIU  R1,R1,#-8
      BNE     R1,R2,Loop
```

Estado de REG
R1=64

	F0	F2	F4	F6	F8	F10	F12	...
Qi	Load2		Mul2					
Vi								

Ojo latencia del 1er load

Ciclo 11: finaliza segundo LD.

Instr		J	K	Issue	Ejecución	escritura
L.D	F0	0	R1	1	9	10
MUL.D	F4	F0	F2	2		
S.D	F4	0	R1	3		
L.D	F0	0	R1	6	10	11
MUL.D	F4	F0	F2	7		
S.D	F4	0	R1	8		

	Ocupada	Direc	Vi	Qi
Load1	NO			
Load2	NO			
Load3	SI	64		
Store1	SI	80		Mul1
Store2	SI	72		Mul2
Store3	NO			

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
3	Mul1	SI	MULT	M(80)	R(F2)		
4	Mul2	SI	MULT	M(72)	R(F2)		

```
Loop: L.D     F0,0(R1)
      MUL.D   F4,F0,F2
      S.D     F4,0(R1)
      DADDIU  R1,R1,#-8
      BNE     R1,R2,Loop
```

Estado de REG
R1=64

	F0	F2	F4	F6	F8	F10	F12	...
Qi			Mul2					
Vi	M(72)							

Ciclo 15: finaliza primer MULT.

Instr		J	K	Issue	Ejecución	escritura
L.D	F0	0	R1	1	9	10
MUL.D	F4	F0	F2	2	11-14	15
S.D	F4	0	R1	3		
L.D	F0	0	R1	6	10	11
MUL.D	F4	F0	F2	7	12-15	
S.D	F4	0	R1	8		

	Ocupada	Direc	Vi	Qi
Load1	NO			
Load2	NO			
Load3	SI	64		
Store1	SI	80	M()*F2	
Store2	SI	72		Mul2
Store3	NO			

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
	Mul1						
	Mul2	SI	MULT	M(72)	R(F2)		

```
Loop: L.D      F0,0(R1)
      MUL.D    F4,F0,F2
      S.D      F4,0(R1)
      DADDIU   R1,R1,#-8
      BNE      R1,R2,Loop
```

Estado de REG
R1=64

	F0	F2	F4	F6	F8	F10	F12	...
Qi			Mul2					
Vi	M(72)							

Ciclo 16: finaliza segundo MULT, se ejecuta primer ST.

Instr		J	K	Issue	Ejecución	escritura
L.D	F0	0	R1	1	9	10
MUL.D	F4	F0	F2	2	11-14	15
S.D	F4	0	R1	3	16	--
L.D	F0	0	R1	6	10	11
MUL.D	F4	F0	F2	7	12-15	16
S.D	F4	0	R1	8		

	Ocupada	Direc	Vi	Qi
Load1	NO			
Load2	NO			
Load3	SI	64		
Store1	NO			
Store2	SI	72	M()*F2	
Store3	NO			

Estado de ER

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
	Mul1						
	Mul2						

```
Loop: L.D      F0,0(R1)
      MUL.D    F4,F0,F2
      S.D      F4,0(R1)
      DADDIU   R1,R1,#-8
      BNE      R1,R2,Loop
```

Estado de REG
R1=64

	F0	F2	F4	F6	F8	F10	F12	...
Qi								
Vi	M(72)		M()*F2					

Ciclo 17: se ejecuta segundo store.

Instr		J	K	Issue	Ejecución	escritura
L.D	F0	0	R1	1	9	10
MUL.D	F4	F0	F2	2	11-14	15
S.D	F4	0	R1	3	16	--
L.D	F0	0	R1	6	10	11
MUL.D	F4	F0	F2	7	12-15	16
S.D	F4	0	R1	8	17	--

	Ocupada	Direc	Vi	Qi
Load1	NO			
Load2	NO			
Load3	SI	64		
Store1	NO			
Store2	NO			
Store3	NO			

Estado de ER

```
Loop: L.D     F0,0(R1)
      MUL.D   F4,F0,F2
      S.D     F4,0(R1)
      DADDIU  R1,R1,#-8
      BNE     R1,R2,Loop
```

Tiempo	FU	Ocupada	Operación	S1 Vj	S2 Vk	ER.P.J Qj	ER.P:K Qk
	Add1						
	Add2						
	Add3						
	Mul1						
	Mul2						

Estado de REG
R1=64

	F0	F2	F4	F6	F8	F10	F12	...
Qi								
Vi	M(72)		M()*F2					

Nomenclatura

- Instrucción aritmética: **D← OP (S1, S2)**. Instrucción aritmética que realiza la operación OP sobre el contenido de los registros S1 y S2, y deposita el resultado en el registro D.

- Instrucción Load: **D← Mem[Dirección]**. Carga en el registro D el contenido de la posición de memoria "Dirección".

- Instrucción Store: **Mem[Dirección]← S**. Almacena el contenido del registro S en la posición de memoria "Dirección".

- Estación de reserva x: **ER(x)(Busy, OP, Qj, Vj, Qk, Vk)**.

- Registro x: **Reg(x)(Qi, Vi)**.

- Load Buffer x: **LB(x)(Busy, Dir)**.

- Store Buffer x: **SB(x)(Busy, Dir, Qi, Vi)**.

Fase Issue:

Tipo de instrucción	Esperar hasta que ...	Hacer ...
Aritmética: D ← OP (S1, S2)	La estación de reserva ER(x) está libre y es capaz de ejecutar OP	ER(x).Busy ← Yes ER(x).OP ← OP ER(x).Q_j ← Reg(S1).Q_i ER(x).V_j ← Reg(S1).V_i ER(x).Q_k ← Reg(S2).Q_i ER(x).V_k ← Reg(S2).V_i Reg(D).Q_i ← x
Load: D ← Mem[Dirección]	El Load Buffer LB(x) está libre	LB(x).Busy ← Yes LB(x).Dir ← Dirección Reg(D).Q_i ← x
Store: Mem[Dirección] ← S	El Store Buffer SB(x) está libre	SB(x).Busy ← Yes SB(x).Dir ← Dirección SB(x).Q_i ← Reg(S).Q_i SB(x).V_i ← Reg(S).V_i

Fase Ejecución:

Tipo de instrucción	Esperar hasta que ...	Hacer ...
Aritmética: D ← OP (S1, S2)	(ER(x).Q_j = 0) Y (ER(x).Q_k = 0)	Ejecutar cálculo OP sobre la UF usando operandos V_j y V_k Generar RESULTADO
Load: D ← Mem[Dirección]	(La dirección efectiva está disponible) Y (LB(x).Dir no tiene dependencias respecto de Stores lanzados antes) (Uso de cola de Load / Store: explicación posterior)	RESULTADO ← Mem[LB(x).Dir]
Store: Mem[Dirección] ← S	(La dirección efectiva está disponible) Y (SB(x).Dir no tiene dependencias con Load ni Sores previos) Y (SB(x).Q_i) = 0)	Mem[SB(x).Dir] ← SB(x).V_i SB(x).Busy ← No

Fase Write:

Tipo de instrucción	Esperar hasta que …	Hacer …
Aritmética: D ← OP (S1, S2)	(Ejecución completa en ER(x)) Y (CDB disponible)	Escribir sobre CDB: (x, RESUL) $\forall z$ (Si Reg(z).Q_i) = x) ⇒ (Reg(z).Q_i = 0) Y (Reg(z).V_i = RESUL) $\forall z$ (Si ER(z).Q_j) = x) ⇒ (ER(z).Q_j = 0) Y (ER(z).V_j = RESUL) $\forall z$ (Si ER(z).Q_k) = x) ⇒ (ER(z).Q_k = 0) Y (ER(z).V_k = RESUL) $\forall z$ (Si SB(z).Q_i) = x) ⇒ (SB(z).Q_i = 0) Y (SB(z).V_i = RESUL) ER(x).Busy = No
Load: D ← Mem[Dirección]	(Acceso a memoria completo en LB(x)) Y (CDB disponible)	Escribir sobre CDB: (x, RESUL) $\forall z$ (Si Reg(z).Q_i) = x) ⇒ (Reg(z).Q_i = 0) Y (Reg(z).V_i = RESUL) $\forall z$ (Si ER(z).Q_j) = x) ⇒ (ER(z).Q_j = 0) Y (ER(z).V_j = RESUL) $\forall z$ (Si ER(z).Q_k) = x) ⇒ (ER(z).Q_k = 0) Y (ER(z).V_k = RESUL) $\forall z$ (Si SB(z).Q_i) = x) ⇒ (SB(z).Q_i = 0) Y (SB(z).V_i = RESUL) LB(x).Busy = No
Store: Mem[Dirección] ← S	Nada	Nada

2.2.5. Ventajas e incovenientes

- Elimina el cuello de botella de los registros.
- Evita EDL y EDE.
- Permite el unrolling en HW.
- No esta limitado a bloques básicos si existe predicción de saltos.
- Complejidad.
- Muchas cargas de registros asociativas por ciclo.
- CDB limita el rendimiento.
- Excepciones imprecisas.

2.2.6. Reduciendo la penalización de los saltos

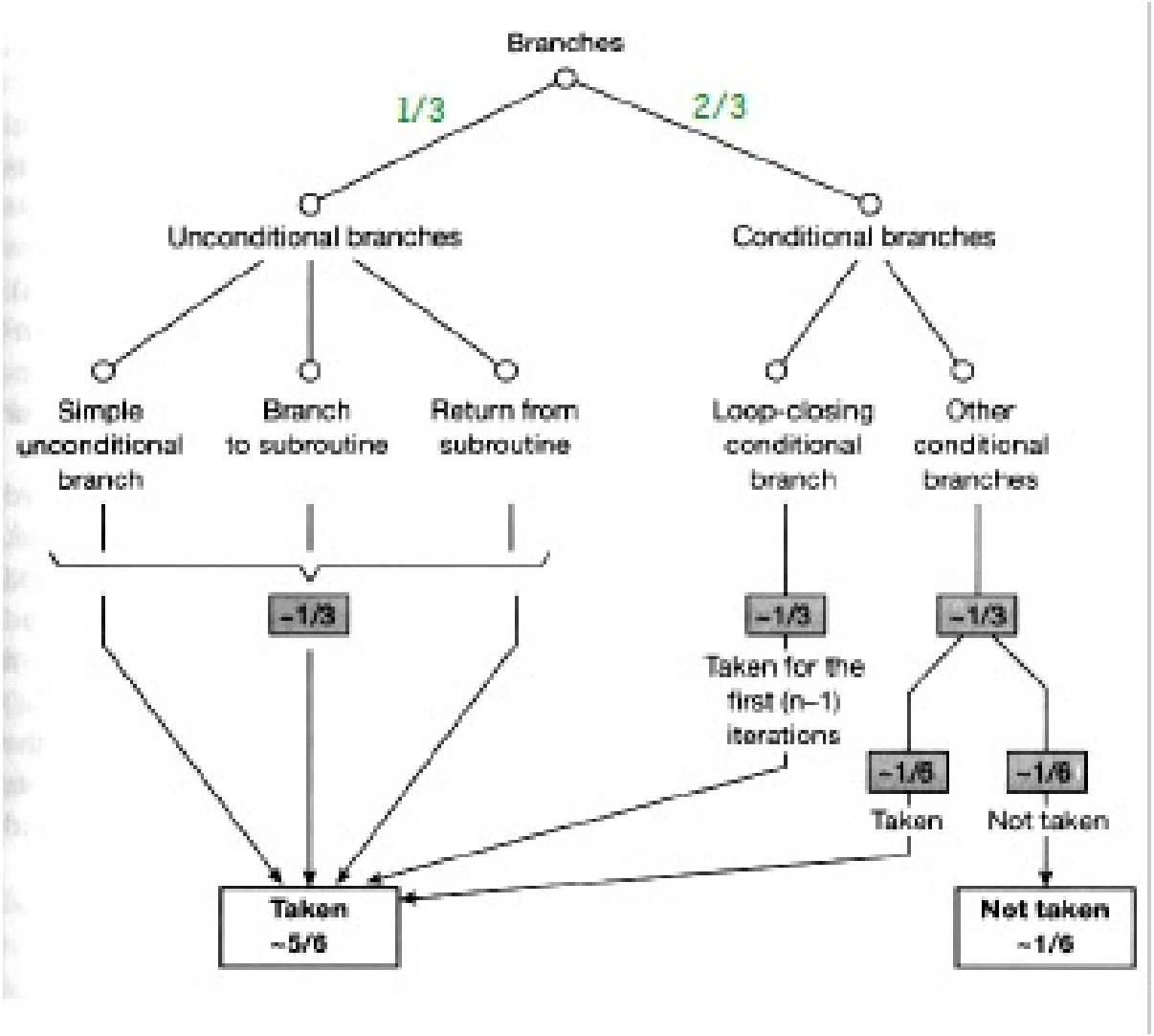
Tipos de saltos

En promedio

- Instrucciones de salto: 1 de cada 5 instrucciones.
- Saltos condicionales: 2 de cada 3 saltos.
- Saltos incondicionales: 1 de cada 3 saltos.
- Saltos tomados: 5 de cada 6 saltos.
- Saltos condicionales tomados: 3 de cada 4 saltos condicionales.
- Saltos incondicionales tomados: todos.

Conclusión: en promedio

- 1 de cada 6 instrucciones es un salto tomado.
- 1 de cada 8 instrucciones es un salto condicional.
- 1 de cada 10 instrucciones es un salto condicional y tomado.



Programas enteros de una de cada 4-5 instrucciones. Flotantes 1 de cada 10-20 instrucciones.

Conclusión:

- Frecuencia de los saltos depende del programa.
- El comportamiento depende del tipo de programa.

Predicción

Cuando se detecta una instrucción de salto condicional sin resolver

- Se supone o predice el camino del salto: tomado o no tomado (Taken- Untaken).
- Si el salto se predice como tomado se predice la dirección del salto.
- La ejecución continúa de forma especulativa a lo largo del camino supuesto.

Cuando se resuelve la condición

Si la predicción fue correcta: La ejecución se confirma y continúa normalmente.
Si la predicción fue incorrecta (fallo de predicción o "misprediction"):

- Se descartan todas las instrucciones ejecutadas especulativamente.
- Se reanuda a lo largo del camino correcto.

Problemas a resolver en instrucciones de salto:

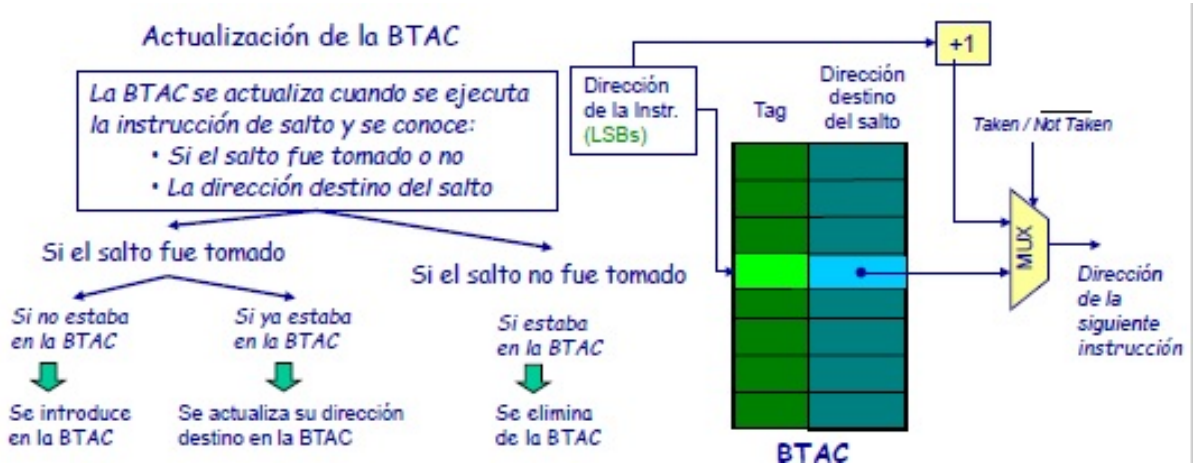
- Predecir la dirección de la instrucción destino del salto con un retardo mínimo (para saltos tomados).
- Predecir el camino que tomará el salto:
  - TAKEN (Tomado).
  - UNTAKEN (No Tomado).

Acceso rápido a la dirección destino del salto I

Branch Target Address Cache (BTAC)

Cache que almacena la dirección destino de los últimos saltos tomados.
Cuando se acerca a una instrucción:

- Se accede simultáneamente a la BTAC utilizando la dirección de la instrucción.
- Si la dirección de la instrucción está en BTAC, entonces es un salto. Si el salto se predice como tomado la dirección destino del salto se lee de la BTAC.



Acceso a la dirección destino del salto II

Alternativasde diseño de la BTAC:

- Cache de Acceso Directo:

  - **Ventaja**: Menor coste.
  - **Desventaja**: "Aliasing" (destrucción de información si dos saltos compiten por la misma entrada).

- Cache Totalmente Asociativa:
  - **Ventaja**: menos Aliasing.
  - **Desventaja**: Mayor coste HW.

- Cache Asociativa por conjuntos: Solución intermedia.

Ejemplos: Pentium (256), Pentium II (512), Pentium 4 (4K), AMD 64 (2K).
Variación Branch Target Instruction Cache:

- Almacenar la instrucción "más ventajas" si tiempo de acceso a las instrucciones es alto.
- Ejemplos: AMD K6, K7, NexGen Nx586.

Clasificación de técnicas de predicción de saltos

Estática:

- Fija.
- Basada en Sentido del salto.
- Basada en Opcode.
- Dirigida por Compilador.

Dinámica:

- Historia Local.
- Historia Global.
- Híbridos.

Predicción estática

Predición Fija:
ALWAYS TAKEN:

- Predecir todos los saltos como tomados.
- Mayor número de aciertos de predicción (3 de cada saltos condicionales son tomados).
- Mayor coste hardware (necesita almacenar la dirección destino del salto).

ALWAYS NOT TAKEN:

- Predecir todos los saltos no tomados.

- Menor número de aciertos de predicción (sólo 1 de cada 4 saltos condicionales no es tomado).
- Menor coste hardware.

Predicción basada en el sentido del salto:
Saltos hacia atrás: TOMADOS. La mayoría de saltos hacia atrás corresponden a bucles.
Saltos hacia delante: NO TOMADOS. La mayoría de saltos hacia delante corresponden a IF-THEN-ELSE.

IMPORTANTE

Mal comportamiento en programas con pocos bucles y muchos IF-THEN-ELSE.

Predicción estática

Predicción basada en el OPCODE de la instrucción de salto.
Fundamento: La probabilidad de que un salto sea tomado del tipo de salto. El salto es tomado para ciertos códigos de operación y no tomado para otros.

Predicción dirigida por el compilador

Basada en el tipo de construcción: El compilador predice si el salto será tomado o no dependiendo del tipo de construcción de control.
Basado en profiling: El compilador predice en función del comportamiento de esa instrucción en ejecuciones previas del programa.
Especificadopor el programador: El programa indica al compilador si el salto debe ser tomado o no (mediante directivas específicas).

IMPORTANTE

- Se añade un Bit de Predicción al opcode de la instrucción.
- El compilador activa o desactiva este bit para indicar su predicción.

Predictores Dinámicos

La predicción se realiza observando el comportamiento de las instrucciones de salto en las últimas ejecuciones.
Historia:

- Predictores de 1 bit de historia.
- Predictores de 2 bits de historia (bimodal).
- Predictores de 3 bits de historia.

Evolución:

- Predictores correlacionados.

- Predictores híbridos.

Predictor de un BIT:

- Utilizan un bit de predicción por cada instrucción de salto.
- El bit de predicción refleja el comportamiento de la última ejecución de la instrucción de salto → Indica si en la anterior ejecución el salto fue tomado o no.

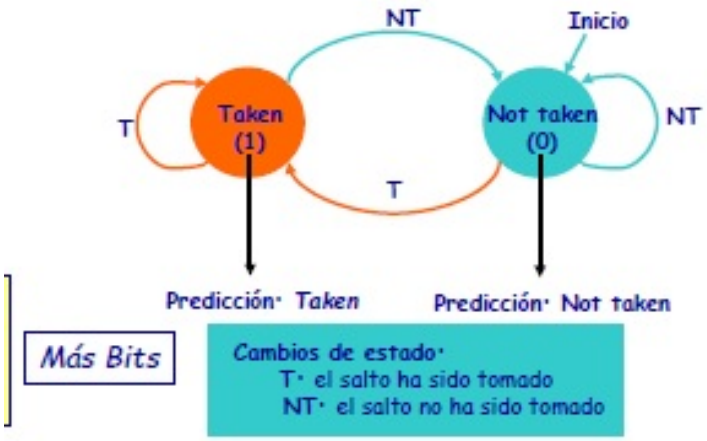
> **Predicción**
>
> - El salto se predice como Taken si en la última ejecución fue tomado.
> - El salto se predice como Not Taken si en la última ejecución no fue tomado.

Funcionamiento:
Máquina de dos estados:

- Not taken (0).
- Taken (1).

Registro de historia: Contador saturado de 1 bit.
Predicción: Valor del registro de historia.
Limitación:

- Sólo se registra el comportamiento de la última ejecución del salto.
- Dos malas predicciones en los cambios.



Predictor de dos bits (BIMODAL):

- Utilizan dos bits de predicción por cada instrucción de salto.
- Estos bits reflejan el comportamiento de las últimas ejecuciones de ese salto.

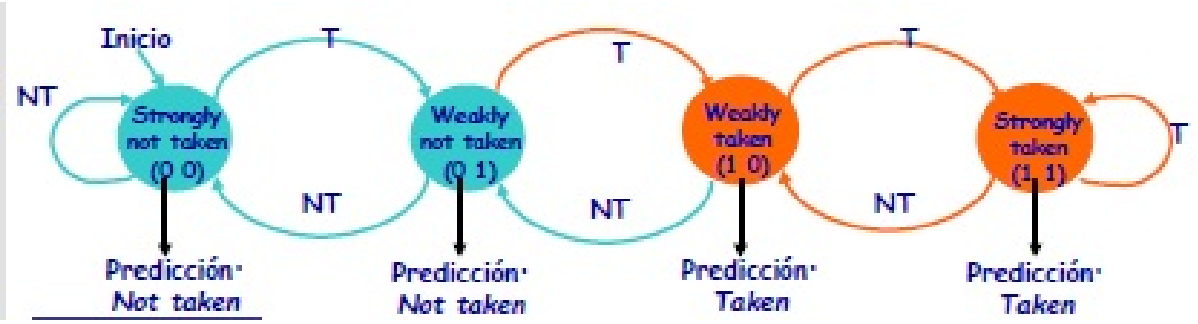
> **Predicción**
>
> - un salto que se toma repetidamente se predice como Taken.
> - Un salto que no se toma repetidamente se predice como Not taken.
> - Si un salto toma una dirección inusual una sola vez, el predictor mantiene la predicción usual.

Funcionamiento:
Máquina de cuatro estados:

- Strongly not taken (00).

- Weakly not taken (01).
- Weakly taken (10).
- Strongly taken (11).

Registro de historia: Contador saturado de 2 bits.
Predicción: Bit más significativo del registro de historia.



Predictores Dinámicos

Implementación de los bits de predicción:

- Branch Target Buffer (BTB): Añade los bits de predicción a las entradas de la BTAC. La BTAC con bits de predicción se denomina BTB.
- Tabla de historia de saltos (BTH): Utiliza una tabla especial, distinta de la BTAC para almacenar los bits de predicción.

Branch Target Buffer (BTB): bits acoplados.
La BTB almacena:

- La dirección destino de los últimos saltos tomados.
- Los bits de predicción de ese salto.

Los campos de la BTB se actualizan después de ejecutar el salto y se conoce:

- Si el salto fue tomado o no → Actualizar bits de predicción.
- La dirección destino del salto → Actualizar BTA.

Imita un sólo bit de predicción:

- Si la instrucción de salto está en la BTB → El salto se predice como tomado.
- Si la instrucción de salto no está en la BTB → El salto se predice como no tomado.

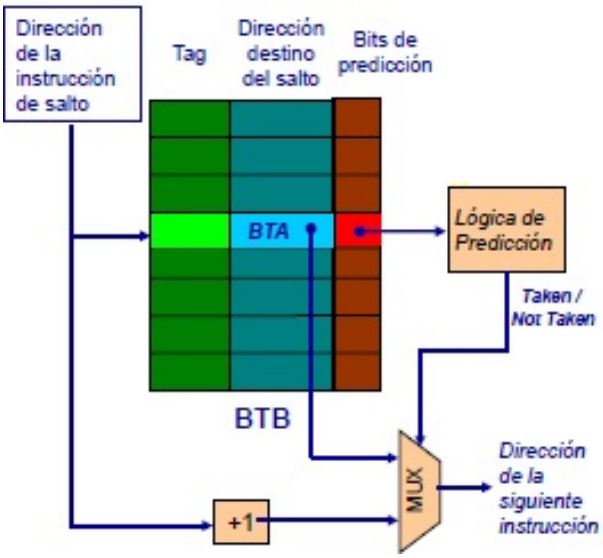


Figura 2.14: BTA

Desventaja: Sólo se pueden predecir aquellas instrucciones de salto que están en la BTB.
Tabla de historia de saltos (BHT): bits desacoplados.
Existen dos tablas distintas:

- La BTAC que almacena la dirección destino de los últimos saltos tomados.
- La BHT que almacena los bits de predicción de todas las instrucciones de salto condicional.

Ventaja: Puede predecir instrucciones que no están en la BTAC (más entradas en BHT que en BTAC).
Desventaja: Aumenta el hardware necesario → 2 tablas (más bit de marca).
Acceso a la BHT:
Usando los bits menos significativos de la dirección:

- Sin TAGs → Menor coste.
- Compartición de entradas → Se degrada el rendimiento.

Asociativa por conjuntos:

- Mayor coste → Tablas pequeñas.
- Para un mismo coste hardware → Peor comportamiento.

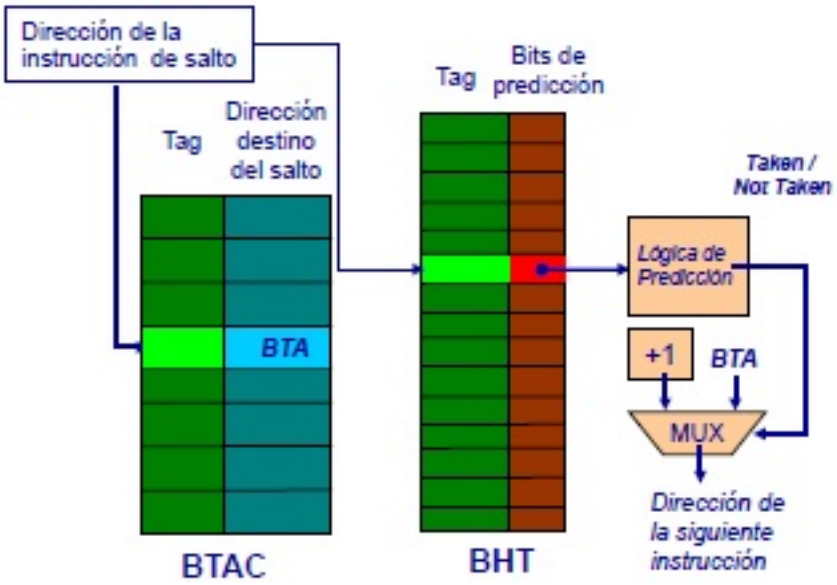


Figura 2.15: BHT

Ejemplo

Relación entre los dos saltos.

```
        BNEZ  R1, L1 ; salto b1 (Salto si d ≠ 0)
        ADDI  R1, R0, #1 ; Como d=0, hacer d=1
L1:     SUBI  R3, R1, #1 ; R3=d(R1)-1
        BNEZ  R3, L2 ; salto b2 (Salto si d≠ 1)
        ----------
L2:
```

Caso 1: d=0,1,2,...

Valor de d	d ≠ 0?	salto b1	d antes de b2	d≠1?	salto b2
0	no	NT	1	no	NT
1	si	T	1	no	NT
2	si	T	2	si	T

Si b1 no se toma, entonces b2 tampoco: correlación entre saltos.
Comportamiento del predictor de un bit (estado inicial "not taken" NT).

```
        BNEZ   R1, L1 ; salto b1 (Salto si d ≠ 0)
        ADDI   R1, R0, #1 ; Como d=0, hacer d=1
L1:     SUBI   R3, R1, #1 ; R3=d(R1)-1
        BNEZ   R3, L2 ; salto b2  (Salto si d≠ 1)
        ..........
L2:
```

Caso 2: d=2,0,2,0,...

Valor de d	Predicción de b1	b1	Nueva predicción de b1	Predicción de b2	b2	Nueva predicción de b2
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

Muchos fallos de predicción.

Solución

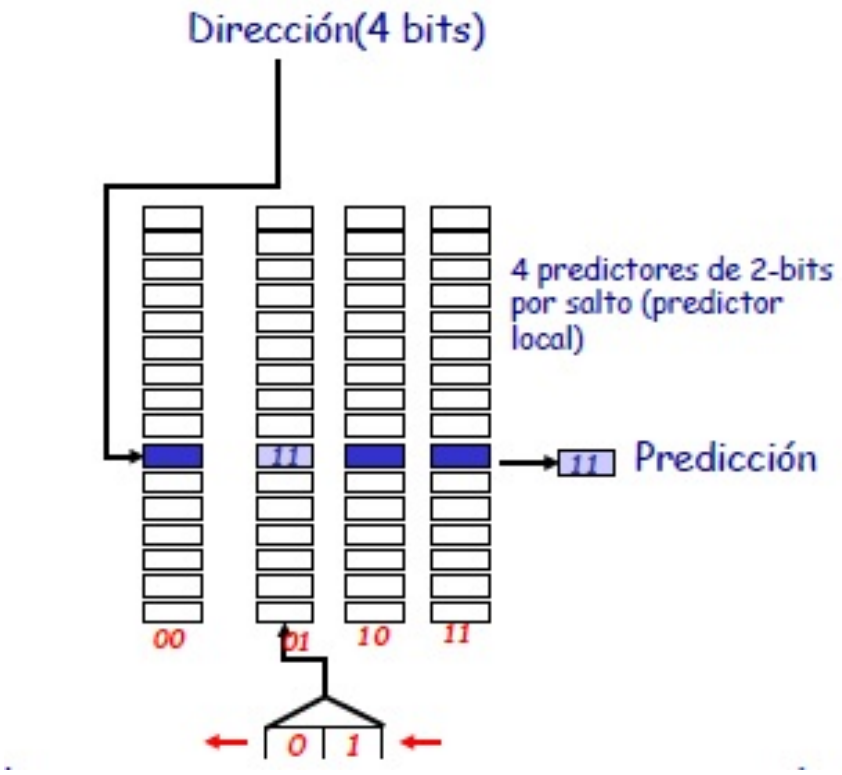
Predictor de dos niveles (1,1): Para cada salto existen 2^1 predictores de 1 bit. El comportamiento último salto (1) determina el predictor que se usa.
Predictor de dos niveles (m,n): Para cada salto existen 2^m predictores de n bits. El comportamiento de los últimos m saltos determinan el predictor que se usa.
Significado de los bits de predicción en un predictor (1,1):

Dos predictores de un bit (P0/P1)

Casos posibles

Bits de predicción: P0 / P1	Predicción si el último salto no tomado: usar P0	Predicción si el último salto tomado: usar P1
NT / NT	NT	NT
NT / T	NT	T
T / NT	T	NT
T / T	T	T

Ejemplo: Tamaño de un predictor (2,2) de 4k entradas: $4 \times 2 \times 4k = 32$ Kb.
Implementación para predictor de dos niveles (2,2) con 16 entradas:



2-bits de historia Global de saltos: 2 Saltos anteriores.
Ejemplo: 01= Penúltimo "not taken" y último "taken".
Comportamiento del predictor de dos niveles (1,1). Estado inicial "not taken" NT.

```
        BNEZ   R1, L1 ; salto b1 (Salto si d ≠ 0)
        ADDI   R1, R0, #1 ; Como d=0, hacer d=1
L1:     SUBI   R3, R1, #1 ; R3=d(R1)-1
        BNEZ   R3, L2 ; salto b2  (Salto si d≠ 1)
        ----------
L2:
```

Caso 2: d=2,0,2,0,...
Sólo se predice mal la 1º iteración (d=2).

d = ?	Predicción de b1	b1	Nueva predicción de b1	Predicción de b2	b2	Nueva predicción de b2
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T /NT	NT	T/NT	NT/T	NT	NT/T

Subrayado en rojo: Bit de predicción seleccionado en cada caso, en función del comportamiento del salto anterior.

Predictores híbridos

Cada uno de los predictores estudiados tienen sus ventajas e inconvenientes.
Combinando el uso de distintos predictores y aplicando uno o otro según convenga, se pueden obtener predicciones mucho más correctas.
Mezcla varios predictores y añade un mecanismo del predictor.
Mecanismo de selección: Elige, en cada caso, el predictor que haya dado mejores resultados hasta el momento.

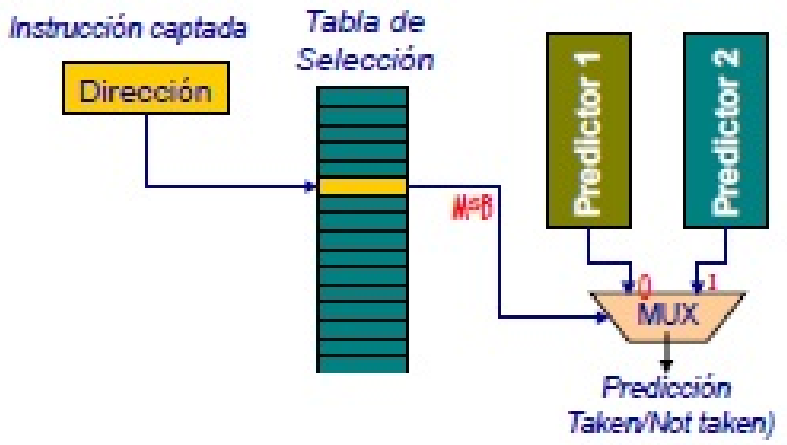


Figura 2.16: Predictor hibrido

Implementación del mecanismo de selección

Importante

Para combinar dos predictores, P1 y P2, se utiliza una tabla de contadores saturados de dos bits indexada por la dirección de la instrucción de salto.

P1	P2	Actualiz. del contador
Fallo	Fallo	Contador no varía
Fallo	Acierto	Cont= Cont+1
Acierto	Fallo	Cont= Cont-1
Acierto	Acierto	Contador no varía

- Si P2 acierta más que P1 → Cont aumenta.
- Si P1 acierta → Contador disminuye.

P1	P2	Actualiz. del contador
Fallo	Fallo	Contador no varía
Fallo	Acierto	Cont= Cont+1
Acierto	Fallo	Cont= Cont-1
Acierto	Acierto	Contador no varía

Ejemplo: Alpha 21264

Predictor competitivo (Tournament Predictor).
Predictor Local: Predicción de un salto en función del comportamiento previo de ese mismo salto.
Considera las 10 últimas ejecuciones del salto.
Predictor global: Predicción de un salto en función del comportamiento de los últimos 12 saltos ejecutados.
Juez: Decide cuál de las dos predicciones se aplica.
Selecciona el predictor que esté manifestando el mejor comportamiento.
Actualización: Al resolver cada salto.

- Se actualizan los predictores en función de su acierto o fallo.
- Si los dos predictores hicieron una predicción distinta, se actualiza el juez para que favorezca que acertó.

Gran importancia para la ejecución especulativa en 21264 (hasta 80 instrucciones en la ventana).
Tasas de predicción correcta (benchmarks): 90-100 %.

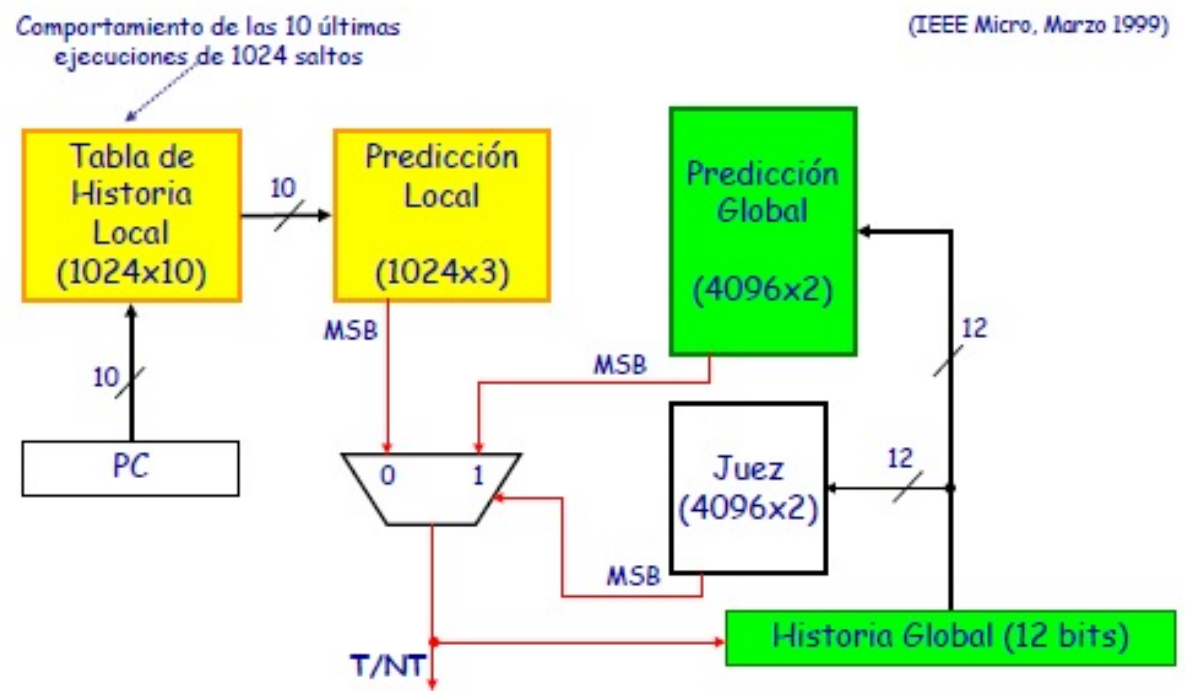


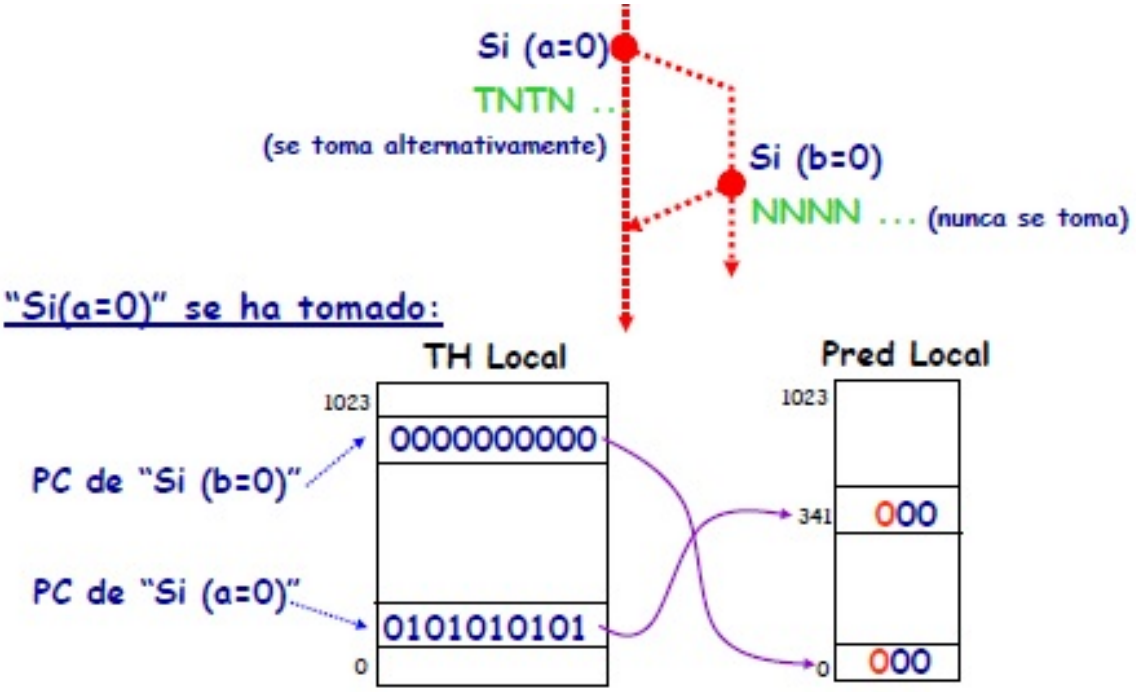
Figura 2.17: Alpha 21264

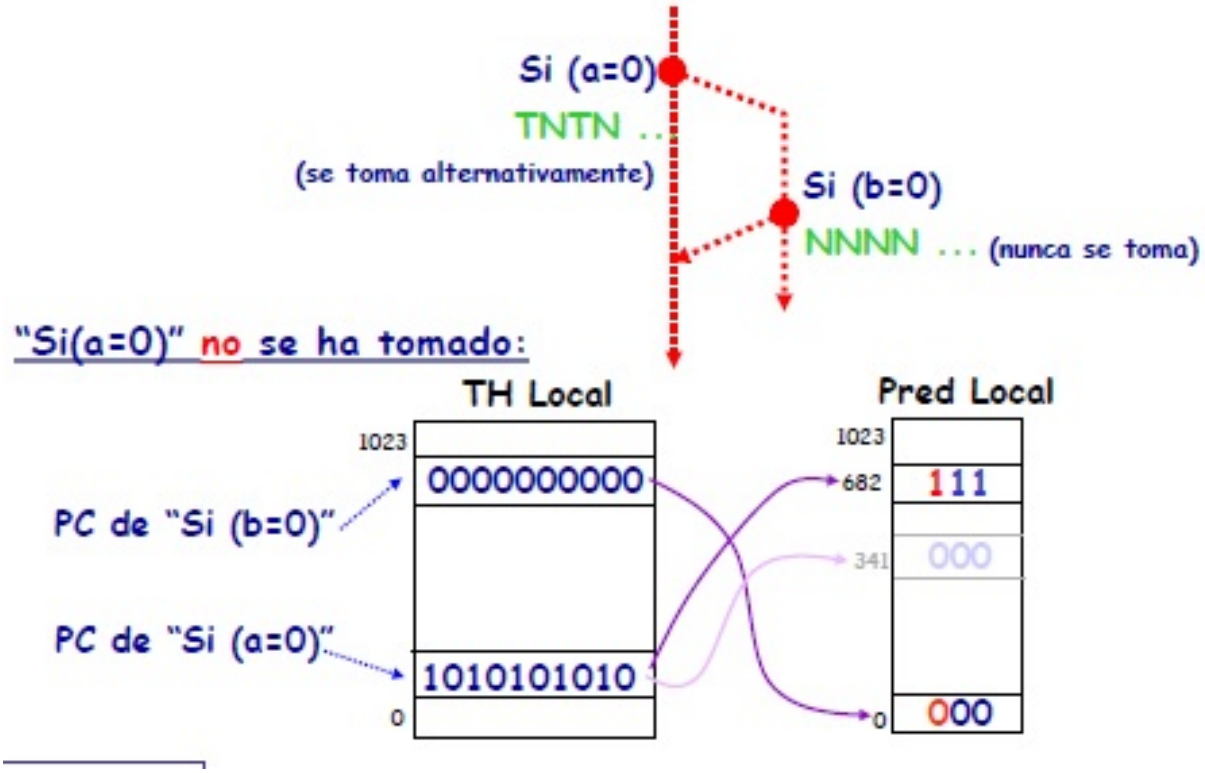
Juez:

- Acierto global y fallo local → Incrementa.
- Fallo global y acierto local → Decrementa.

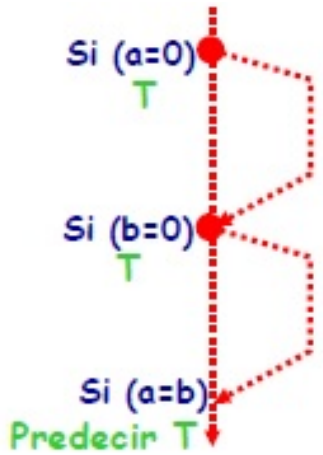
Ejemplo de funcionamiento

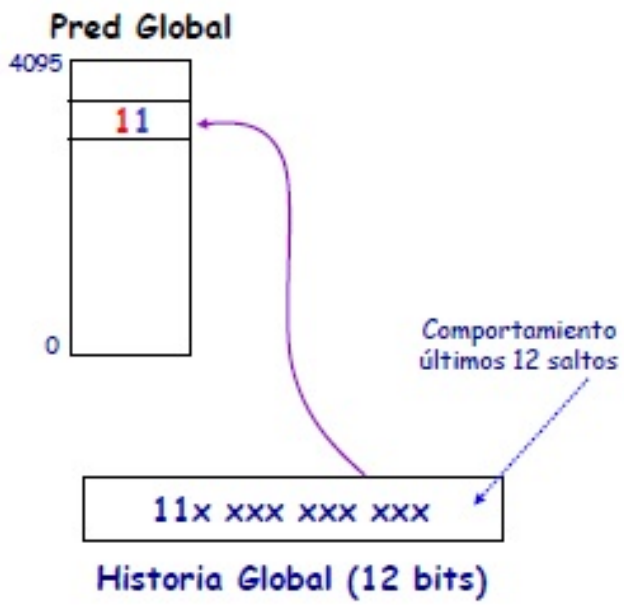
Programa con dos saltos que tiene el comportamiento descrito





Programa con tres saltos que tiene el comportamiento descrito





Comportamiento

La ventaja del predictor hibrido es su capacidad de seleccionar el predictor correcto para un determinado salto.
Muy importante para programas enteros.
Un predictor hibrido selecciona el global casi 40 % de las veces para SPEC integer y menos del 15 % de las veces para SPEC FP.
Predicción de los retornos:
La precisión de los predictores con los retornos es muy baja: La dirección de retorno es diferente en función de la llamada.

Recuperación de fallos de predicción (misprediction)

Tareas básicas:

- Descartar los resultados de las instrucciones ejecutadas especulativamente.
- Reanudar la ejecución por el camino con un retardo mínimo.

Descarte de los resultados

Importante

Los resultados de estas instrucciones especulativas se almacenan en registros temporales (registros de renombramiento o Buffer de reordenamiento).
Estas instrucciones no modifican los contenidos de los registros de la arquitectura ni de la memoria.

Si la ejecución fue correcta: Se actualizan los registros de la arquitectura y/o memoria.
Si la ejecución fue incorrecta: Se descartan los resultados de los registros temporales.

Reanudación de la ejecución por el camino correcto

El procesador debe guardar, al menos, la dirección de comienzo del camino alternativo.
Si la predicción fue "Not taken": El procesador debe calcular y almacenar la dirección destino del salto.
Si la predicción fue "Taken": El procesador debe almacenar la dirección de la instrucción siguiente al salto.
Reducción de los retardos en la recuperación de fallos

> **Importante**
>
> El procesador debe guardar, no solo la dirección del camino alternativo, sino prebuscar y almacenar algunas instrucciones de este camino.

Si la predicción fue "Taken":

- El procesador almacena la dirección del camino secuencial.
- El procesador prebusca y almacena las primeras instrucciones secuenciales.

Si la predicción fue "Not taken":

- El procesador calcula y almacena la dirección destino del salto.
- El procesador prebusca y almacena las primeras instrucciones del destino de salto.

2.2.7. Tratamiento de Saltos: Otras alternativas

Ejecución condicional de instrucciones

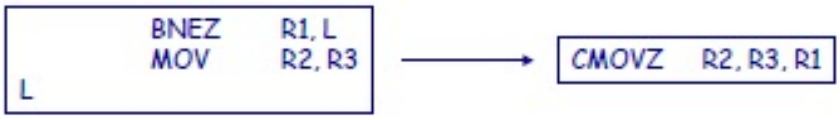
Eliminar, parcialmente, los saltos condicionales mediante instrucciones de ejecución condicional.
Una instrucción de ejecución está formada por:

- Una condición.
- Una operación.

Ejecución condicional:

- Si la condición es cierta → La instrucción se ejecuta.
- Si la condición es falsa → La instrucción se comporta como NOP.

Ejemplo:



> **Importante**
>
> **Ventaja**: Buena solución para implementar alternativas simples de control.
> **Desventaja**: Consumen tiempo en todos los casos. Más lentas que las incondicionales.
> **Ejemplos**: ARM, Alpha, Hp-Pa, MIPS, Sparc.

2.3. Especulación

La predicción de saltos introduce especulación.
Dos tipos de instrucciones en el procesador:

- Las independientes.
- Las que dependen de una predicción de salto. Su finalización depende del acierto o fallo en la predicción.

¿Cómo podemos implementar esta distinción con un modelo de ejecución con finalización Fuera de orden?

Modificando el Algoritmo de Tomasulo para forzar finalización en orden.

Algoritmo de Tomasulo con especulación

- La etapa WB no almacena resultados en registros.
  - Los resultados se guardan temporalmente en un buffer (ROB).
- Se añade una etapa de finalización (commit): se escriben resultados en reg o memoria.
- Las instrucciones hacen la etapa de finalización en orden.
- Una instr que depende de un salto sin resolver no puede hacer la etapa de finalización.

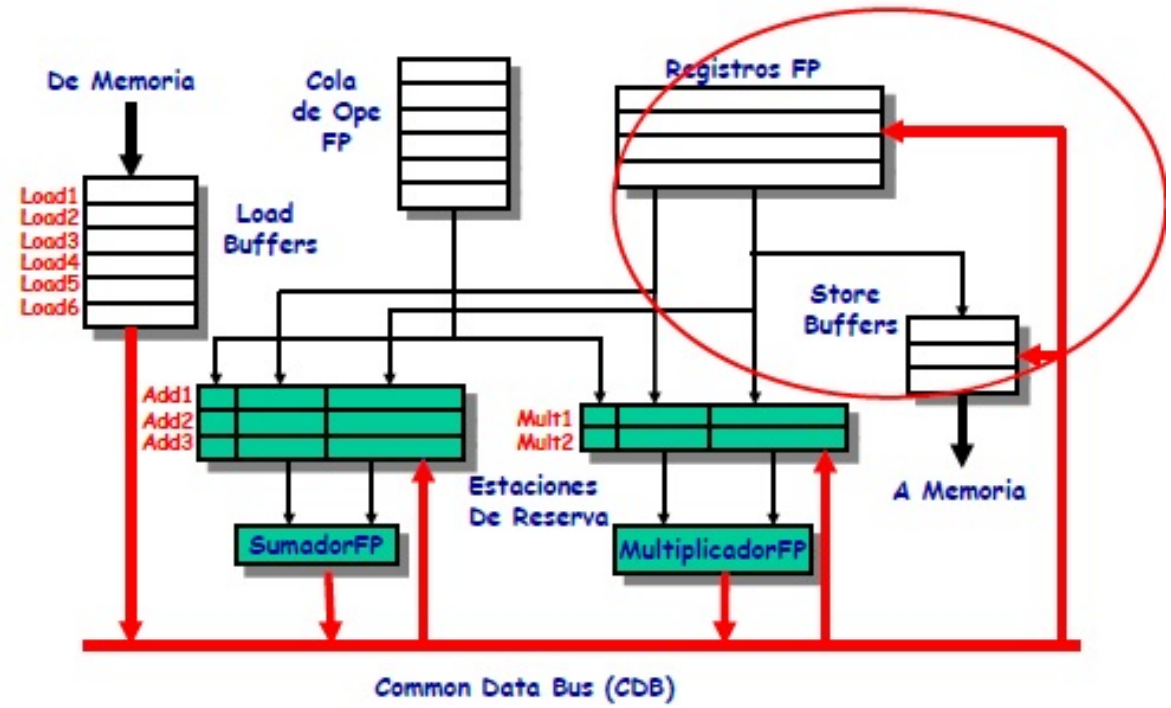


Figura 2.18: Sin especulación

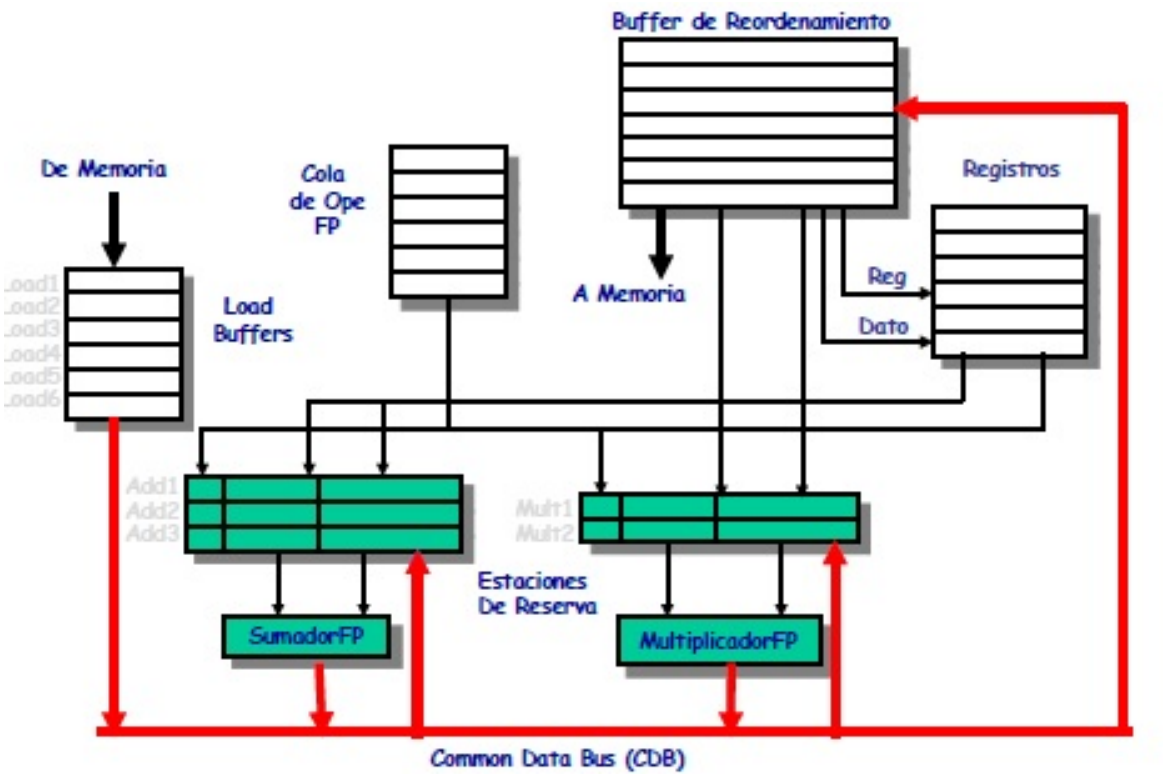


Figura 2.19: Con especulación

El Buffer de Reordenñamiento (ROB):

- Almacena resultados de instrucciones cuya ejecución ha finalizado, pero:
  - Están a la espera de actualizar registros o memoria (finalización en orden).
  - Son dependientes de un salto (ejecución especulativa).
- Permite el paso de operandos entre instrucciones especuladas con dependencia LDE.

Los operandos de una instrucción pueden llegar hasta la ER desde:

- CDB (la instrucción que genera el operando todavía no ha realizado la fase de escritura).
- ROB (la instrucción que genera el operando se ha ejecutado, pero no ha actualizado el banco de registros).
- Registros (la instrucción que genera operando ha finalizado completamente).

Estructura del ROB: cada entrada contiene 4 campos.

- Tipo de instrucción:
  - Salto (sin reg destino).
  - Store (destino en memoria).
  - Aritmética/Load (con destino en registro).
- Destino:

  - Número de registro (Aritmética/Load).
  - Dirección de memoria (Store).

- Valor:
  - Resultado de la ejecución de la instrucción. Guarda el valor hasta que se actualiza registro destino o memoria.
    - En Store: Mientras el valor no está disponible, guarda el nº de la entrada del ROB donde está la instrucción que lo producirá.

- Listo.
  - La instrucción ha completado la fase de ejecución y el resultado está disponible en el campo "Valor".

2.3.1. Algoritmo de Tomasulo con especulación

Los 4 estados del algoritmo:

- **Issue**: Toma la instrucción de la cola. Es necesario: ER con entrada libre y Buffer de Reordenamiento (ROB) con entrada libre.
  Toma operandos de registros o de resultados almacenados en ROB por instrucciones previas.
  - Marca R. Destino con nº de entrada del ROB asignada.
  - Marca ER asignada con nº de entrada del ROB asignada.

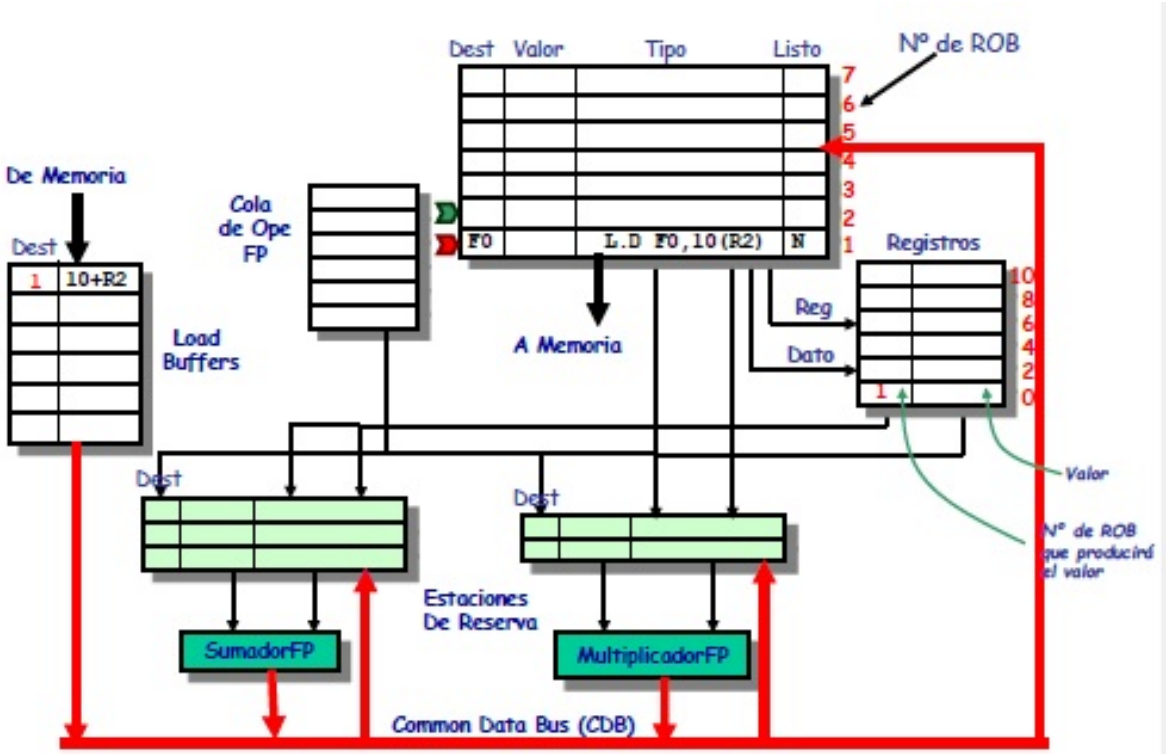
- **Ejecución**: Opera sobre los operandos.
  Espera hasta que los operandos estén disponibles. Chequea CDB.

- **Escribe resultados**: Finaliza ejecución.
  Escribe a través de CDB en todas las ER de Fus y entradas del ROB que estén a la espera del resultado. Libera ER. No escribe registros ni memoria.
  Envía por CDB resultado+ nº de entrada del ROB a la que se dirige.

- **Commit**: Actualiza registros desde el ROB.
  Cuando la Instrucción esta en la cabecera del ROB y resultado presente: Actualiza Registro (o escribe memoria) y elimina la instrucción del ROB.

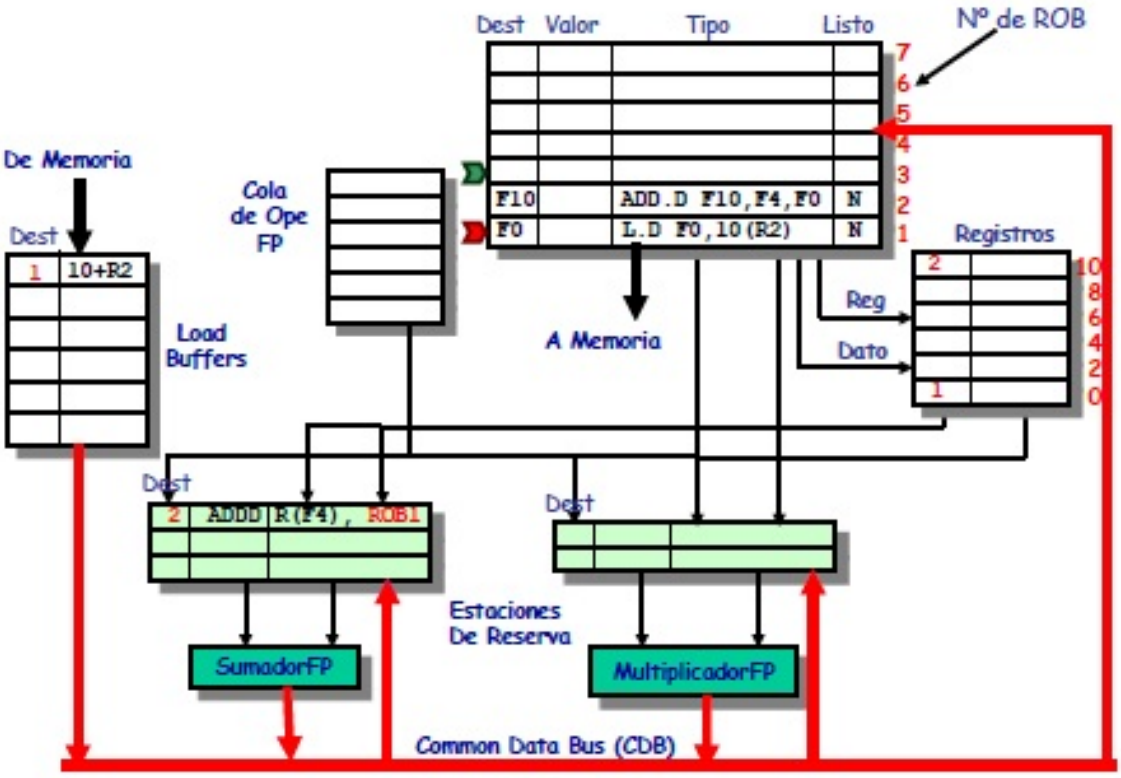
Ejemplo

Se lanza: L.D F0, 10(R2).
Buffer de Reordenamiento

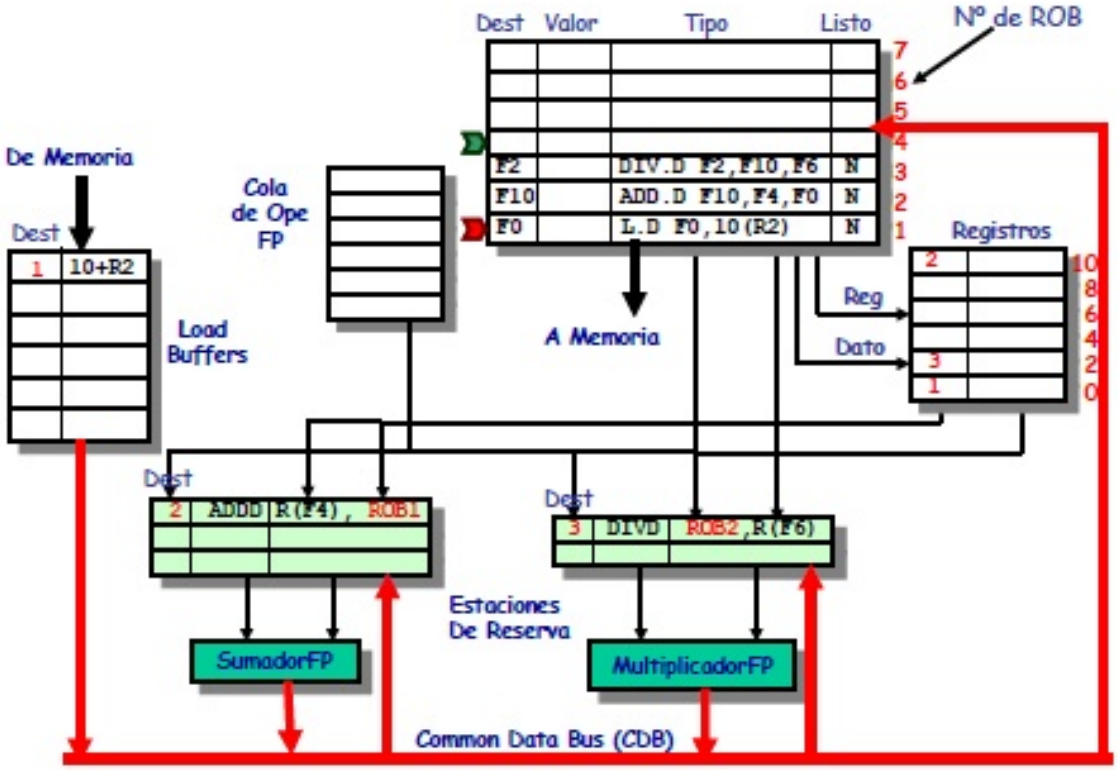


Los LB y ER no tienen un nº de TAG pero tienen un campo "Destino" (nº de entradas del ROB donde se escribirá el resultado).
Se lanza: ADD.D F10, F4, F0.

Buffer de Reordenamiento



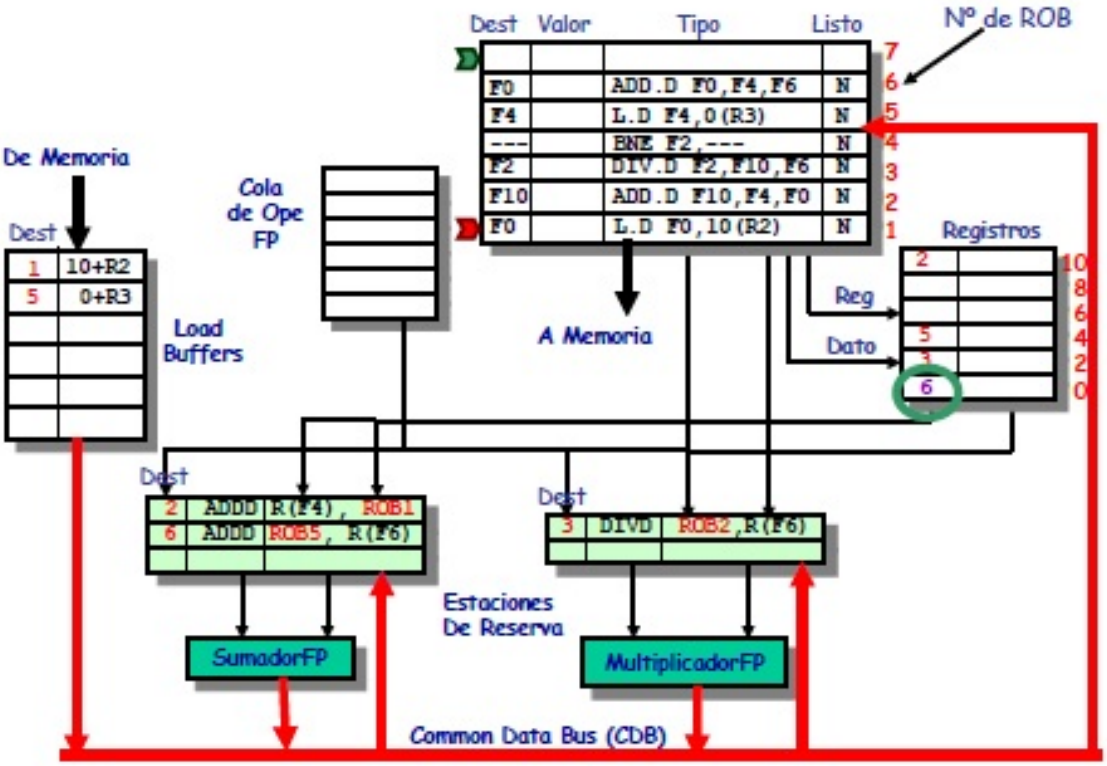
Se lanza: DIV.D F2, F10, F6.
Buffer de Reordenamiento



Se lanza:

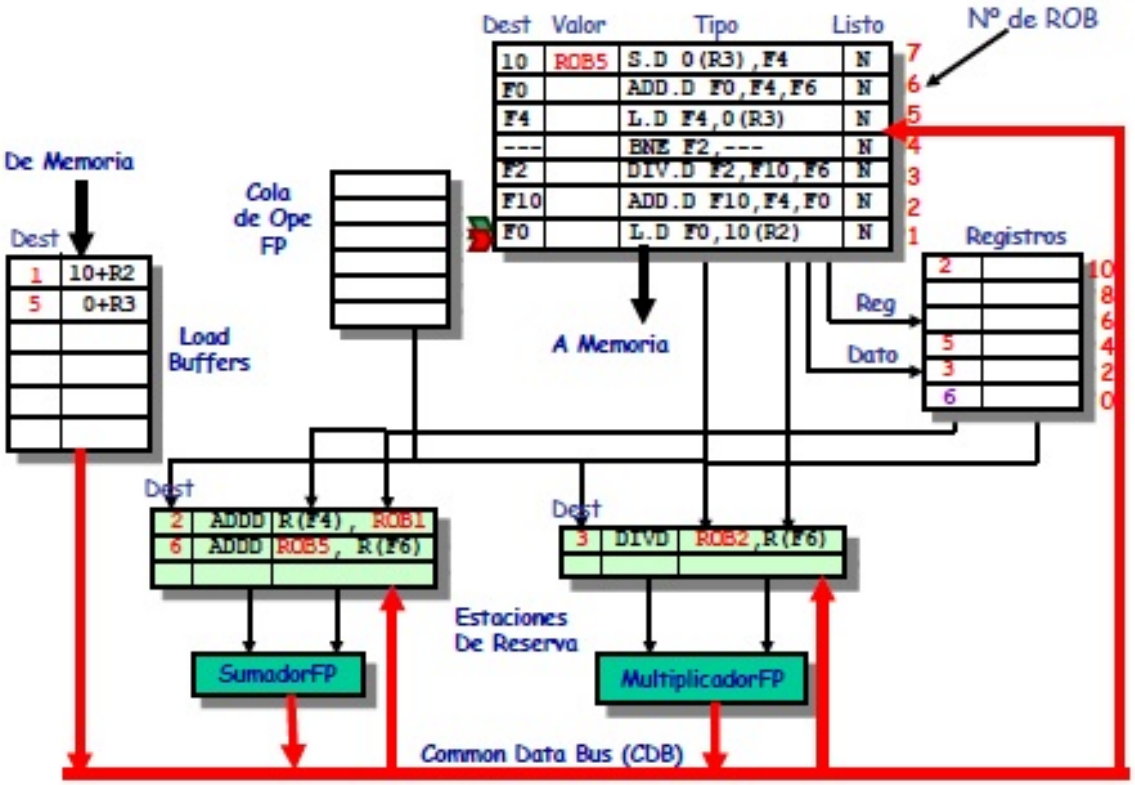
- BNE F2–.
- L.D F4, 0(R3).
- ADD.D F0, F4, F6.

Buffer de Reordenamiento



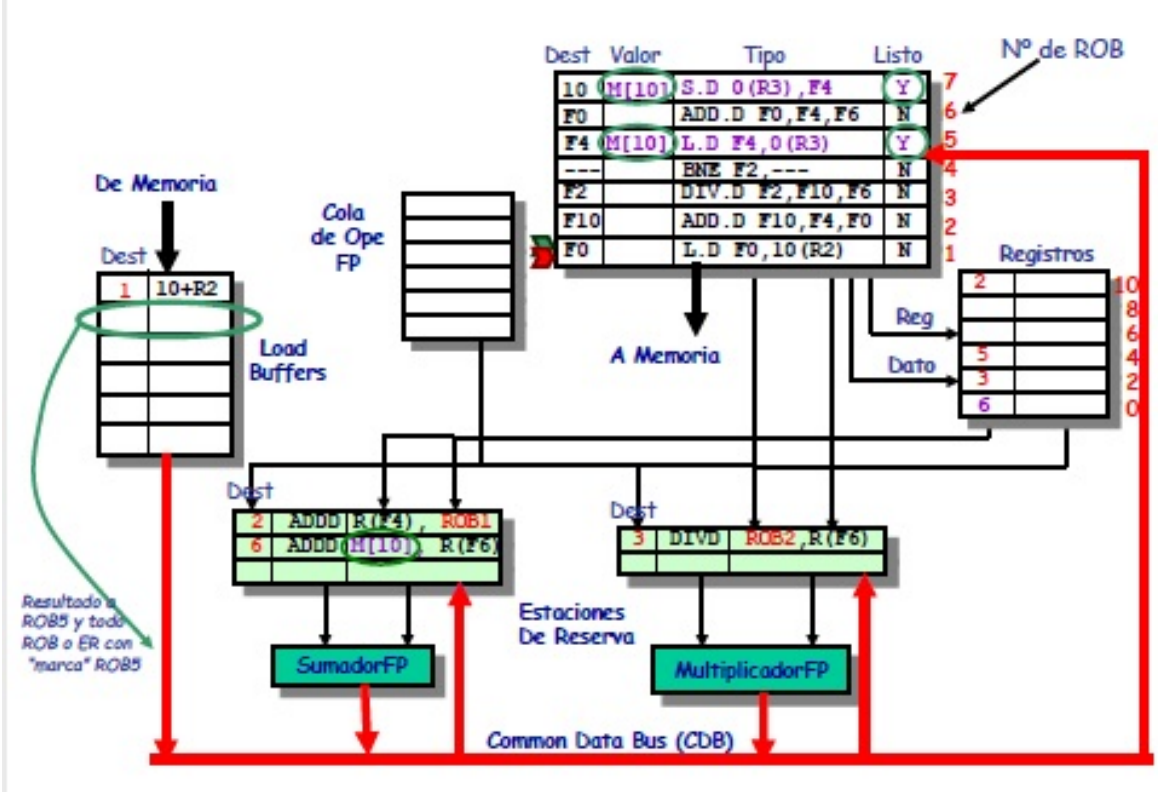
Se lanza: S.D 0(R3), F4 (sup R3=10).

Buffer de Reordenamiento



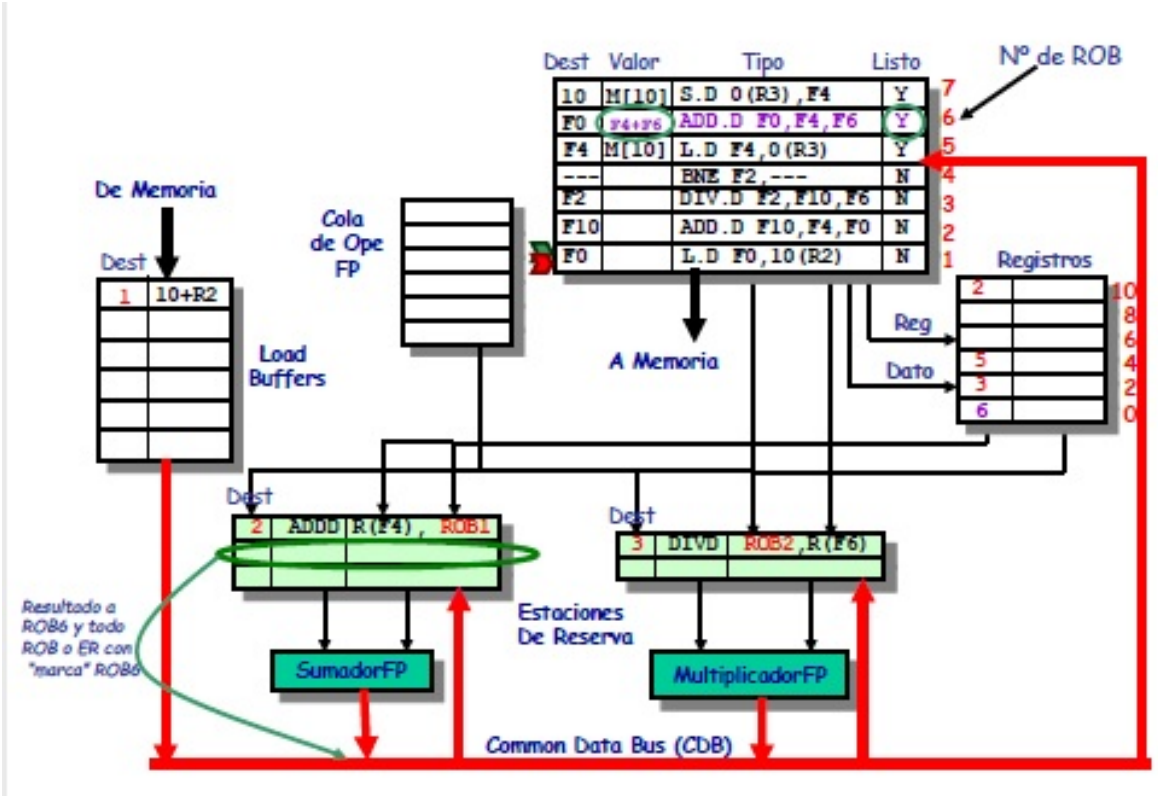
Se lanza: L.D F4, 0(R3).

Buffer de Reordenamiento

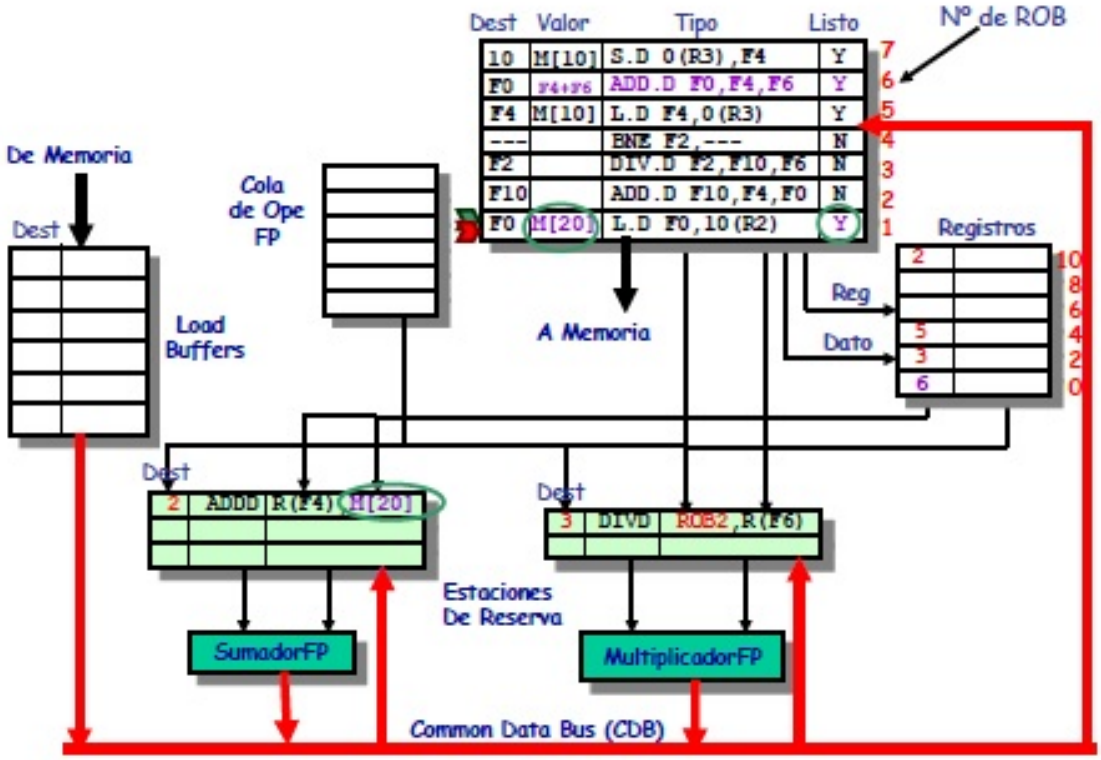


Se ejecuta: ADD.D F0, F4, F6.

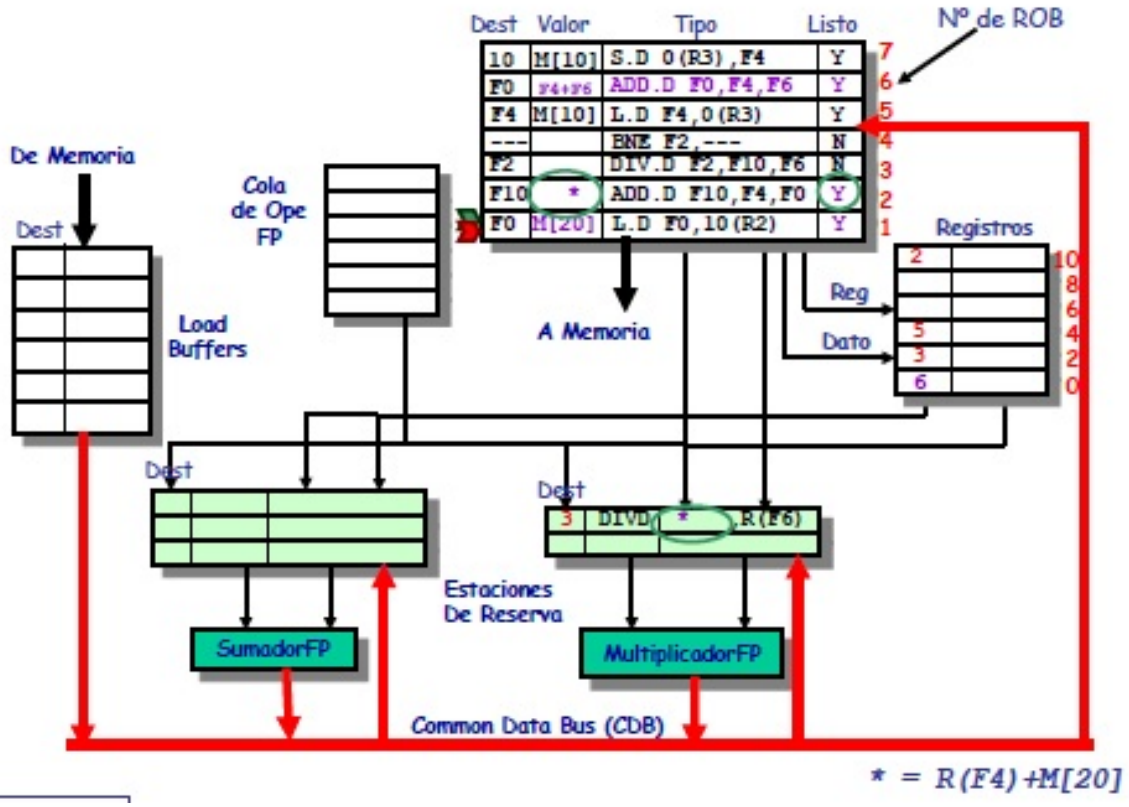
Buffer de Reordenamiento



Se ejecuta: L.D F0, 10(R2), supR2=10.
Buffer de Reordenamiento

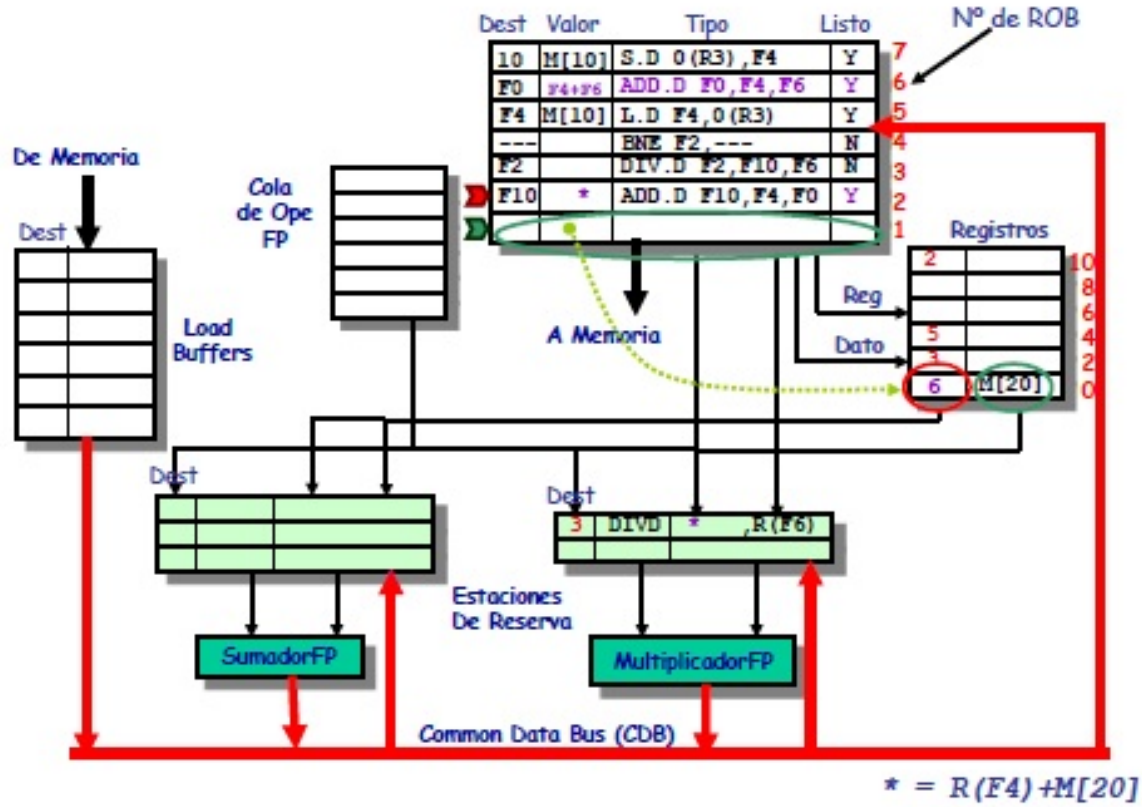


Se ejecuta: ADD.D F10, F4, F0.
Buffer de Reordenamiento

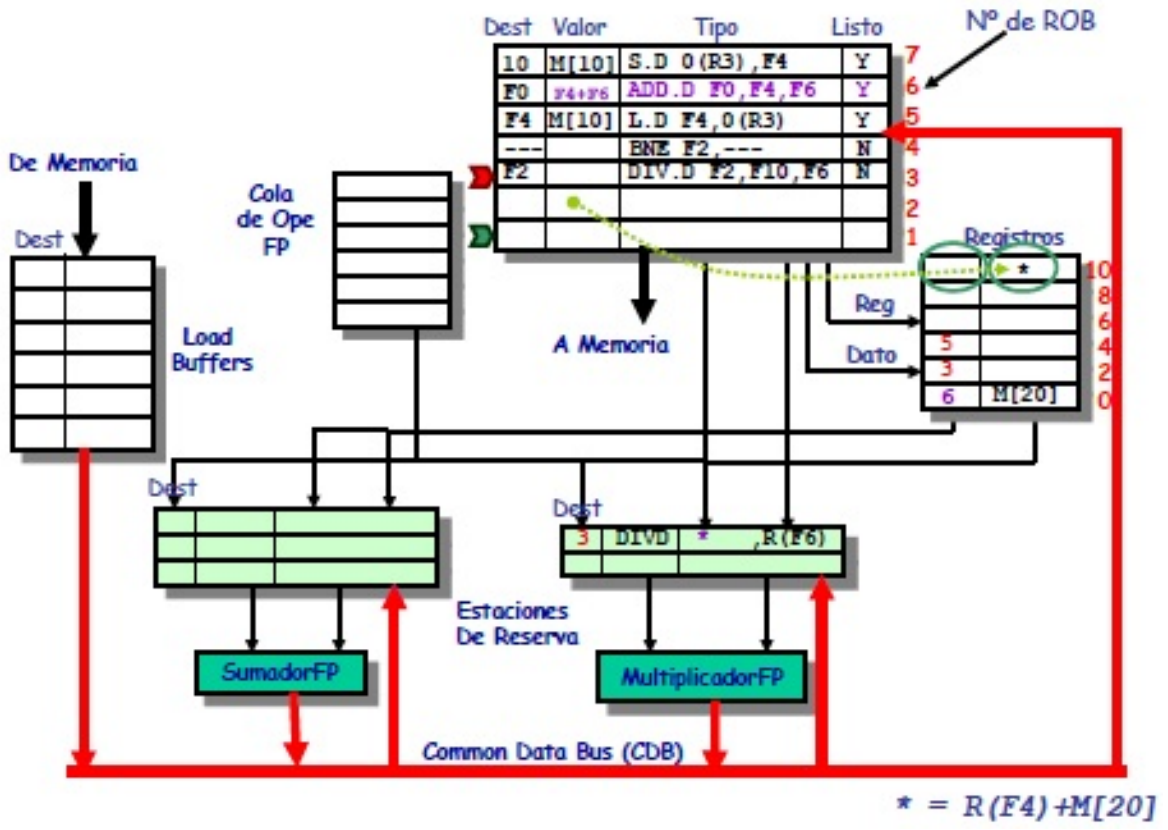


Finaliza (Commit): L.D F0, 10(R2).

Buffer de Reordenamiento



Finaliza (Commit): ADD.D F10, F4, F0.
Buffer de Reordenamiento



2.3.2. Saltos e interpretaciones

El ROB permite recuperarse de saltos mal predichos e implementar un modelo de excepciones precisas.
Si una instrucción de salto bien predicha llega a cabecera de ROB → Eliminarla de ROB.
Si una instrucción de salto mal predicha llega a cabecera de ROB:

- Borrar contenido del ROB, ER y Buffer de Load.
- Borrar marcas (campo "Nº de ROB") de todos los registros.
- Buscar instrucción correcta.

Si una instrucción genera una interrupción:

- Registrar la petición en el ROB.
- Si la instrucción llega a la cabecera del ROB (no especulada), entonces reconocer la interrupción.
- Cualquier instrucción anterior habrá finalizado. Por tanto ninguna instrucción anterior puede provocar una excepción.

2.3.3. Riesgo a través de memoria

Riesgo EDE y EDL: no pueden aparecer dado la actualización de memoria se hace en orden.
Esperar hasta que la instrucción ST se halle en la cabecera de ROB → Todos los LD y ST anteriores se han completado.
Riesgos LDE: Podrían poducirse si un LD accede a la posición de memoria A, habiendo en el ROB un ST previo que almacena el resultado en A. Se evitan mediante el siguiente mecanismo:

- Un LD no ejecuta el acceso a memoria is hay un ST previo en el ROB con la misma dirección de memoria.
- Tampoco se ejecuta el LD si está pendiente el cálculo de la dirección efectiva de algún ST del ROB.

2.4. Explicación Ejercicios Resueltos

Ejercicio 1. Sea la siguiente secuencia de código de instrucciones en punto flotante para un computador similar al DLX que aplica gestión dinámica de instrucciones basada en el algoritmo de Tomasulo:

```
LD     F6,x(R1)
LD     F2,y(R1)
MULTD  F0,F2,F4
SUBD   F8,F6,F2
DIVD   F6,F0,F6
ADDD   F10,F0,F6
ADDD   F6,F8,F2
SD     z(R1),F6
```

Indica el estado de la unidad de ejecución en punto flotante en el momento que todas las instrucciones han sido lanzadas a ejecución en la misma, si la latencia de la memoria es tal que sólo la primera se ha ejecutado completamente, habiéndose cargado en F6 el dato x. Suponer que el resto de registros Fi, poseen un valor dado por la expresión [Fi].
Suponer que hay suficientes estaciones de reserva y buffers load/store, y no se puede escribir un dato desde el CDB e iniciar en el mismo ciclo la operación en espera de ese dato.
Suponer que un mismo ciclo no se puede escribir un dato en el bus común e iniciar la ejecución de una operación que estuviera en espera de ese dato.

Solución ejercicio 1:
Para que todas las instrucciones hayan sido lanzadas a ejecución y solo la primera se haya ejecutado completamente, es necesario que las instrucciones de LOAD tengan una latencia de 6 ciclos.

Inst	Issue (Se lanza a)	Execute	Write
l1: LD F6, x(R1)	1 Load 1	2-7	8
l2: LD F2, y(R1)	2 Load 2	3-8	9
l3: MULTD F0,F2,F4	3 Mul1	10-?	
l4: MULTD F8,F6,F2	4 Add1	10-?	
l5: DIVD F6,F0,F6	5 Mul2		
l6: ADDD F10,F0,F6	6 Add2		
l7: ADDD F6,F8,F2	7 Add3		
l8: SD z(R1), F6	8 Store1		

Vamos a ir realizándolo ciclo por ciclo, empezando por el Ciclo 1:
Ciclo 1:
Se lanza: LD F6,x(R1).

Cuadro 2.1: Estaciones de Reserva

Instr alojada	Nombre ER	Busy	Ope.	Vj	Qj	Vk	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mul1	No					
	Mul2	No					

Cuadro 2.2: Registros

	F0	F2	F4	F6	F8	F10	F12	F14	F16
Qi				Load 1					
Vi									

Cuadro 2.3: Load Buffers

Instr alojada		Busy	Dirección
l1	Load1	Yes	X+R1
	Load2	No	
	Load3	No	

Cuadro 2.4: Store Buffers

Instr alojada		Busy	Dirección	Qi	Vi
	Store1	No			
	Store2	No			
	Store3	No			

Ciclo 2: Se lanza: LD F2, y(R1).

Cuadro 2.5: Estaciones de Reserva

Instr alojada	Nombre ER	Busy	Ope.	Vj	Qj	Vk	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mul1	No					
	Mul2	No					

Cuadro 2.6: Registros

	F0	F2	F4	F6	F8	F10	F12	F14	F16
Qi		Load2		Load1					
Vi									

Cuadro 2.7: Load Buffers

Instr alojada		Busy	Dirección
l1	Load1	Yes	X+R1
l2	Load2	Yes	Y+R1
	Load3	No	

Cuadro 2.8: Store Buffers

Instr alojada		Busy	Dirección	Qi	Vi
	Load1	No			
	Load2	No			
	Load3	No			

Para ir resaltando los pasos en los ciclos se va a utilizar azul y rojo.
Ciclo 3: Se lanza: MULTD F0,F2,F4.

Cuadro 2.9: Estaciones de Reserva

Instr alojada	Nombre ER	Busy	Ope.	Vj	Qj	Vk	Qk
	Add1	No					
l3	Add2	No					
	Add3	No					
	Mul1	Yes	Mul		Load2	[F4]	
	Mul2	No					

Cuadro 2.10: Registros

	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**F14**	**F16**
Qi	Mul1	Load2		Load1					
Vi									

Cuadro 2.11: Load Buffers

Instr alojada		**Busy**	**Dirección**
l1	Load1	Yes	X+R1
l2	Load2	Yes	Y+R1
	Load3	No	

Cuadro 2.12: Store Buffers

Instr alojada		**Busy**	**Dirección**	**Qi**	**Vi**
	Store1	No			
	Store2	No			
	Store3	No			

Ciclo 4: Se lanza: SUBD F8,F6,F2.

Cuadro 2.13: Estaciones de Reserva

Instr alojada		**Busy**	**Dirección**	**Qi**	**Vi**
	Store1	No			
	Store2	No			
	Store3	No			

Cuadro 2.14: Registros

	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**F14**	**F16**
Qi	Mul1	Load2		Load1	Add1				
Vi									

Cuadro 2.15: Load Buffers

Instr alojada		**Busy**	**Dirección**
l1	Load1	Yes	X+R1
l2	Load2	Yes	Y+R1
	Load3	No	

Cuadro 2.16: Store Buffers

Instr alojada		**Busy**	**Dirección**	**Qi**	**Vi**
	Store1	No			
	Store2	No			
	Store3	No			

Ciclo 5: Se lanza: DIVD F6,F0,F6.

Cuadro 2.17: Estaciones de Reserva

Instr alojada	**Nombre ER**	**Busy**	**Ope.**	**Vj**	**Qj**	**Vk**	**Qk**
l4	Add1	Yes					
	Add2	No					
	Add3	No					
l3	Mul1	Yes	Mul		Load2	[F4]	
l5	Mul2	Yes	Div		Mul1		Load1

Cuadro 2.18: Registros

	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**F14**	**F16**
Qi	Mul1	Load2		Mul2	Add1				
Vi									

Cuadro 2.19: Load Buffers

Instr alojada		**Busy**	**Dirección**
l1	Load1	Yes	X+R1
l2	Load2	Yes	Y+R1
	Load3	No	

Cuadro 2.20: Store Buffers

Instr alojada		**Busy**	**Dirección**	**Qi**	**Vi**
	Store1	No			
	Store2	No			
	Store3	No			

Ciclo 6: Se lanza: ADDD F10,F0,F6.

Cuadro 2.21: Estacionesde Reserva

Instr alojada	**Nombre ER**	**Busy**	**Ope.**	**Vj**	**Qj**	**Vk**	**Qk**
l4	Add1	Yes	Sub		Load1		Load2
l6	Add2	Yes	Add		Mul1		Mul2
	Add3	No					
l3	Mul1	Yes	Mul		Load2	[F4]	
l5	Mul2	Yes	Div		Mul1		Load1

Cuadro 2.22: Registros

	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**F14**	**F16**
Qj	Mul1	Load2		Mul2	Add1	Add2			
Vi									

Cuadro 2.23: Load Buffers

Instr alojada		**Busy**	**Dirección**
l1	Load1	Yes	X+R1
l2	Load2	Yes	Y+R1
	Load3	No	

Cuadro 2.24: Store Buffers

Instr alojada		**Busy**	**Dirección**	**Qi**	**Vi**
	Store1	No			
	Store2	No			
	Store3	No			

Ciclo 7: Se lanza: ADDD F6,F8,F2.

Cuadro 2.25: Estaciones de Reserva

Instr alojada	**Nombre ER**	**Busy**	**Ope.**	**Vj**	**Qj**	**Vk**	**Qk**
l4	Add1	Yes	Sub		Load1		Load2
l6	Add2	Yes	Add		Mul1		Mul2
l7	Add3	Yes	Add		Add1		Load2
l3	Mul1	Yes	Mul		Load2	[F4]	
l5	Mul2	Yes	Div		Mul1		

Cuadro 2.26: Registros

	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**F14**	**F16**
Qj	Mul1	Load2		Add3	Add1	Add2			
Vi									

Cuadro 2.27: Load Buffers

	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**F14**	**F16**
Qj	Mul1	Load2		Add3	Add1	Add2			
Vi									

Cuadro 2.28: Store Buffers

Instr alojada		**Busy**	**Dirección**	**Qi**	**Vi**
	Store1	No			
	Store2	No			
	Store3	No			

Ciclo 8: Se lanza: SD z(R1), F6. Acaba: LD F6,x(R1).

Cuadro 2.29: Estaciones de Reserva

Instr alojada	**Nombre ER**	**Busy**	**Ope.**	**Vj**	**Qj**	**Vk**	**Qk**
l4	Add1	Yes	Sub	[F6]			Load2
l6	Add2	Yes	Add		Mul1		Mul2
l7	Add3	Yes	Add		Add1		Load2
l3	Mul1	Yes	Mul		Load2	[F4]	
l5	Mul2	Yes	Div		Mul1	[F6]	

Cuadro 2.30: Registros

	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**F14**	**F16**
Qj	Mul1	Load2		Add3	Add1	Add2			
Vi									

Cuadro 2.31: Load Buffers

Instr alojada		**Busy**	**Dirección**
l1	Load1	No	
l2	Load2	Yes	Y+R1
	Load3	No	

Cuadro 2.32: Store Buffers

Instr alojada		**Busy**	**Dirección**	**Qi**	**Vi**
l8	Load1	Yes	Z+R1	Add3	
	Load2	No			
	Load3	No			

Ejercicio 3. Cierto programa consume la mayor parte de su tiempo de ejecución en el

siguiente fragmento de código:

$$Y = Y + X$$

donde X e Y son vectores. Dicho programa se pretende ejecutar sobre un computador con gestión dinámica de instrucciones que usa el método de Tomasulo. El computador posee operaciones de punto flotante en su juego de instrucciones, funciona a una frecuencia de 1 GHz, el CPI de las instrucciones enteras es 1 y el delay slot de los saltos es de 1 instrucción.
Las características de las unidades funcionales son las siguientes:

	Nº FUs	latencia (ciclos)	Segmentada?	Buffers o ER
Load	1	2	No	3
+	1	3	No	2
*	1	4	No	2
Store	1	2	No	3

El código generado por el compilador es el siguiente:

```
loop: LD    F2,X(R1)
      LD    F4,Y(R1)
      ADDD  F4,F2,F4
      SD    Y(R1),F4
      BNEZ  R1,loop
      SUB   R1,R1,#8
```

Suponer que en un mismo ciclo no se puede escribir un dato en el bus común e iniciar la ejecución de una operación que estuviera en espera de ese dato.
Calcula el tiempo necesario, expresado en segundos, para procesar el fragmento del código, así como la velocidad de ejecución del mismo en MFLOPS para vectores de tamaño muy grande.

Solución ejercicio 3:
Para ver la operación de ER desarrollamos algunas iteraciones. Tienen un comportamiento repetitivo, por lo que no vamos a desarrollarlas más allá de 8 iteraciones.

ITERACIÓN 1	**ISSUE**	**EXECUTE**	**WRITE**	**Load**	**Store**	**Add**
LD F2, X(R1)	1	2-3	**4**	1	0	0
LD F4, Y(R1)	2	4-5*	**6**	2	0	0
ADDD F4,F2,F4	3	7-9$	**10**	2	0	1
SD Y(R1), loop	4	11-12$	-	2	1	1
BNEZ R1, loop	5	6	-	1	1	1
SUB R1, R1,#8	6	7	8	1	1	1

ITERACIÓN 2	ISSUE	EXECUTE	WRITE	Load	Store	Add
LD F2, X(R1)	7	9-10$	**11**	1	1	1
LD F4, Y(R1)	8	11-12*	**13**	2	1	1
ADDD F4,F2,F4	9	14-16$	**17**	2	1	2
SD Y(R1), loop	10	18-19$	-	2	2	2
BNEZ R1, loop	11	12	-	2	1	1
SUB R1, R1,#8	12	13	14	1	2	1

ITERACIÓN 3	ISSUE	EXECUTE	WRITE	Load	Store	Add
LD F2, X(R1)	13	15-17$	**18**	2	1	1
LD F4, Y(R1)	14	18-19*	**20**	2	1	1
ADDD F4,F2,F4	15	21-23$	**24**	2	1	2
SD Y(R1), loop	16	25-26$	-	2	2	2
BNEZ R1, loop	17	18	-	2	2	2
SUB R1, R1,#8	18	19-20+	21	2	2	1

ITERACIÓN 4	ISSUE	EXECUTE	WRITE	Load	Store	Add
LD F2, X(R1)	19	23-24$	**25**	2	1	1
LD F4, Y(R1)	20	25-26*	**27**	3	1	1
ADDD F4,F2,F4	21	28-30$	**31**	2	1	2
SD Y(R1), loop	22	32-33$	-	2	2	2
BNEZ R1, loop	23	24	-	2	2	2
SUB R1, R1,#8	24	25	26	2	2	2

ITERACIÓN 5	ISSUE	EXECUTE	WRITE	Load	Store	Add
LD F2, X(R1)	25	27-28$	**29**	3	2	1
LD F4, Y(R1)	26	29-31*+	**32**	3	2	1
ADDD F4,F2,F4	27	33-35$	**36**	3	1	2
SD Y(R1), loop	28	37-38$	-	2	2	2
BNEZ R1, loop	29	30	-	2	2	2
SUB R1, R1,#8	30	31-32+	33	1	2	2

ITERACIÓN 6	ISSUE	EXECUTE	WRITE	Load	Store	Add
LD F2, X(R1)	31	34-36$	**37**	2	2	2
LD F4, Y(R1)	32	37-38*	**39**	3	2	1
ADDD F4,F2,F4	33	40-42$	**43**	2	2	2
SD Y(R1), loop	34	44-45$	-	2	2	2
BNEZ R1, loop	35	36	-	2	2	2
SUB R1, R1,#8	36	37	38	2	2	2

ITERACIÓN 7	ISSUE	EXECUTE	WRITE	Load	Store	Add
LD F2, X(R1)	37	39-40$	**41**	3	2	1
LD F4, Y(R1)	38	41-43*+	**44**	3	2	1
ADDD F4,F2,F4	39	45-47$	**48**	3	1	2
SD Y(R1), loop	40	49-50$	-	2	2	2
BNEZ R1, loop	41	42	-	2	2	2
SUB R1, R1,#8	42	43-44+	45	1	2	2

ITERACIÓN 8	ISSUE	EXECUTE	WRITE	Load	Store	Add
LD F2, X(R1)	43	46-48$+	**49**	2	2	2
LD F4, Y(R1)	44	49-50*	**51**	3	2	1
ADDD F4,F2,F4	45	52-54$	**55**	2	2	2
SD Y(R1), loop	46	56-57$	-	2	2	2
BNEZ R1, loop	47	48	-	2	2	2
SUB R1, R1,#8	48	49	50	2	2	2

El símbolo * significa riesgo estructural para la memoria y $ significa dependencia verdadera.
La iteración 7 ocurre igual que la 5, pero 12 ciclos después, y lo mismo pasa con la 6 y con la 8. A partir de la iteración 8, las iteraciones impares tienen el mismo comportamiento que la 7 y las pares igual que la iteración 8.
Las instrucciones se dan a un ciclo por reloj, por lo tanto CPI=1.
Ahora vamos a calcular el tiempo de ejecución sin saber el número de iteraciones.

$$T = \text{n}^{\text{o}} \text{ ciclos} \times tc = \text{n}^{\text{o}} \text{ ciclos}/f$$

$$\text{n}^{\text{o}} \text{ ciclos}(\text{n}^{\text{o}} \text{ iteraciones} \times 6)\text{ciclos}; f = 10^9 \quad \text{ciclos}/s$$

$$T = [(\text{n}^{\text{o}} \text{ iteraciones} \times 6)\text{ciclos}]/(10^9 \quad \text{ciclos}/s) = \text{n}^{\text{o}} \text{ iteraciones} \times 6 \times 10^{-9} s$$

Ahora vamos a calcular el rendimiento en MFLOPS. Es importante tener en cuenta que las iteraciones se hace una operación en PF.

$$\text{Rendimiento} = 1 \quad \text{FLOP/iteración} = 1/6 \quad \text{FLOP/ciclo}$$

$$1/6 \quad \text{FLOP/ciclo} = (1/6 \quad \text{FLOP/ciclo}) \times (10^9 \quad \text{ciclos}/s) = 10^9/6 \quad \text{FLOP}/s$$

Tenemos el resultado en FLOP/s, por lo que lo tendremos que pasar a MFLOPS.

$$(10^9 \times 10^6)/6 = 166 \quad \text{MFLOPS}$$

Ejercicio 4. Un computador con gestión dinámica de instrucciones que usa el método de Tomasulo trabaja a una frecuencia de 1.2 GHz. El delay slot de los saltos es de 1 instrucciones y todas las unidades funcionales multiciclo son segmentadas lineales, con las siguientes etapas:

Load	ADDD	Store
3 etapas	4 etapas	3 etapas

Se pretende ejecutar el siguiente fragmento de código sobre dicha máquina:

```
        ; R1 = 256
loop:   SUB   R1, R1, #4
        LD    F0, y(R1)
        LD    F2, z(R1)
        LD    F4, t(R1)
        ADDD  F6, F4, F0
        ADDD  F8, F2, F0
        ADDD  F6, F6, F8
        BNEZ  R1, loop
        SD    x(R1),F6
```

Suponer que en un mismo ciclo no se puede escribir un dato en el bus común e iniciar la ejecución de una operación que estuviera en espera de ese dato.
¿Cuántas estaciones de reserva, buffers de lectura y buffers de escritura hacen falta para ejecutarlo lanzando una instrucción por ciclo? Calcula los MFLOPS obtenidos al ejecutar el programa.

Solución ejercicio 4:
Es importante saber que * significa riesgo estructural por acceso a memoria, y $ significa dependencia verdadera.

				ER ocupadas		
ITERACIÓN 1	**ISSUE**	**EXECUTE**	**WRITE**	**Load**	**Store**	**Add**
loop: SUB R1, R1,#4	1	2	**3**	0	0	0
LD F0,y(R1)	2	4-6$	**7**	1	0	0
LD F2, z(R1)	3	5-7*	**8**	2	0	0
LD F4, t(R1)	4	6-8*	**9**	3	0	0
ADDD F6,F4,F0	5	10-13$	**14**	3	0	1
ADDD F8,F2,F0	6	9-12$	**13**	3	0	2
ADDD F6,F6,F8	7	15-18$	**19**	3	0	3
BNEZ R1,loop	8	9	-	2	0	3
SD x(R1),F6	9	20-22$	-	1	1	3

				ER ocupadas		
ITERACIÓN 2	**ISSUE**	**EXECUTE**	**WRITE**	**Load**	**Store**	**Add**
loop: SUB R1, R1,#4	10	11	**12**	0	1	3
LD F0,y(R1)	11	13-15$	**16**	1	1	3
LD F2, z(R1)	12	14-16*	**17**	2	1	3
LD F4, t(R1)	13	15-17*	**18**	3	1	3
ADDD F6,F4,F0	14	19-22$	**23**	3	1	3
ADDD F8,F2,F0	15	18-21$	**22**	3	1	3
ADDD F6,F6,F8	16	24-27$	**28**	3	1	4
BNEZ R1,loop	17	18	-	2	1	4
SD x(R1),F6	18	29-31$	-	1	2	4

				ER ocupadas		
ITERACIÓN 3	**ISSUE**	**EXECUTE**	**WRITE**	**Load**	**Store**	**Add**
loop: SUB R1, R1,#4	19	20	**21**	0	2	4
LD F0,y(R1)	20	22-24$	**25**	1	2	3
LD F2, z(R1)	21	23-25*	**26**	2	2	3
LD F4, t(R1)	22	24-26*	**27**	3	2	3
ADDD F6,F4,F0	23	28-31$	**32**	3	1	3
ADDD F8,F2,F0	24	27-30$	**31**	3	1	3
ADDD F6,F6,F8	25	33-36$	**37**	3	1	4
BNEZ R1,loop	26	27	-	2	1	4
SD x(R1),F6	27	38-40$	-	1	2	4

La iteración de la ER es la misma en la tercera iteración y en la segunda, pero ocurre 9 ciclos más tarde. La ejecución de instrucciones también es igual en la tercera y en la segunda.
A partir de la tercera iteración, las iteraciones transcurrirán como en la tercera, pero 9 ciclos después. Por lo tanto, nos basta con 3 Load Buffers, 2 Store Buffers y 4 ER de la suma.
Ahora sabemos que hay 3 FLOP por cada iteración, por tanto, transformado en ciclos sería 3 FLOP/ 9 ciclos. Con esto ya podemos calcular el rendimiento:

$$\text{Rendimiento} = \frac{3 \quad \text{FLOP} \times 1{,}2 \times 10^9 \frac{\text{ciclos}}{s}}{9 \quad \text{ciclos}} = \frac{3600 \times 10^6 \quad \text{FLOP}}{9s} = 400 \quad \text{MFLOPS}$$

Ejercicio 7. Disponemos de un procesador dotado de planificación dinámica de instrucciones según el método de Tomasulo. Se dispone de las unidades funcionales que se muestran en la tabla siguiente:

UF	Cantidad	Latencia	Segmentación
FP ADDD	2	3	Si
FP MUL/DIV	1	4/10	No

Suponer que en un mismo ciclo no se puede escribir un dato en el bus común e iniciar la ejecución de una operación que estuviera en espera de ese dato, y que existe una estación de reserva por cada unidad funcional.
Para la secuencia de instrucciones de la tabla siguiente, indica en qué ciclo (o ciclos) de reloj se realiza cada una de sus fases. Indica qué tipo de riesgos se producen.

Instrucción	**Issue Emisión**	**Ejecución**	**Escribe resultado**
ADDD F2,F2,F4			
ADDD F0,F4,F0			
MULTD F4,F2,F0			
ADDD F6,F0,F0			
DIVD F4,F4,F6			
ADDD F2,F6,F6			
ADDD F0,F2,F6			
MULTD F2,F2,F0			

Solución ejercicio 7:

Instrucción	**Issue Emisión**	**Ejecución**	**Escribe resultado**
ADDD F2,F2,F4	1	2-4	5
ADDD F0,F4,F0	2	3-5	6
MULTD F4,F2,F0	3	7-10 (LDE)	11
ADDD F6,F0,F0	6 (ER)	7-9	10
DIVD F4,F4,F6	12 (ER)	13-22	23
ADDD F2,F6,F6	13	14-16	17
ADDD F0,F2,F6	14	18-20 (LDE)	21
MULTD F2,F2,F0	24 (ER)	25-28	29

Los riesgos de ejecución están entre paréntesis, hay un riesgo de ER y LDE.

Ejercicio 8. Disponemos de un procesador dotado de planificación dinámica de instrucciones según el método de Tomasulo. Se dispone de las unidades funcionales que se muestran en la tabla siguiente:

UF	Cantidad	Latencia	Segmentación
FP ADD/SUB	2	3/3	No
FP MUL/DIV	2	6/12	No

Suponer que en un mismo ciclo no se puede escribir un dato en el bus común e iniciar la ejecución de una operación que estuviera en espera de ese dato, y que existe una estación de reserva por cada unidad funcional.
Para la secuencia de instrucciones de la tabla siguiente, indica en qué ciclo (o ciclos) de reloj se realiza cada una de sus fases. Indica qué tipo de riesgos se producen.

Instrucción	**Issue Emisión**	**Ejecución**	**Escribe resultado**
DIVD F2,F2,F6			
ADDD F4,F6,F4			
MULD F8,F2,F4			
DIVD F0,F6,F4			
ADDD F2,F4,F0			
ADDD F8,F8,F10			
MULD F0,F2,F8			
SUBD F12,F2,F4			

Solución ejercicio 8:
El riesgo LDE se va a señalizar con * y el riesgo estructural (no suficientes estaciones de reserva) con $.

Instrucción	**Issue Emisión**	**Ejecución**	**Escribe resultado**
DIVD F2,F2,F6	1	2-13	14
ADDD F4,F6,F4	2	3-5	6
MULD F8,F2,F4	3	15-20*	21
DIVD F0,F6,F4	15$	16-27	28
ADDD F2,F4,F0	16	29-31*	32
ADDD F8,F8,F10	17	22-24*	25
MULD F0,F2,F8	22$	33-38*	39
SUBD F12,F2,F4	26$	33-35*	36

Ejercicio 9. Sea un proceso segmentado con planificación dinámica mediante el algoritmo de Tomasulo.

- Los datos que se escriben en la etapa de escritura no se pueden usar hasta el ciclo siguiente.
- Las instrucciones SGT1 y BNEZ tienen tratamiento de instrucción entera.
- Existe un único Bus de Datos Común (BDC).
- La unidad de ejecución tiene las siguientes características:

ESTACIONES RESERVA	CANTIDAD
FP ADD	2
FP DIVD	1
FP MUL	2
INT ALU	2
LOAD	2
STORE	2

UF	CANTIDAD	LATENCIA	SEGMENTADA
FP ADD	1	2	SI
FP DIV	1	6	SI
FP MUL	1	3	SI
INT ALU	1	1	SI
MEM	1	2	SI

Sea el siguiente fragmento de código:

```
LOOP: LD F2 0(R1)
      MULD F4,F2,F0
      LD F6 0(R2)
      DIVD F4,F6,F4
      SD 0(R1) F4
      MULD F4,F6,F0
      ADD F4,F8,F2
      ADDI R1,R1,#8
      ADDI R2,R2,#8
      SGTI R3,R2,DONE
      BNEZ R3,LOOP
```

Suponer que en un mismo ciclo no se puede escribir un dato en el bus común e iniciar la ejecución de una operación que estuviera en espera de ese dato.

A. Indicar el diagrama instrucción-tiempo para la primera iteracion.

B. Suponiendo que se le añade especulación basada en la utilización de un buffer de reordenamiento (ROB), indicar el diagrama instrucción-tiempo para la primera iteración. Se supone que el papel de los "store buffers" es asumido por el ROB.

Solución ejercicio 9:
Apartado A:

		ISSUE	EJECUCIÓN	WRITE
1	LD F2 0(R1)	1	2-3	4
2	MULD F4,F2,F0	2	5-6-7 (LDE CON 1)	8
3	LD F6 0(R2)	3	4-5	6
4	DIVD F4,F6,F4	4	9-14 (LDE CON 2)	15
5	SD 0(R1) F4	5	16-17 (LDE CON 4)	
6	MULD F4,F8,F2	6	7-8-9	10
7	ADD F4,F8,F2	7	8-9	11 (BCD CON 6)
8	ADDI R1,R1,#1	8	9	12 (BCD CON 6 Y 7)
9	ADDI R2,R2,#8	9	10	13 (BCD CON 7 Y 8)
10	SGTI R3,R2,DONE	13 (ESTRUCTURAL HASTA QUE LIBERA 8)	14	16 (BCD CON 4)
11	BNEZ R3,LOOP	14	17 (LDE CON 10)	18

Solución apartado B:
Con ROB no existen estaciones de reserva de store. Las instrucciones de store con dependencias del tipo LDE el dato se envía directamente al ROB. Por esto, las instrucciones de DIVD y store escriben en el ROB en el mismo ciclo de reloj.

		ISSUE	EJECUCIÓN	WRITE	COMMIT
1	LD F2 0(R1)	1	2-3	4	5
2	MULD F4,F2,F0	2	5-6-7	8	9
3	LD F6 0(R2)	3	4-5	6	10
4	DIVD F4,F6,F4	4	9-14 (LDE CON 2)	15	16
5	SD 0(R1) F4	5		15 (LDE CON 4)	17
6	MULD F4,F6,F0	6	7-8-9	10	18
7	ADD F4,F8,F2	7	8-9	11 (BCD CON 6)	19
8	ADDI R1,R1,#1	8	9	12 (BCD CON 6 Y 7)	20
9	ADDI R2,R2,#8	9	10	13 (BCD CON 7 Y 8)	21
10	SGTI R3,R2,DONE	13 (ESTRUCTURAL HASTA QUE LIBERA 8)	14	16 (BCD CON 4)	22
11	BNEZ R3,LOOP	14	17 (LDE CON 10)	18	23

Ejercicio 10. Se dispone de un procesador segmentado con un juego de instrucciones de tipo entero basadas en el procesador DLX. Las fases que atraviesa una instrucción son:

- **IF**: Búsqueda de la instrucción.
- **ID**: Decodificación de la instrucción. Cálculo de la condición de salto y dirección de destino en cada instrucción bifurcación.
- **EX**: Fase de ejecución aritmética, y cálculo de la dirección de acceso a memoria, en su caso.
- **ME**: Acceso a memoria de datos, en su caso.
- **WB**: Escritura en registros.

Dicho procesador posee un predictor del tipo **Branch Target Buffer** de **2 bits**, que se accede durante la fase IF, obteniendo la respuesta final de dicha fase.
Dada la siguiente secuencia de código, y suponiendo que el buffer del predictor contiene el estado "predicción no-salta-fuerte" para la instrucción "BNEZ R1.loop", mostrar las fases de las dos instrucciones que se ejecutan inmediatamente después de este salto, en cada una de las cuatro primeras iteraciones. En caso de que alguna instrucción se cancele, marcar con una "X" las fases que no ejecuta.

```
        ...
        ADD   R1,R0,#4
loop:   SUB   R5,R5,R2
        ADD   R2,R2,#1
        SUB   R1,R1,#1
        BNEZ  R1,loop
        AND   R6,R5,#63
        OR    R6,R6,#128
        ...
```

Solución ejercicio 10:
Vamos a suponer en lo referente a la implementación, que el contador saturado es de 2 bits, la predicción es el bit más significativo y el valor inicial del predictor es 00.
Vamos a suponer en la primera iteración que el valor inicial de R1=4, y antes del primer salto R1=3.

Primera iteración:

BNEZ	IF	ID					
AND		IF	X				
SUB			IF	ID	EX	M	WB

	Predicción	Comport. salto	Nueva predicc.
R1=3	NT	T (fallo)	01

Segunda iteración:

BNEZ	IF	ID					
AND		IF	X				
SUB			IF	ID	EX	M	WB

	Predicción	Comport. salto	Nueva predicc.
R1=2	NT	T (fallo)	10

Tercera iteración:

BNEZ	IF	ID					
AND		IF	ID	EX	M	WB	
SUB			IF	ID	EX	M	WB

	Predicción	Comport. salto	Nueva predicc.
R1=1	T	T (acierto)	11

Cuarta iteración:

BNEZ	IF	ID					
SUB		IF	X				
AND			IF	ID	EX	M	WB

	Predicción	Comport. salto	Nueva predicc.
R1=0	T	NT (fallo)	10

Ejercicio 11. Se quiere diseñar con predicción dinámica de saltos por medio de una tabla de historia de saltos. La mayoría de los programas que el procesador ejecutará están compuestos por bucles anidadados a tres niveles:

```
FOR i = 1 ... n
     FOR j = 1 ... n
          FOR k = 1 ... n
               ....
               ....
          END
     END
END
```

¿Cuantos fallos se producirán con las dos alternativas estudiadas en clase, usando 1 bit (2 estados) y 2 bits (4 estados), si en ambos casos se parte de no saltar (en el caso de 2 bits se parte del "no saltar fuerte")? Suponer n grande.

Solución ejercicio 11:
Para el predictor de 1 bit suponemos que cada vez entra en un bucle, un fallo en la primera iteración y un fallo en la última iteración.

Bucle	Nº de veces que se entra	Nº de fallos
i	1	2
j	n	$2n$
k	n^2	$2n^2$

TOTAL: $2n^2 + 2n + 2$.
Para el predictor de 2 bits, la primera vez que se entra en un bucle de dos fallos (00,01) con un total de 6 fallos iniciales (2×3) y en las sucesivas entradas en el bucle un solo fallo en la última iteración.

Bucle	Nº de veces que se entra	Nº de fallos
i	1	2
j	n	n^2
k	n^2	n^2

TOTAL: $n^2 + n + 1 + 6$.

Ejercicio 12. Supongamos un procesador con predicción dinámica de saltos que usa un predictor competitivo (Tournament Predictor) como el del Alpha 21264. Supongamos un programa que posee dos únicas instrucciones de salto con la siguiente estructura:

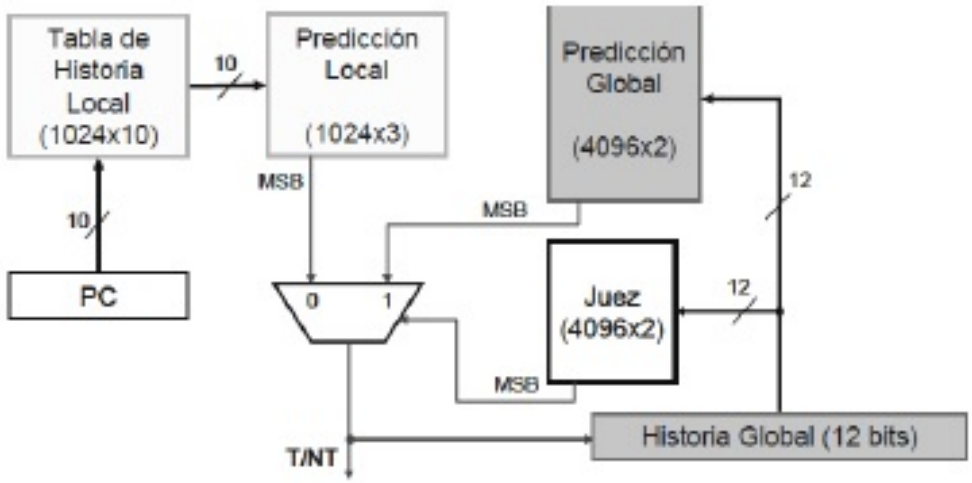
```
        Inst1
loop:   Inst2
        Inst3
        Inst4
        BEQ R1, loop
        Inst6
        Inst7
        BNE R2, loop
```

donde "Insti" representa una instrucción que no es de salto. Supongamos que la instrucción BEQ se ha ejecutado 1000 veces con el siguiente comportamiento: T-T-NT-NT-T-T-NT-NT-... El comportamiento de la instrucción BNE es que se toma siempre.

A. Determinar qué entradas de la Tabla de Predicción Local han sido accedidas como consecuencia de las 100 últimas ejecuciones de la instrucción BEQ y determinar cuál es el contenido de dichas entradas al final de las 1000 ejecuciones.

B. Si la instrucción BEQ se ejecuta 1000 veces más, siguiendo el mismo patrón de comportamiento, ¿cuántos fallos de predicción se producen en estas 1000 ejecuciones?

Solución ejercicio 12:
Solución apartado A:



Ahora vamos a ver las posiciones modificadas en la Tabla de predicción local como consecuencia de las últimas 100 ejecuciones:

Posición 204 (0011001100)
Posición 409 (0110011001)
Posición 809 (1100110011)
Posición 614 (1001100110)

Ne las posiciones 204 y 409 encontramos el 111, mientras que por otro lado, encontramos en las posiciones 819 y 614 el 000.
En las posiciones 819 y 614 el salto no es tomado y es independiente, y los bits de ambas entradas se saturarán a 000.
En las posiciones 204 y 409 el salto es tomado y ambas posiciones estarán saturadas en 111.
Solución apartado B:
El predictor acertará siempre en cada una de las iteraciones, por lo que el número de fallos será 0.

Capítulo 3

Tema 3: Lanzamiento múltiple, Límites de ILP, Multithreading

En este capítulo vamos a ver los siguientes conceptos.

- Introducción: CPI <1.
- Lanzamiento múltiple de instrucciones: Supescalar.
  - Superescalar con planificación estática.
  - Superescalar con planificación dinámica (Out-of-Order).
    - Anatomía del pipeline: alternativas.
    - Caso de estudio.
- Límites de ILP.
- Ejemplo: Implementaciones x86, evolución $\mu-$arquitecturas Intel.
- Thread Level Parallelism y Multithreading.
- Explicación de ejercicios resueltos.

3.1. Introducción: CPI <1

Con el CPI< 1 podemos lanzar y ejecutar simultáneamente múltiples instrucciones por ciclo.
Con un **procesador escalar** no se puede ejecutar más de una instrucción en, al menos, una de sus etapas, ya que, su CPI máximo ideal es 1.
En un **superescalar** no existe la anterior restricción, y tenemos los siguientes recursos:

- Más área de silicio disponible.
- Técnicas para resolver las dependencias de datos: Planificación.
- Técnicas para resolver las dependencias de control: Especulación.

3.2. Lanzamiento múltiple de instrucciones: Superescalar

IMPORTANTE

Es un desarrollo de la arquitectura con pipeline.
Las etapas del pipeline se replican y en cada ciclo de reloj se emiten varias instrucciones.
Se generan nuevos peligros por dependencias.

3.2.1. Procesador Superescalar con planificación estática

IMPORTANTE

El schedule es estático. Las instrucciones se emiten y ejecutan en orden.
La emisión es dinámica. El hardware puede decidir no emitir todas las instrucciones si hay dependencias.

Chequea riesgos en etapa ID: Capacidad para chequear conflictos de varias instrucciones a la vez.
Si una instrucción presenta conflictos, bloquea la instrucción conflictiva y pasa a las siguientes.
Implementaciones típicas: 2 instrucciones a la vez (entera, salto, memoria+punto flotante).
Caso particular: Very Long Instruction Word (VLIW).

- La etapa ID no chequea riesgos.
- El compilador conoce el HW disponible en la etapa de ejecución y sus latencias de uso → Forma "paquetes" agrupando varias instrucciones independientes que se descodifican y se lanzan a ejecución simultáneamente.
- El número de instrucciones en cada paquete es fijo. Si el compilador no encuentra suficientes instrucciones libres de riesgos para formar paquete, rellena con NOP.
- Puede adaptarse bien al ámbito de los sistemas empotrados.

3.2.2. Procesador Superescalar con planificación dinámica

IMPORTANTE

Las instrucciones se emiten en orden, pero se ejecutan fuera de orden (scheduling dinámico).

Analiza dependencias de varias instrucciones a la vez.

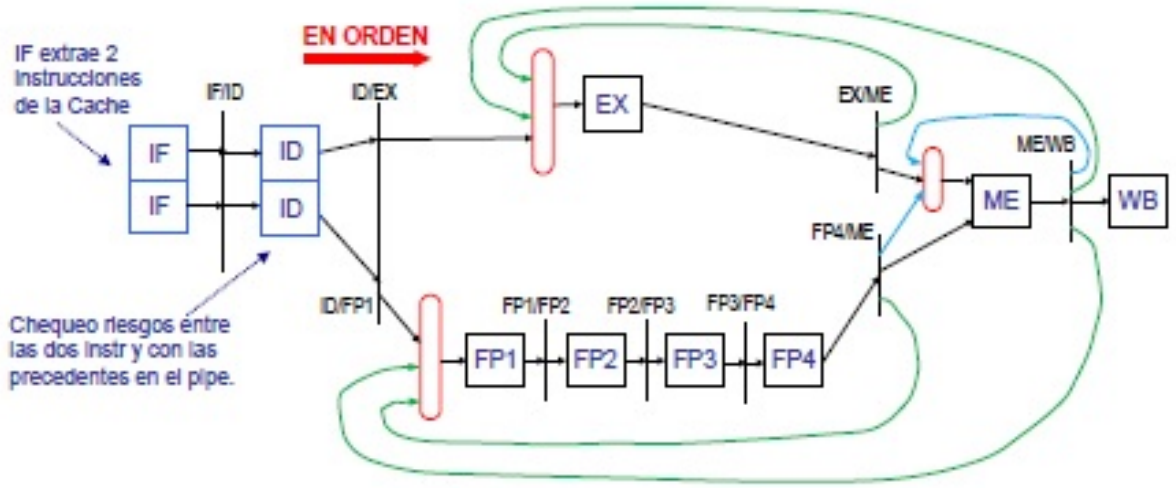
- Si conflictos → Manda a colas de espera.
- Puede mandar a las unidades funcionales (UFs) un número variable de instrucciones por ciclo (en función de la disponibilidad de operandos y de UFs).

Decodificación en orden → Ejecución en desorden → Finalización en orden.

Superescalar con planificación estática

Completando las etapas que solamente admitían una instrucción:

- Las etapas IF e ID pueden procesar dos instrucciones por ciclo.
- Si una es de FP y la otra de tipo ENT pueden enviarse en un mismo ciclo.
- Dependencias entre ENT y FP: instrucciones de LD/ST. Efecto de las dependencias provocadas por Load.
- Incremento de los problemas y provocados por los saltos.



En la etapa ME solamente el camino superior puede acceder realmente a la memoria. El otro la puntea.
En la etapa WB puede finalizar una instrucción de FP y una ENT si escriben en bancos de registros separados.
Desarrollo para ejecución superescalar: se desarrolla una iteración más.

- 5 iteraciones en 12 ciclos → 2.4 ciclos por iteración.

- CPI=12/17=0.706; IPC=1/CPI=1.417

El código máquina está compactado en la memoria: Instrucciones INT y FP intercaladas. El hw de lanzamiento detecta si puede lanzar una o dos instrucciones.

	Instrucción entera		Instrucción FP		Ciclo
Loop:	L.D	F0,0(R1)			1
	L.D	F6,-8(R1)			2
	L.D	F10,-16(R1)	ADDD	F4,F0,F2	3
	L.D	F14,-24(R1)	ADDD	F8,F6,F2	4
	L.D	F18,-32(R1)	ADDD	F12,F10,F2	5
	S.D	F4,0(R1)	ADDD	F16,F14,F2	6
	S.D	F8,-8(R1)	ADDD	F20,F18,F2	7
	S.D	F12,-16(R1)			8
	DADDIU	R1,R1,#-40			9
	S.D	F16,16(R1)			10
	BNE	R1,R2,Loop	Separadas por 2 ciclos		11
	S.D	F20,8(R1)			12

Ventajas

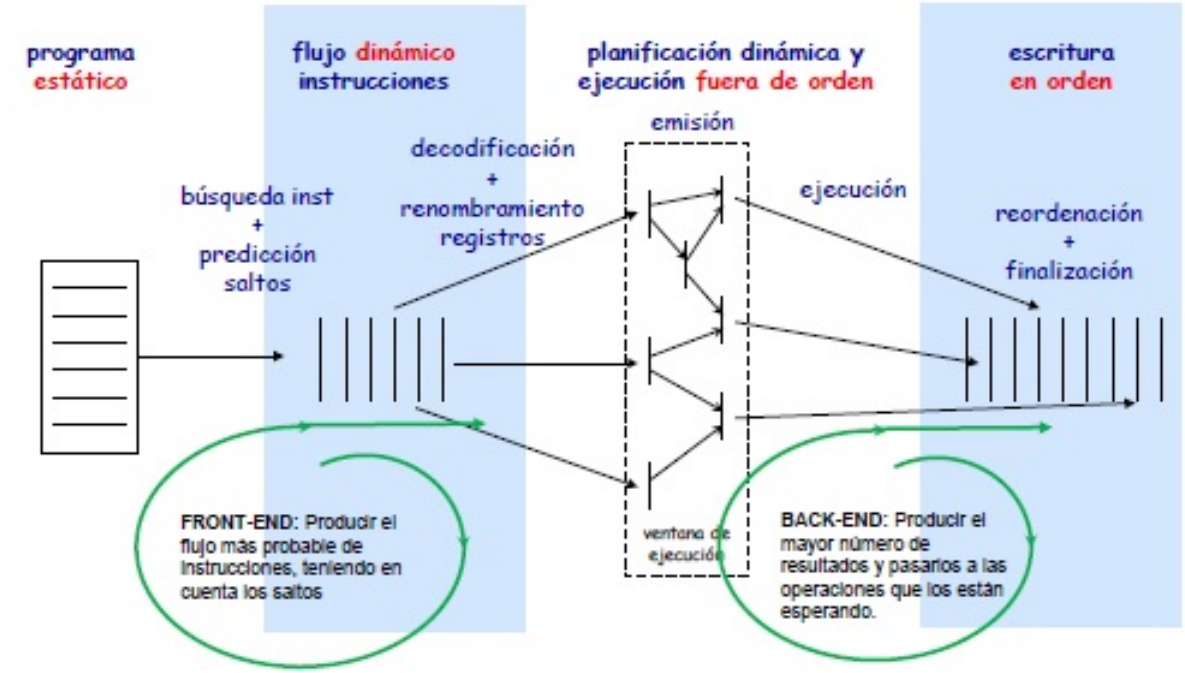
- No modifica el código. Compatibilidad binaria.
- No riesgos en ejecución: si hay riesgo → bloqueo de lanzamiento.

Desventajas

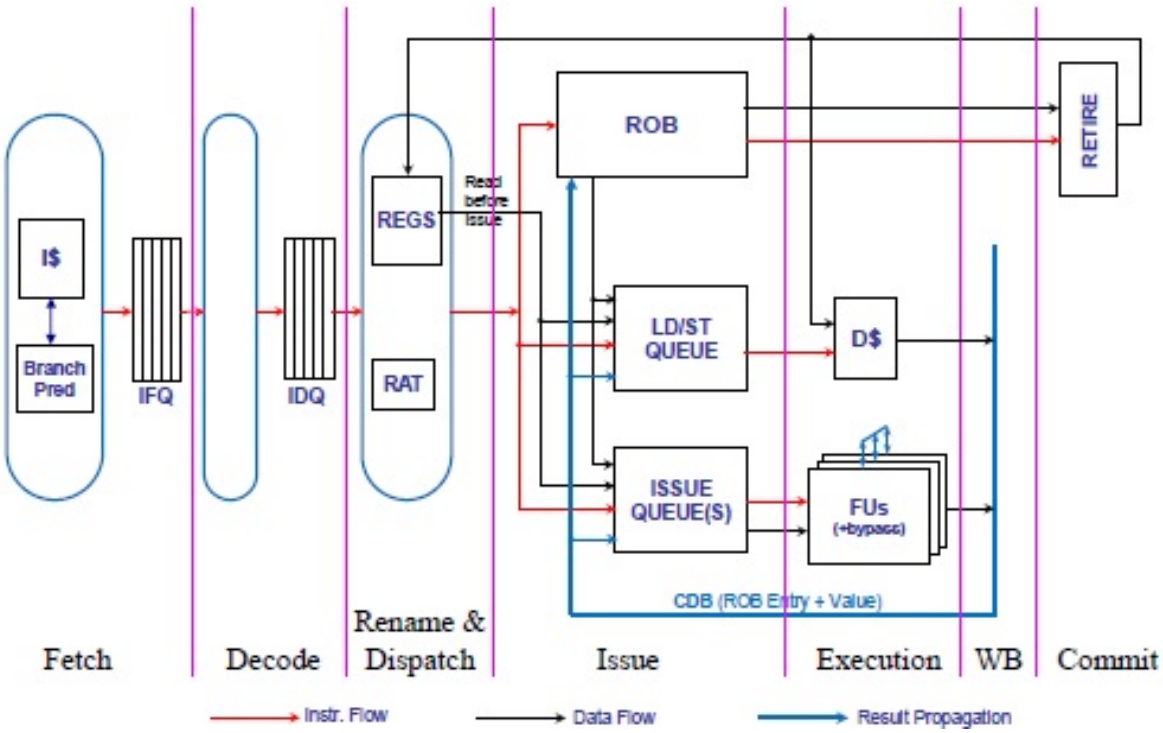
- Mezcla de instrucciones. Solo obtiene CPI de 0.5 en programas con 50 % de FP.
- Bloqueos enn el lanzamiento (igual que en MIPS básico).
- Planificación fija: No puede adaptarse a cambios en ejecución.
- Los códigos deben replanificarse para cada nueva implementación.

Superescalar con planificación dinámica

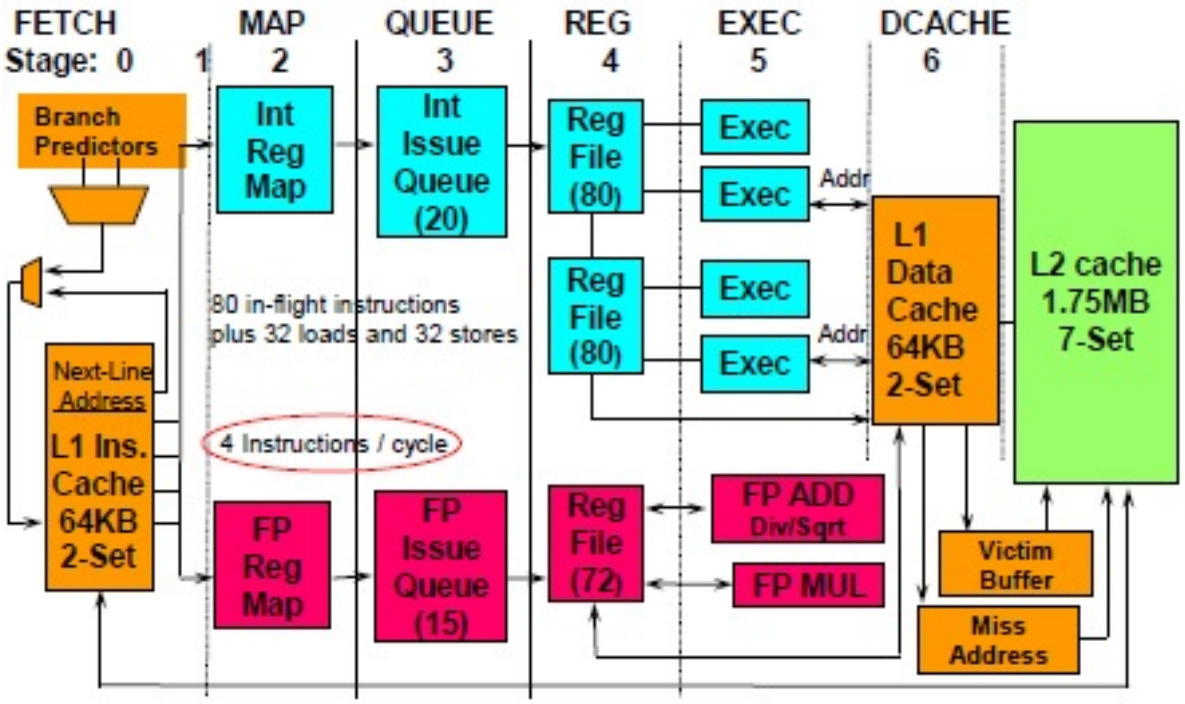
Ejecución fuera de orden. Finalización en orden.



Esquema general de organización en etapas (varias alternativas posibles):

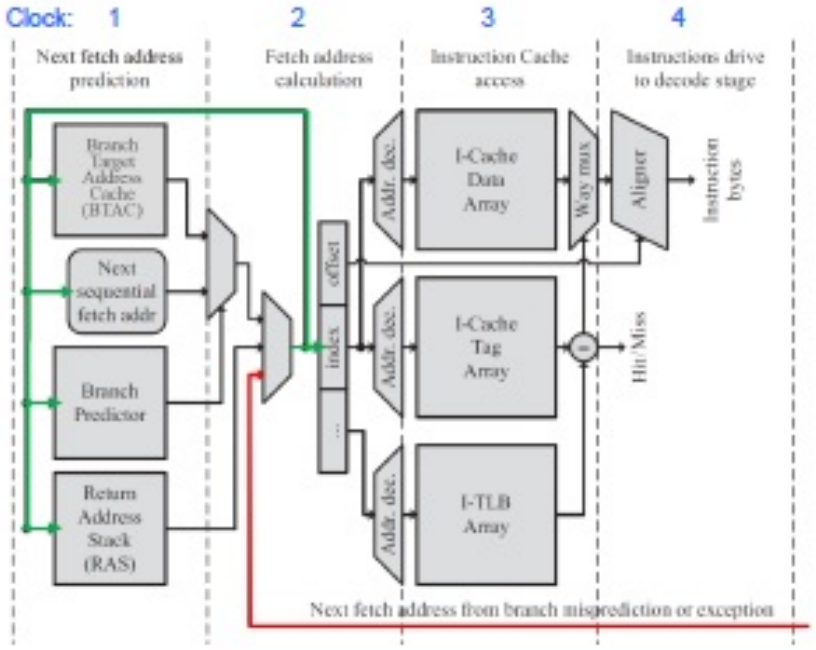


EV 7 ALPHA 21364 Core:



Fetch

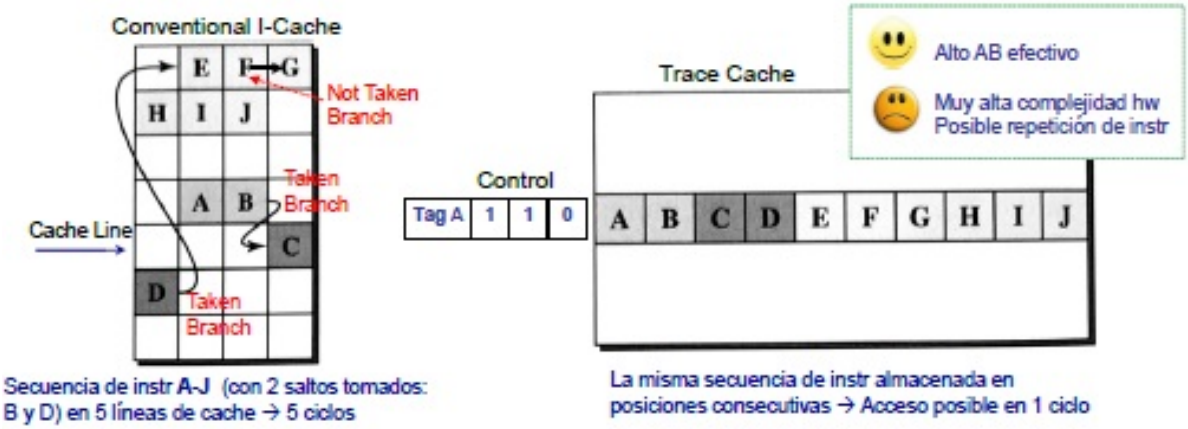
El diseño mostrado trabaja en cuatros ciclos para sostener altas frecuencias de reloj pero cada ciclo puede comenzar una nueva búsqueda.
La dirección enviada a la I-Cache (línea verde) se usa también en los predictores.
Lectura de varias instrucciones por ciclo: sencillo cuando las instrucciones están en la misma línea de cache. Crea efecto de los saltos.



Cache convencional. Acceso a grupos de instr consecutivas en memoria → Un salto tomado (incluso bien predicho) hace inútil el resto del grupo.

Idea alternativa: Trace Cache. Almacenar grupos de instrucciones consecutivas en el flujo dinámico del programa. Usada en Pentium 4.

- Control (directorio): TAG de la primera instr. y comportamiento de las instrucciones de salto (T/NT).
- El predictor debe predecir varios saltos a la vez.
- CACHE HIT: El TAG de la siguiente dirección a buscar y el comportamiento predicho para los siguientes saltos coincide con Control de una línea de la cache.
- Problema: Identificar y guardar trazas.



Decode

- Entrada: cadena de bytes extraídas de la cache.
- Salida: Instrucciones separadas+ señales control para pipe.
- Decodificación en RISC: longitud de instrucción fija+ pocos formatos → Proceso "relativamente" sencillo.
  Puede consumir un solo ciclo de reloj.

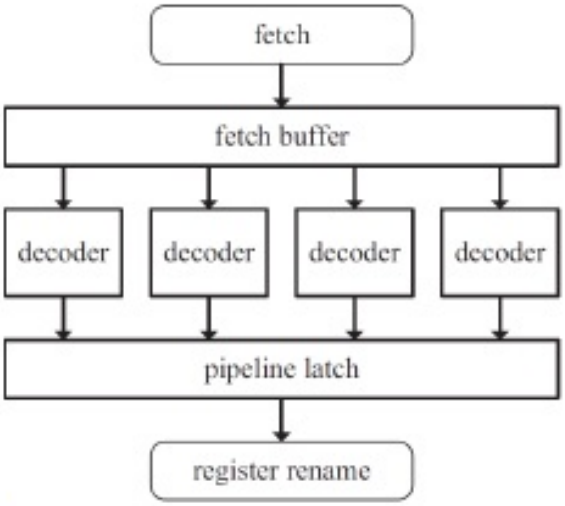


Figura 3.1: Decodifican 4 instrucciones en paralelo, que se pasan a la siguiente etapa

Segmentación con repertorios de instrucciones complejas (CISC): Caso x86.

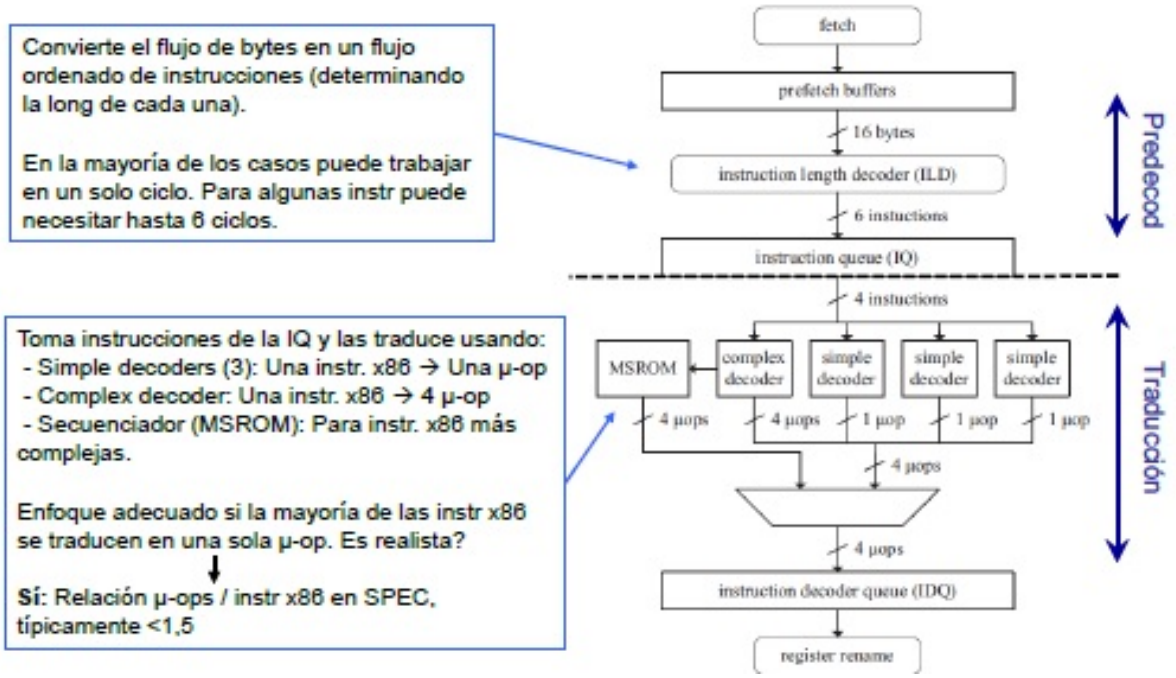
- ¿Como segmentar un repertorio de instrucciones entre 1 y 7 bytes?
  - El Opcode no está en la misma posición.
  - El Opcode no siempre tiene la misma altura.
  - Instrucciones que mezclan operandos en registros y memoria.

Traducción dinámica de instrucciones:

> IMPORTANTE
>
> El flujo de instrucciones x86 se traduce en la etapa decode o flujo de instrucciones tipo RISC.

- Primeras implementaciones: Intel P6, AMD K5.
- En P6 se traducen las instrucciones originales x86 a "$\mu-$ operaciones" de 118 bits.
- Cada instrucción original se traduce a una secuencia de 1 a 4 "μ-operaciones".
- Formato μ-op: código+ 3 operandos. Modelo: load/store. Muy descodificadas.
- Las instrucciones más complejas son traducidas por una secuencia almacenada en una memoria ROM.
- **Bucles**: posibilidad de ejecutar μ-ops repetidamente. Sucesivas iteraciones no requerían traducción dinámica.

Pipe de decodificación de Intel Nehalem (2008)



Rename & Dispatch

Rename

Asignar los elementos de almacenamiento que van a usarse para guardar el resultado (destino) de una instrucción.
Además: cambiar los nombres de registro fuente, teniendo en cuenta los renombramientos de las instrucciones anteriores (tabla).

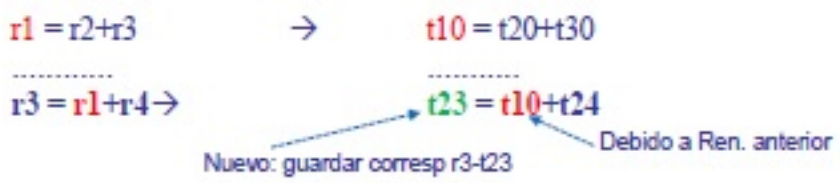


Figura 3.2: Ejemplo

Alternativas de diseño:

- Mediante el ROB: entradas del ROB guardan resultados especulados.
- Mediante un Buffer de Renombramiento:
  - Los resultados especulados no se guardan en el ROB.
  - Buffer adicional.
- Fichero de registros unificados: Solo existen los registros físicos; parte de ellos representan a los registros arquitectónicos.

Dispatch

Reservar entradas en ROB, Issue Queue(s) (y LD/ST Q para las instr de carga/almacenamiento).

- La Issue Queue se corresponde con el concepto de Estación de Reserva definido por Tomasulo en el IBM 360/91.
- Aunque los procesadores modernos esta fase se llama "dispatch", se corresponde con lo que en Tomasulo se denominó "issue".
- El término "issue" (emisión) se reserva para la fase en que una instrucción se emite hacia la correspondiente UF para comenzar la ejecución.

Lectura de operandos. Alternativas:

- Lectura antes de issue (Read before issue). Como Tomasulo:
  Operandos disponibles → copiados a Issue Q. Si no disponibles → copiar identificador del operando (nº de ROB/nº de reg físico).
- Lectura después de issue (Read after issue).
  Se propone la lectura de operandos al momento de la ejecución.

Renombrado de un grupo de instrucciones en paralelo:

- El problema de las tablas de renombrado solamente contienen información sobre las instrucciones anteriores.
- Pero las instrucciones deben tener en cuenta el renombrado de las instrucciones anteriores.

Alternativas:

- Renombrado secuencial: Multiplicar (en la medida de lo posible) la frecuencia de reloj de esta etapa por un factor igual al nº de instr, a renombrar.
- Renombrado combinacional: teniendo en cuenta los operandos y destinos de todas las instr del grupo.

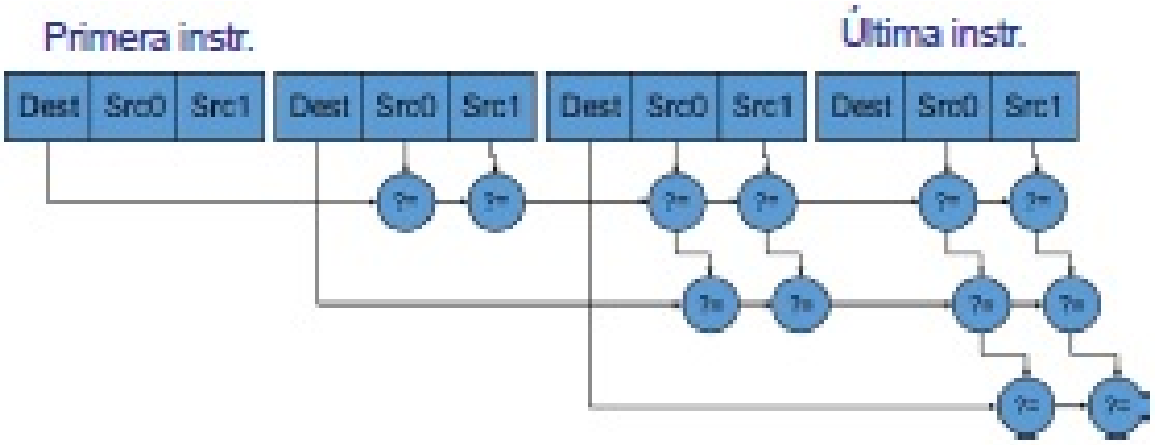
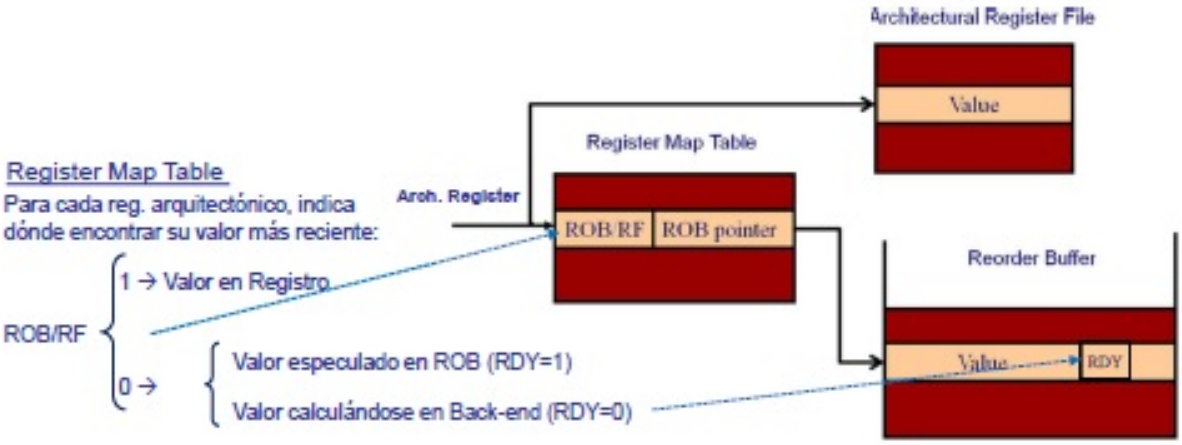


Figura 3.3: Chequeos necesarios para renombrar un grupo de instrucciones

Renombrado mediante ROB: Esquema similar usado en Intel Core 2.



Alternativa: Buffer de Renombramiento (esquema usado en IBM Power 3).

- No todas las instrucciones producen un resultado → Desperdicio de espacio en el ROB.

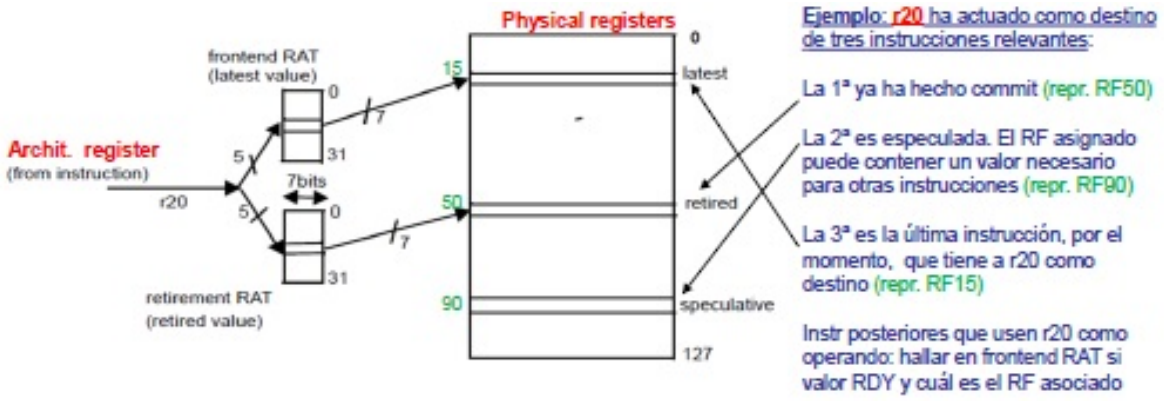
- Idea: En el ROB se guardan punteros a un conjunto auxiliar de registros (Buffer de Renombramiento) en lugar de valores.
- Los valores se guardan en el Buffer de Renombramiento: Asignar entradas en dispatch, liberar en commit.

Fichero de registros unificado:

- Los resultados, especulados o no, se guardan en un único fichero de registros (registros físicos, RF), con un nº registros mayor que nº registros arquitectónicos (RA).
- El ROB se limita a mantener el orden de las instr, pero no almacena valores.
  - Ventaja: en la fase commit no hay que copiar resultados del ROB a registros.
  - Desventaja: Cada entrada del ROB guarda el nº de RF donde se debe almacenar el resultado de la operación y el nº del correspondiente RA (Commit).
- Los RA se mapean dinámicamente sobre los RF: En cualquier momento un mismo RA puede estar mapeado sobre varios RF.
- Debe existir un RF encargado de almacenar el último valor asignado no especulativamente (en commit) a cada RA.
- Un RF quedará libre (reutilizable) cuándo contenga un valor que ya no vaya a ser utilizado: mantener lista de RF libres.
  Si RFi se está usando para contener el valor no especulado de RAj y se ejecuta la fase commit de una instrucción cuy reg destino es también RAj → el valor de la RFi ya no se necesita → RFi puede reutilizarse.

Ejemplo: 32 RA y 128 RF.

- **Frontend RAT**: Para cada RA, indica el RF asignado la última vez que dicho RA actúo como destino de una instr. +Bit RDY.
- **Retiremeent RAT**: Para cada RA, indica el RF que contiene su último valor no especulado (puede ser el mismo).



- A cada reg destino se le asigna un RF que esté libre (actualizar la frontend RAT). Este número de RF actuará como TAG (en lugar del nº del ROB asignado).

- Para los registros operando: encontrar RF asociado a cada operando en frontend RAT.

- Si valor disponible, mandar Issue Q (lectura de Issue). Si no, mandar nº de RF (tag).

- Commit: significa básicamente actualizar la retirement RAT.

Supongamos que el programa:

Supongamos que Cod1 hace COMMIT → Asignar: Retirement RAT(20)=50.

- Todavía puede haber instr en Zona A que precisen RF50 como operando.

- RF50 representa el valor no especulado de r20.

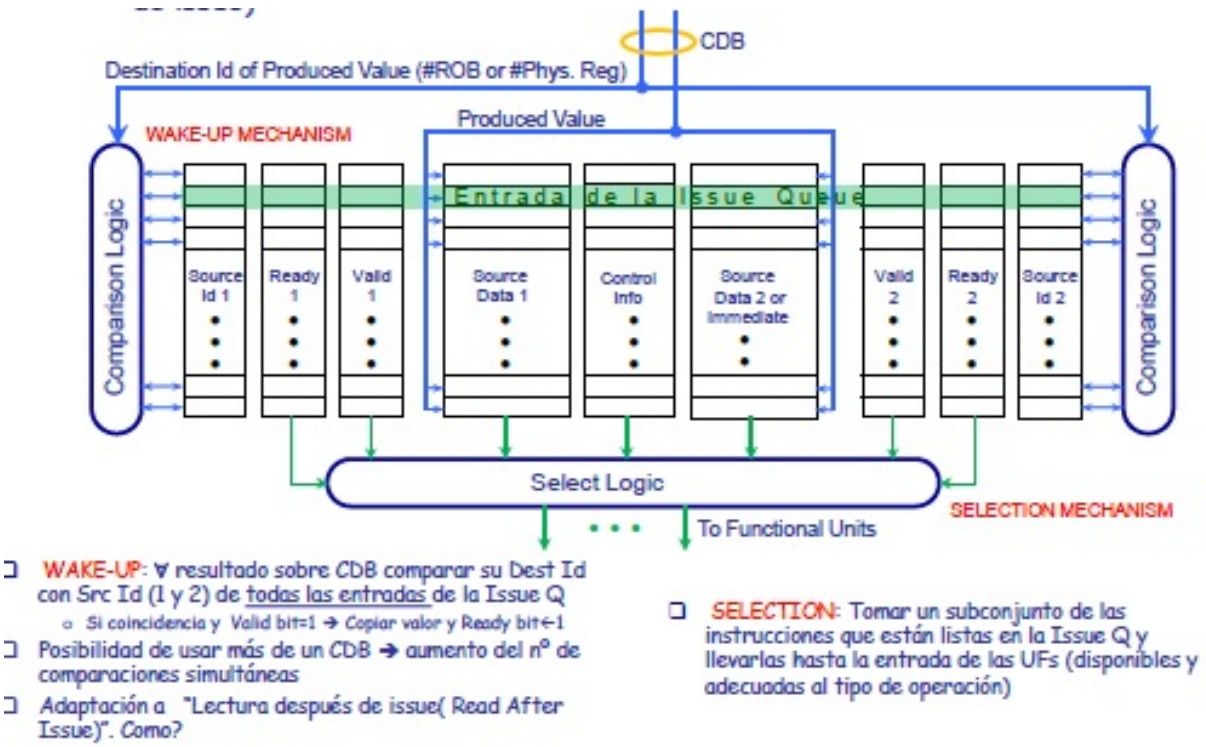
Supongamos que Cod2 hace COMMIT → Todas las instr de Zona A han hecho COMMIT.

- Como Retirement RAT(20)=50→ RF50 ya no tiene utilidad. Poner RF50 en lista de RF libres.

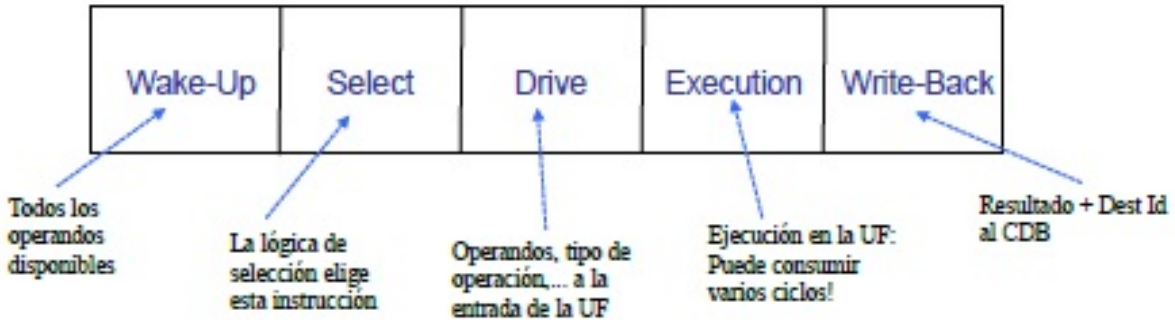
- Asignar: Retirement RAT(20)=90. El valor no especulado de r20 queda representado por RF90.

Issue

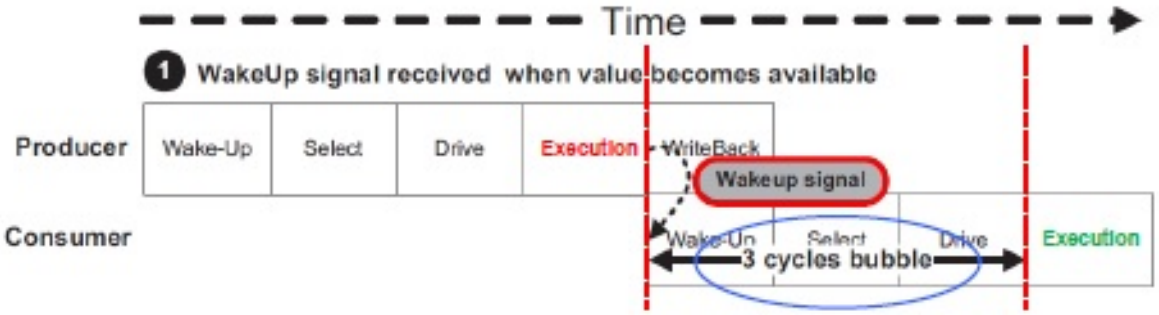
Esquema general de una Issue Queue (con lectura de operandos antes de issue).



De wake-up a write-back:

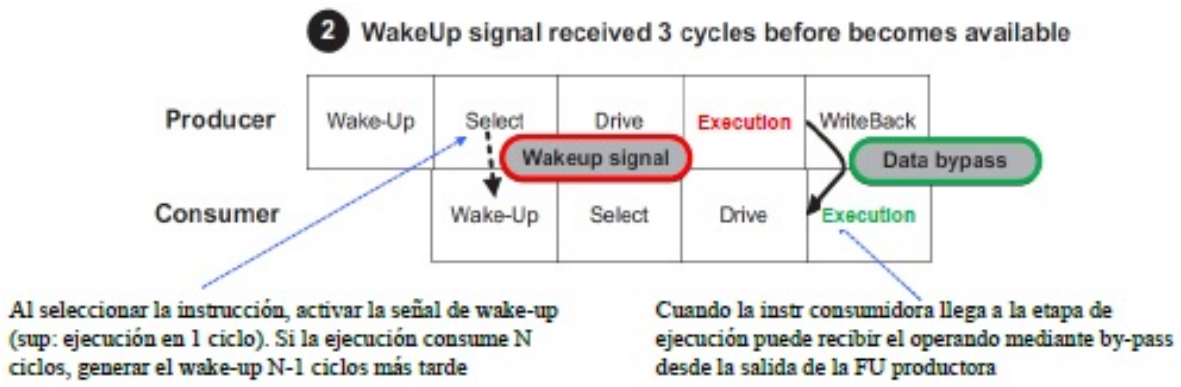


El problema: penalización en instr próximas con dep LDE.



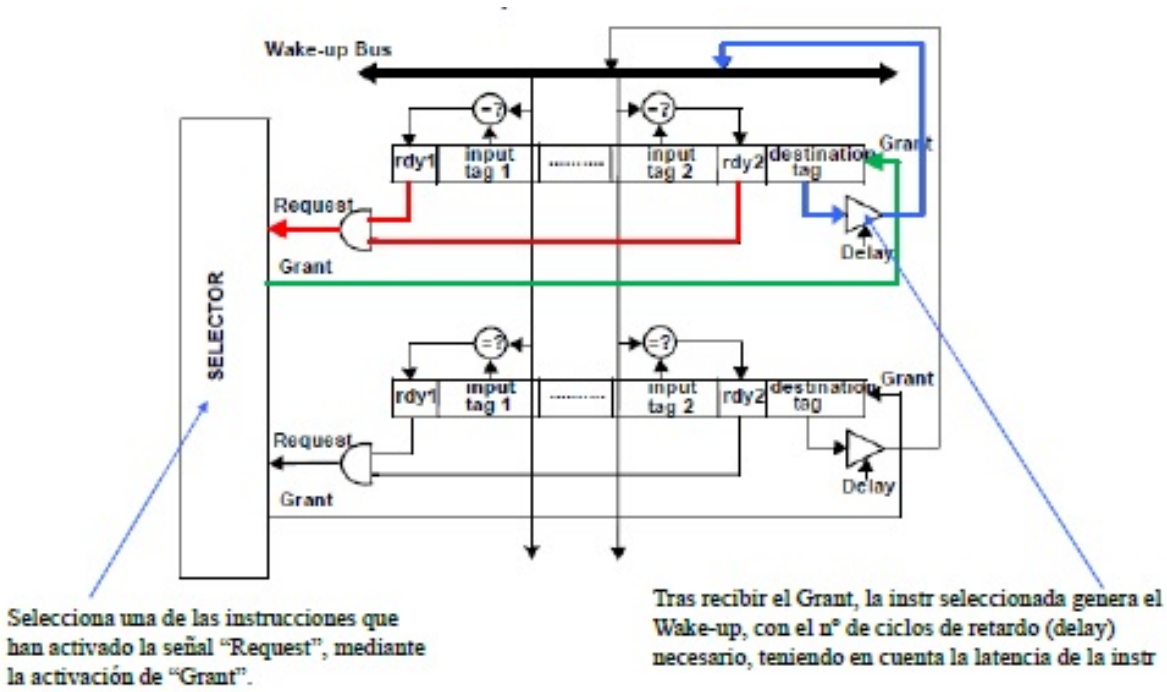
Alternativa: wake-up anticipado.

- las instr aritméticas tienen un tiempo de ejecución conocido de antemano.
- Despertar la instr consumidora varios ciclos antes (dependiendo de la latencia de la productora).

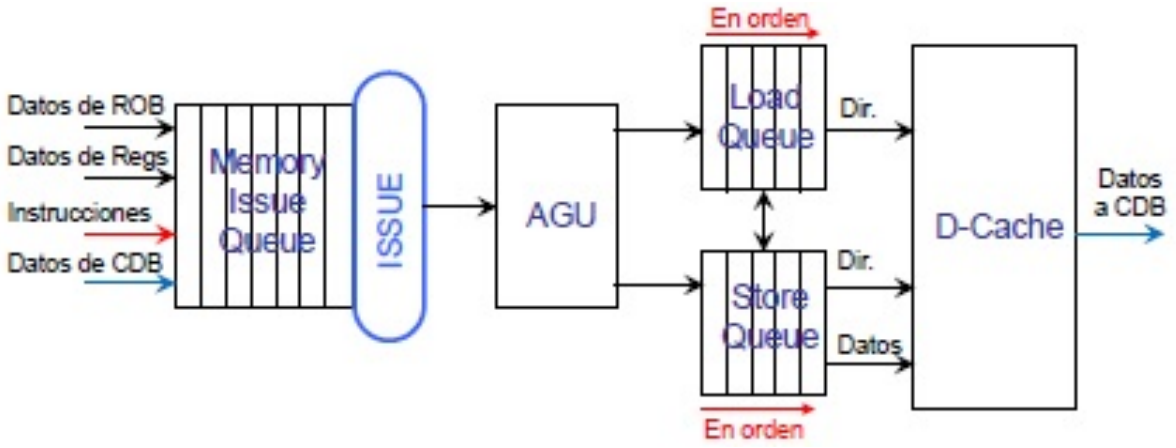


- Cero ciclos de penalización (ejecución "back-to-back") para instr que se ejecutan en un ciclo de reloj
- Debe garantizar el timing previsto (disponible de FU, conflictos de CDB...)
- Problemas cuando el productor es un Load: latencia variable (hit/miss). Necesidad de un mecanismo de reparación.

Ejemplo: selección de 1 instr en cada ciclo. Se muestran 2 entradas de la Issue Q.



Tratamiento de Load y Store: Un posible diseño del bloque genérico LD/ST Q mostrado en la tr.9.



Memory Issue Q: Cada entrada contiene una instr de LD/ST.

- La instr permanece en la cola hasta que captura la información necesaria para calcular la dirección (y para ST también el dato).
- Las entradas como en la Issue Q de las op aritméticas, no tienen porque estar en orden.

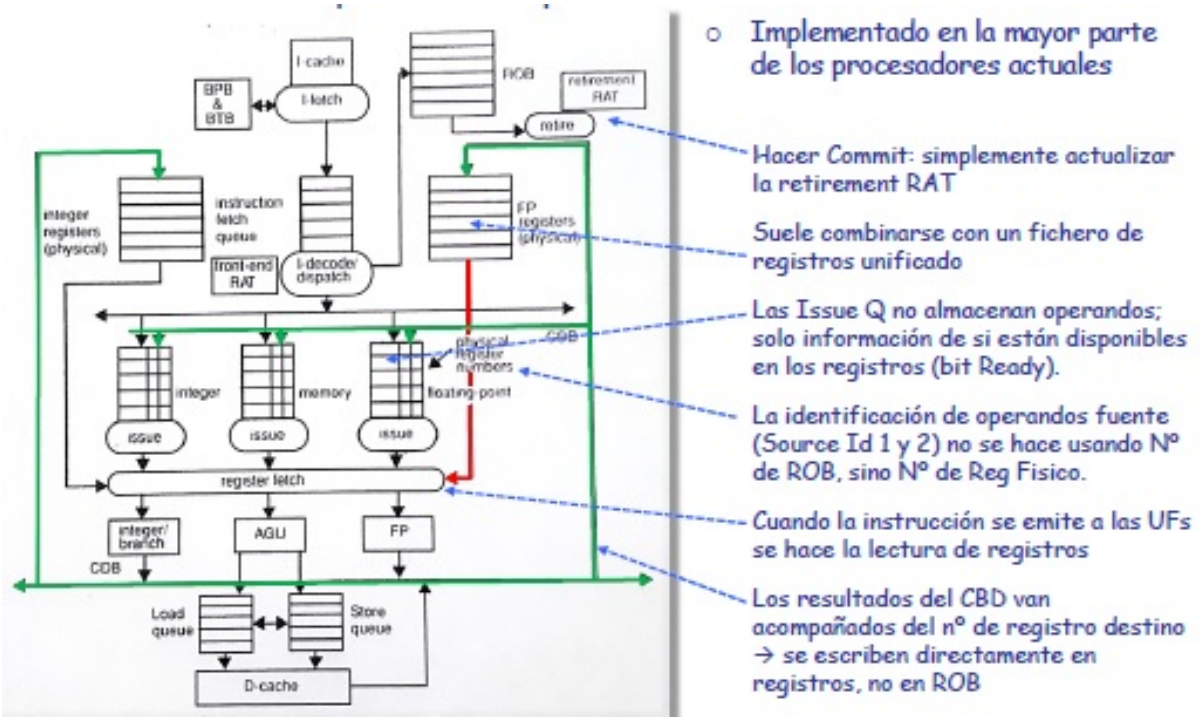
AGU: Address Generation Unit. Calculas las dir y los pasa a la entrada correspondiente de la Load o Store Q.
Store Q: Mantiene los ST en orden. A cada ST se le asigna una entrada en orden en la fase de dispatch (en ese momento es posible que no esté disponible la dir ni el dato).
Load Q: Mantiene los LD en orden. Cada LD se inserta con un TAG que indica la posición en la Store Q del último ST insertado → Permite conocer cuáles son los ST más antiguos que el LD.
Tratamiento de Load y Store:

- STORE:
  - Dispatch:
    - Asignar entrada en ROB en orden.
    - Buscar entrada libre en Memory Issue Q (almacenar inst+operandos para calcular la dirección+dato; o Tags que permitan obtenerlos más tarde).
    - Asignar entrada en Store Q en orden: Contendrá la dir y el dato del ST cuando estén calculados.
  - Issue:
    - Cuando los operandos necesarios están disponibles, se puede mandar la instr a la AGU, y guardar la dir efectiva en la correspondiente entrada de la Store Q.
    - La entrada correspondiente de la Memory Issue Q queda libre.
    - Alternativa: puede ser interesante guardar la dir de un ST en la Store Q, aunque no se disponga todavía del dato. Permite analizar riesgos con los LD más pronto → seguir manteniendo la entrada asignada en la Memory Issue Q.
- LOAD: Dispatch e issue: Tratamiento similar a los ST en cuanto a la dirección.

Desambiguación dinámica de memoria: Comparación de direcciones de LD y ST para evitar riesgos a través de memoria.

- Un ST en la Store Q solamente puede ejecutar en la escritura D-Cache si tiene su data+dirección y está en la cabecera del ROB.
    - Todos los LD y ST anteriores hacen commit → las dependencias WAR y WAW no provocan ningún riesgo.
    - Obviamente, el único ST que puede acceder a la D-cache está también en la cabecera de la Store Q.
    - Un LD puede ejecutar su lectura en la D-Cache tiene su dirección y está dirección no coincide con la de ningún ST más viejo en la Store Q (búsqueda asociativa). Se garantiza la ausencia de riesgos LDE a través de la memoria.
- ¿Qué ocurre con un LD cuando existen ST más viejos cuya dirección aún no se conoce?
    - Enfoque conservador: Esperar a conocer esas direcciones.
    - Enfoque optimista: **Load especulativo**.
        - Fundamento: es raro que un LD dependa de un ST próximo → Apostar a que no va haber coincidencia de direcciones (no riesgo LDE) (Guardar dirección del LD).
        - Mecanismo de recuperación: Si cuando se conoce la dirección de un ST hay coincidencia con un LD ya ejecutado, repetir el LD y las instrucciones siguientes (similar a un salto mal predicho).
        - Mecanismo de prevención: Mantener en Fetch una tabla de predicción de Load dependientes, indexada por el PC, con 1 bit por fila (similar a un predictor de saltos).
            - Load mal especulado → poner el 1 bit correspondiente en la tabla.
            - Sucesivas ejecuciones del mismo Load → bit a 1 → Tratamiento conservador.
            - Borrar periódicamente el contenido de la tabla.

Lectura de operandos después de Issue



Execute&WB

Esquema general de las Unidades.
Funcionales en su contexto.

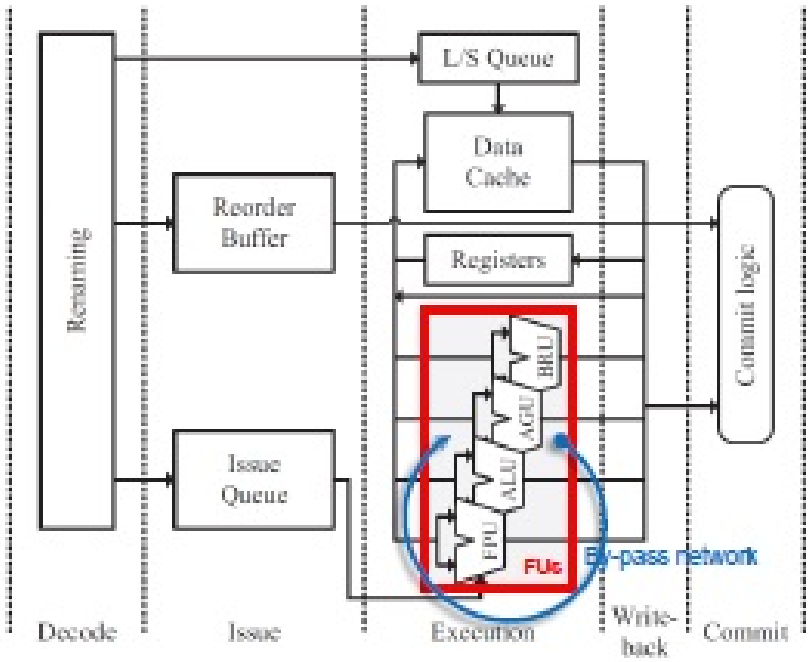


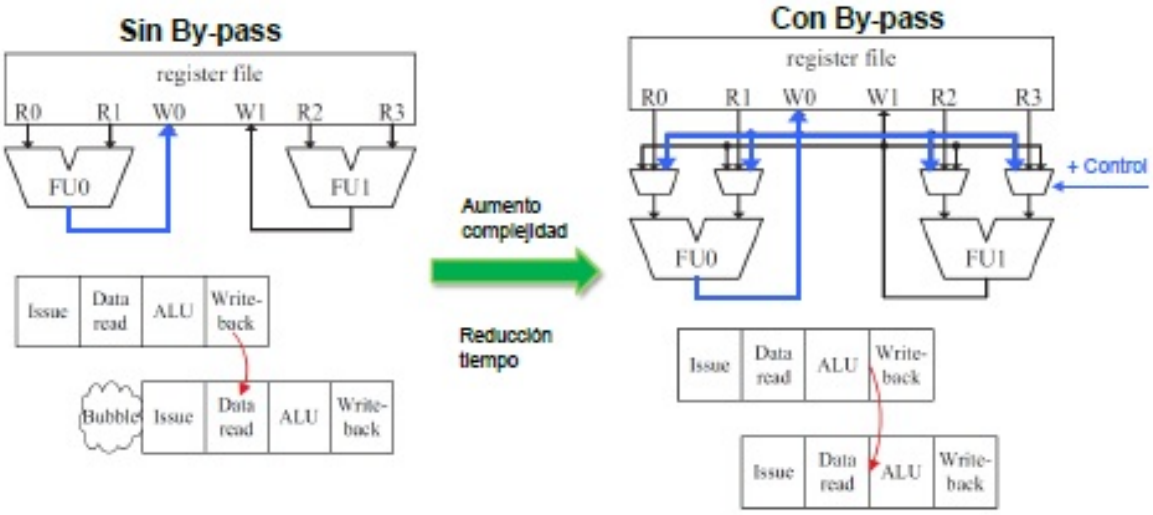
Figura 3.4: Se asume un fichero de registros unificado

Las FUs operan sobre los datos que se hacen llegar a sus entradas, junto con la operación a realizar.
Cuando un resultado está listo (uno o varios ciclos de reloj más tarde) se envía a los

elementos de almacenamiento previstos en la instrucción (Write-back) y se realiza el "wake-up" de instrucciones a la espera del resultado.
El resultado se puede mandar también directamente a las entradas de las FUs que los van a necesitar en el ciclo siguiente, mediante un mecanismo combinacional de anticipación de operandos (by-pass).
By-pass: compromiso entre la complejidad combinacional (puede empeorar ciclo de reloj, consumo) y los ciclos de espera entre instrucciones dependientes (mejora IPC).

- La mayoría de los procesadores actuales usan algún mecanismo de by-pass.
- Hay excepciones importantes (p.ej. IBM POWER5, para disminuir complejidad y aumentar frecuencia de reloj).

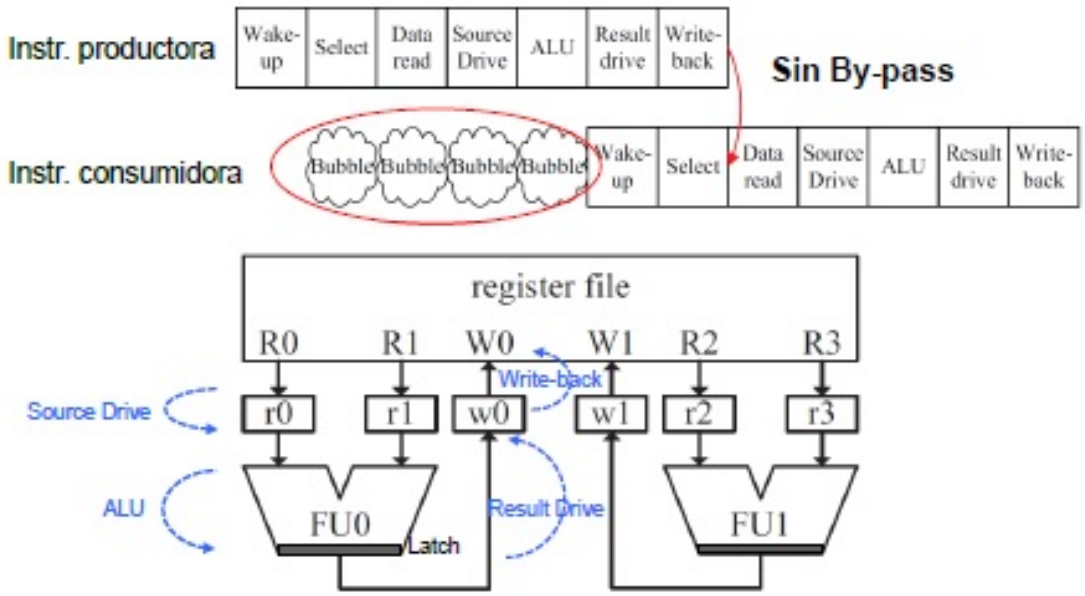
By-pass en un pipeline sencillo. 2 FUs y RF con 4 puertos de lectura/ 2 de escritura.



By-pass en procesadores con alto grado de segmentación. Camina entre RF y FUs segmentado.

- Más complejidad.
- Sin by-pass: mayor retraso de operaciones dependientes próximas.

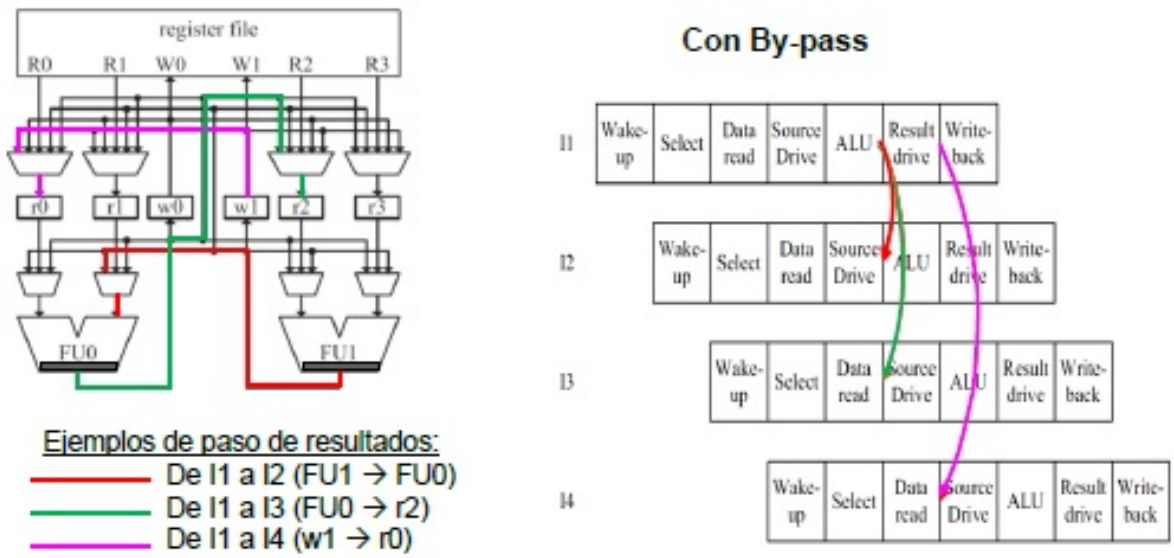
Ejemplo:



Red de by-pass con 2 FUs.

- Aumenta complejidad.
  Alternativa. Clustering: By-pass posible solo entre grupos de FUs.

- Paso del resultado de I1 a I2, I3 e I4.



Commit

Estado arquitectónico: formado por el contenido de la memoria + contenido de los registros utilizables para codificar instrucciones en LM (es decir, registros arquitectónicos o visibles al programador).
Garantiza que el estado arquitectónico, desde el punto de vista de un programador en LM, se actualiza en el mismo orden que en un procesador totalmente secuencial.

- Las instrucciones hacen la fase commit (es decir, se retiran) en el mismo orden en que se ejecutan en el flujo dinámico del programa (recordar que en dispatch se insertan en el ROB en dicho orden).

- Una instrucción I se retira cuando está la cabecera del ROB: Actualiza el estado arquitectónico del procesador. Ejemplo: instr ADD.D F2,F4,F6.

  - **Con valores en el ROB**: Copiar el valor que hay en cabecera del ROB en F2. El resultado del ADD.D se escribe 2 veces: Una en ROB (en WB), otra en F2 (en commit).
  - **Con Fichero de Registros Unificado**: Sup que en dispatch en el Reg. Arq. F2 quedó representado por RF40 → El contenido de la fila 2 de la Retirement RAT pasa a ser 40. El resultado del ADD.D sólo se ha escrito una vez en RF40.

- Las instrucciones más antiguas que I se han retirado → no pueden activar excepciones: Se han ejecutado completamente.

- Las instrucciones más modernas que I están produciendo/han producido resultados especulativos → no modifican el estado arquitectónico.

- Si la instrucción I es un salto mal predicho, el contenido del ROB y todas las instrucciones en las colas deben eliminarse. El ROB debe guardar la predicción de los saltos para poder compararlo en commit (o antes) con el comportamiento real.

Suposición: Fichero de Reg Unificado.

- Restaurar la Frontend RAT. Todos los renombramientos hechos corresponen a instr en ROB (a eliminar) o finalizadas.

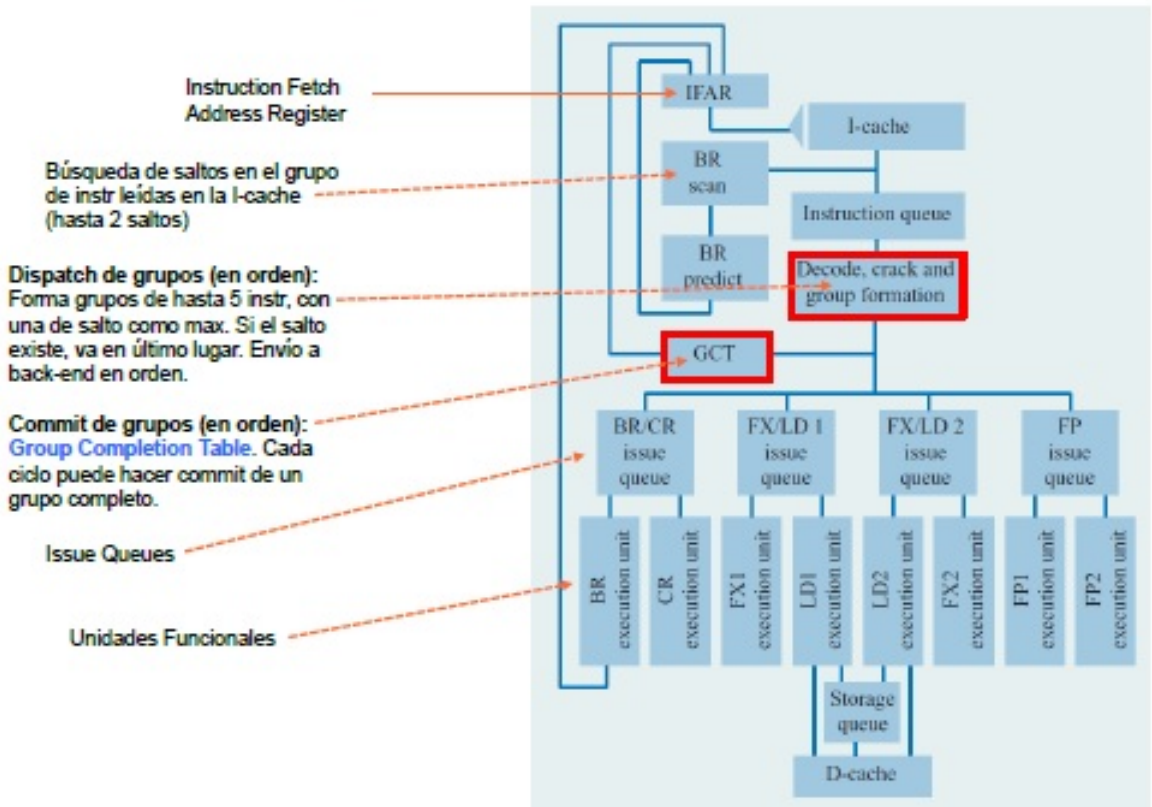
- Solución más simple:

  - La retirement RAT no se ve alterado por el fallo de predicción.
  - Los registros indicados en la retiremetn RAT son los que deben suministrar los operandos a las nuevas instrucciones que referencien dichos registros → Basta con copiar la retirement RAT sobre la frontend RAT y comenzar a buscar instrucciones por la rama correcta.
  - Problema: Gran número de instrucciones en las colas → muchos ciclos de reloj perdidos, mucha energía gastada en trabajo inútil.

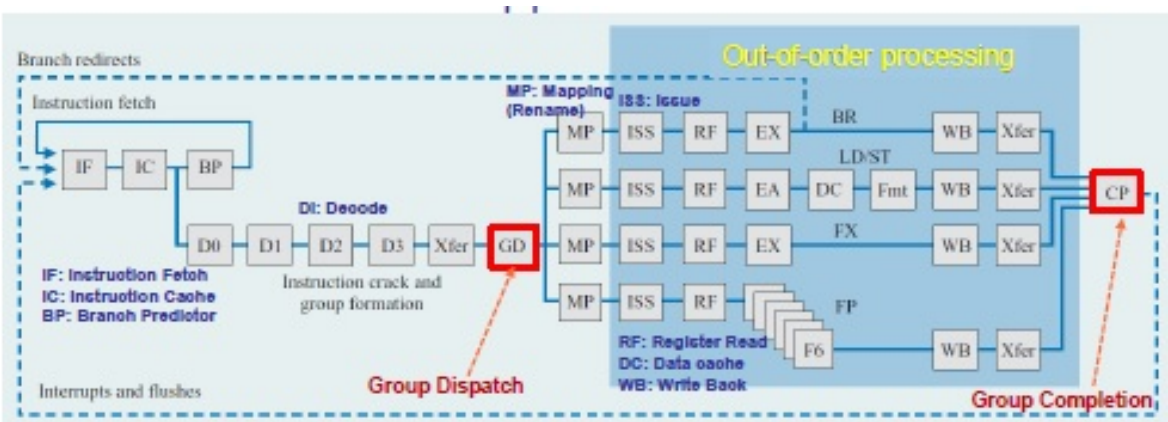
- Soluciones más ágiles: Anticipar detección fallos mal predichos.

  - Ejecutar y finalizar las instr más antiguas. Desechar el salto y todas las instr más jóvenes.

POWER4: Diagrama de bloques.



Estructura del pipeline:



Organización de 1 grupo:

IOP1	IOP2	IOP3	IOP4	BR / NOP

- IOP: operación interna. En general es una instr del LM. Pero algunas instr complejas pueden descomponerse en varias IOPs.
- Solo hay un salto como máximo y va en la última posición. Si aparece antes, entonces los huecos IOPi se rellenan con NOP y el salto se pone en la última posición.
- Dispatch de un grupo precisa: 1 entrada libre en GCT, 1 entrada libre para cada instr del grupo en la Issue Q correspondiente, 1 registro libre para renombrar el reg destino de cada instr del grupo, 1 entrada libre en LD o ST Q para cada instr de LD o ST del grupo.
- La GCT puede alojar hasta 20 grupos. Cuando todas las instr del grupo han finalizado, el grupo entero se retira (o se descarta)

3.3. Límites del ILP

Ventaja: El lanzamiento múltiple permite mejorar el rendimiento sin afectar al modelo de programación.
Desventajas: En los últimos años se ha mantenido el mismo ancho superescalar que tenían los diseños del 1995 y la diferencia entre rendimiento pico y rendimiento obtenido crece.
Suponemos que un procesador superescalar fuera de orden con especulación y con recursos ilimitados:

- Infinitos registros para renombrarlo.
- Predicción perfecta de saltos.
- Caches perfectas.
- Lanzamiento no limitado.
- Desambiguación de memoria perfecta.

Entonces:

- En tal sistema (teórico) la ejecución de instrucciones está solo condicionada por las dependencias LDE.
- Suponemos que la latencia de las UFs es 1.

3.3.1. Comparaciones necesarias en lanzamiento

Ejemplo: Lanzar 6 instrucciones/ciclo (check LDE).

- 1-2 1-3 1-4 1-5 1-6 = 5 instr x 2 operandos = 10 comp
- 2-3 2-4 2-5 2-6 = 4 instr x 2 operandos = 8 comp
- ....
- 5-6 = 1 instr x 2 operandos = 2 comp
- TOTAL = 30 comp

Lanzar n instrucciones implica $n^2 - n$ comparaciones.

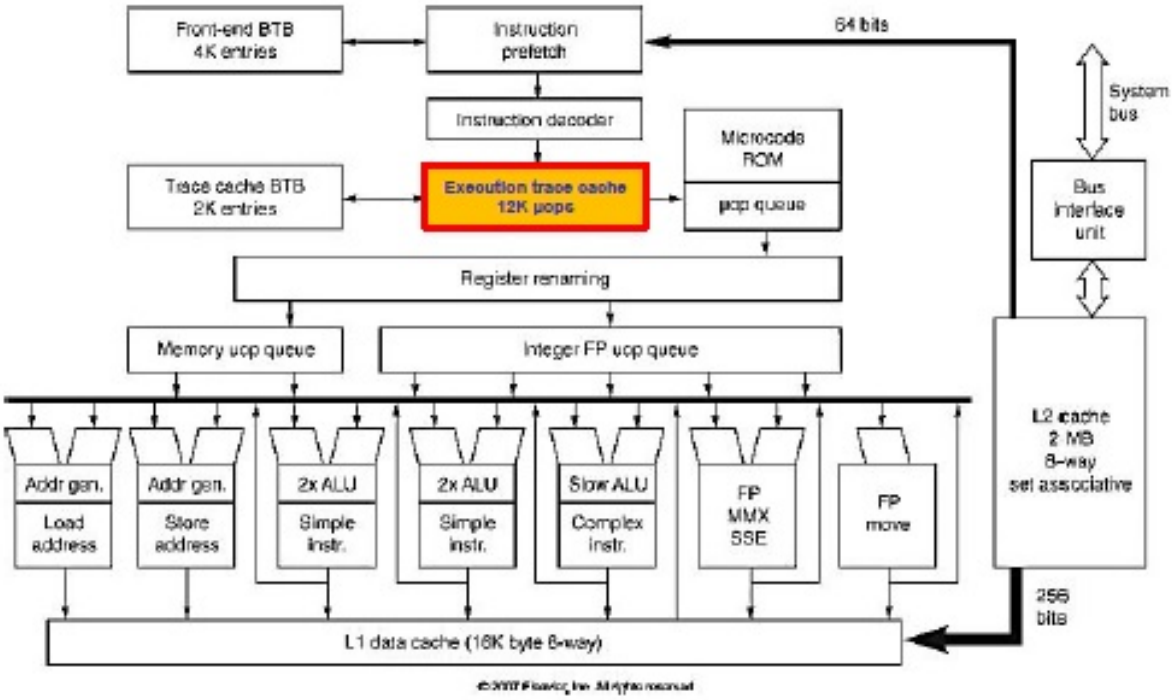
3.3.2. Comparaciones necesarias para paso de operandos

Ejemplo: Pueden acabar 6 instr/ciclo, 80 instr en la ventana, 2 operandos/instr.
$6 \times 80 \times 2 = 960$ comparaciones/ciclo.
Comparamos los operandos de todas las instrucciones que están esperando a ser ejecutadas.

3.3.3. Modelo versus procesador real

	Modelo	Power 5
Instrucciones lanzadas por ciclo	Infinito	4
Ventana de instrucciones	Infinita	200
Registros para renombrado	Infinitas	48 integer+ 40 Fl. Pt.
Predicción de saltos	Perfecta	2 % a 6 % de fallos de predicción
Cache	Perfecta	64 KI, 32 KD, 1.92 MB L2, 36 MB L3 (off chip)
Análisis de Memory Alias	Perfecto	

Ejemplo: evolución procesadores Intel-Pentium 4 (Netburst)



3.3.4. ¿Cómo superar los limites de este estudio?

- Mejoras en los compiladores e ISAs.
- Eliminar riesgos EDL y EDE de la memoria.
- Eliminar riesgos LDE en registros y memoria. Predicción.
- Buscar otros paralelismos (thread).

3.3.5. Buscar paralelismo de más de un thread

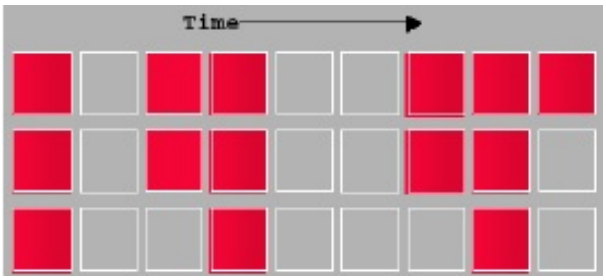
Hay mucho paralelismo en algunas aplicaciones (bases de datos, códigos científicos).
Thread Level Paralellism:

- **Thread**: Proceso con sus propias instrucciones y datos.
  - Cada thread puede ser parte de un programa paralelo de múltiples procesos, o un programa independiente.
  - Cada thread tiene todo el estado (instrucciones, datos, PC, register state,...) necesario para permitir su ejecución.
  - Arquitecturas (multiprocesadores, MultiThreading y multi/many cores).
- **Data Level Parallelism**: Operaciones idénticas sobre grandes volúmenes de datos (extensiones multimedia y arquitecturas vectoriales).

3.3.6. Thread Level Parallelism vs ILP

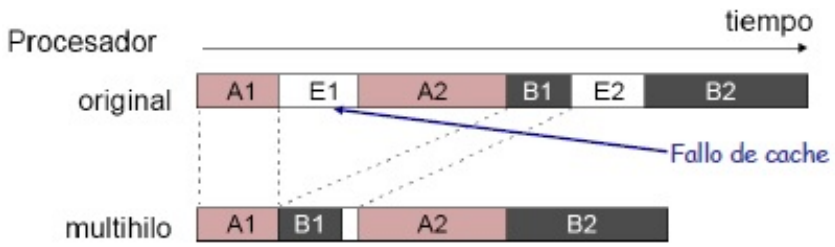
- ILP explota paralelismo implícito dentro de un segmento de código lineal o un bucle.
- TLP representa el uso de múltiples thread que son inherentemente paralelos.
- Objetivo: Usar múltiples streams de instrucciones para mejorar:
  - Throughput de computadores que ejecutan muchos programas diferentes.
  - Reducir el tiempo de ejecución de un programa multithreaded.
- TLP puede ser más eficaz en coste que ILP.

3.4. Multithreading



Causas de baja utilización de UFs:

- No hay instrucciones que ejecutar (cache).
- Hay instr pero faltan operaciones.
- Hay UFs libres, pero no del tiempo necesario.



- Incrementar el trabajo procesado por unidad de tiempo.
- Si los hilos son del mismo trabajo se reduce el tiempo de ejecución.

La técnica multithreading no es ideal y se producen pérdidas de rendimiento.

3.4.1. Ejecución Multithreaded

Multithreading: múltiples threads comparten los recursos del procesador.

- El procesador debe mantener el estado cada thread e.g., una copia de bloque de registros, un PC separado, tablas de páginas separadas.
- La memoria compartida ya soporta múltiples procesos.
- HW para conmutación de thread muy rápido. Mucho más rápido que entre procesos.

¿Cuándo conmutar?

- Cada ciclo conmuta de thread (grano fino).
- Cuando un thread debe parar (por ejemplo fallo de coche).

Multithreading de Grano Fino

Conmuta entre threads en cada instrucción, entrelazando la ejecución de los diferentes thread.
Generalmente en modo "round-robin", los threads bloqueados se saltan.
La CPU debe ser capaz de conmutar de thread cada ciclo.
Ventaja: Puede ocultar stalls de alta y baja latencia, cuando un thread esta bloqueado los otros usan los recursos.
Desventaja: retarda la ejecución de cada thread individual, ya que un thread sin stall es retrasado por reparto de recursos (ciclos) entre threads.
Ejemplos: Niagara y Niagara 2 (SUN).

Multithreading Grano Grueso

Conmuta entre threads solo en caso de largos stalls, como fallos de cache L2.
Ventajas:

- No necesita conmutación entre thread muy rápida.
- No retarda cada thread, la conmutación solo se produce cuando un thread no puede avanzar.

Desventajas: no elimina perdidas por stalls cortos. La conmutación es costosa en ciclos.

- Como CPU lanza instrucciones de un nuevo thread, el pipeline debe ser vaciado.
- El nuevo thread debe llenar el pipe antes de que las instrucciones empiecen a completarse.

Ejemplos: IBM AS/400, Montecito (Itanium2 9000), Sparc64 VI.

Simultaneos Multithreading (SMT)

Dentro de un procesador superescalar fuera de orden hay mecanismos Hw para soportar la ejecución la ejecución de más de un thread.

- Gran numero de registros físicos donde poder mapear los registros arquitectónicos de los diferentes threads.
- El renombrado de registros proporciona un identificador único para los operandos de una instrucción, por tanto instrucciones de diferentes thread se pueden mezclar sin confundir sus operandos.

Solo necesitamos sumar una tabla de renombrado por thread y PC separados. Commit independiente se soporta con un ROB por thread (lógico o físico).
Puede haber conflictos en la jerarquía de memoria.
Ejemplos: Pentium4, Power5 y 6, Nephalem (2008).

3.5. Explicación de ejercicios resueltos

Ejercicio 1. Sea un computador superescalar similar a la versión Tomasulo del MIPS capaz de lanzar a ejecución dos instrucciones sin dependencias por ciclo de reloj (incluyendo operaciones enteras). el siguiente código implementa la operación DAXPY ($X = a*X+Y$) sobre un vector de 100 elementos:

```
foo:   LD     F2, 0(R1)  ; leer X(i)
       MULTD  F4, F2, F0 ; a*X(i)
       LD     F6, 0(R2)  ; leer Y(i)
       ADDD   F6, F4, F6 ; a*X(i)+Y(i)
       SD     0(R2), F6  ; almacenar Y(i)
       ADDI   R1, R1, #8 ; incrementar el índice de X
       ADDI   R2, R2, #8 ; incrementar el índice de Y
       SGTI   R3, R1, done
       BEQZ   R3, foo
```

Las unidades funcionales están segmentadas y se asume que existen suficientes estaciones de reserva.

A. Desenrolla el bucle DAXPY para obtener 4 copias del cuerpo y planificarlo de tal forma que se alcance el mayor IPC posible.

B. ¿Cuántos ciclos emplea cada iteración del bucle original?

C. ¿Cuál es la ganancia producida por el desenrollado del bucle?

Solución eje ricio 1:
Apartado A:
Desarrollo de 4 iteraciones y renombramiento de registros.

```
foo  LD      F2 0(R1)
     MULTD   F4, F2, F0
     LD      F6 0(R2)
     ADDD    F6, F4, F6
     SD      0(R2) F6
     LD      F8 8(R1)        ; Renombrado: F(i+6)← Fi
     MULTD   F10, F8, F0
     LD      F12 8(R2)
     ADDD    F12, F10, F12
     SD      8(R2) F12
     LD      F14 16(R1)      ; Renombrado F(i+1)←Fi
     MULTD   F16, F14, F0
     LD      F18 16(R2)
     ADDD    F18, F16, F18
     SD      16(R2) F18
     LD      F20 24(R1)      ; Renombrado F(i+18)←Fi
     MULTD   F22, F20, F0
     LD      F24, F22, F24
     SD      24(R2) F24
     ADDI    R1, R1, #32
     ADDI    R2, R2, #32
     SGTI    R3, R1, done
     BEQZ    R3, foo
```

Reordenar para que sean independientes dos a dos:

```
foo:  LD      F2 O(R1)
      LD      F8 8(R1)
      MULTD   F4, F2,F0
      MULTD   F10, F8, F0
      LD      F6 0(R2)
      LD      F12 8(R2)
      ADDD    F6, F4, F6
      ADDD    F12, F10, F12
      SD      0(R2) F6
      SD      8(R2) F12
      LD      F14 16(R1)
      LD      F20 24(R1)
      MULTD   F16, F14, F18
      MULTD   F22, F20, F0
      LD      F18, 16(R2)
      LD      F24 2(R2)
      ADDD    F18, F16, F18
      ADDD    F24, F22, F24
      SD      16(R2) F18
      ADDI    R1, R1, #32
      SD      16(R2) F18
      SGTI    R3, R1, done
      ADDI    R2, R2, #32
      BEQZ    R3, foo
```

Hay predicción de saltos.
Hay recursos de planificación, por lo que no se para el pipe.

$$\text{CPI} = 0{,}5 \rightarrow \text{IPC} = 2$$

Solución apartado B:

			Issue	Exec	Mem	Write
foo:	LD	F2, 0(R1)	1	-		
	MULTD	F4, F2, F0	2	-		
	LD	F6, 0(R2)	2	-		
	ADDD	F6, F4, F6	3	-		
	SD	0(R2), F6	4	-		
	ADDI	R1, R1, #8	4			
	ADDI	R2, R2, #8	5	-		
	SGTI	R3, R1, done	5	-		
	BEQZ	R3, foo	6	-		

Suponemos que hay suficientes estaciones de reserva, y además, por lo que vemos en la tabla, se producen 6 ciclos por iteración (CPI=6/9).

Solución apartado C:

$$Ta = 25 \quad \text{iter} \times 24 \quad \text{instr/iter} \times 0{,}5 \quad \text{ciclos/instr} = 300 \quad \text{ciclos}$$

$$Tb = 100 \quad \text{iter} \times 9 \quad \text{instr/iter} \times 6/9 \quad \text{ciclos/instr} = 600 \quad \text{ciclos}$$

$$\text{Speedup} = \frac{Tb}{Ta} = 2$$

Se obtiene una mejora del rendimiento del 100 %.

Ejercicio 2. Sea un computador superescalar similar al del ejercicio anterior, pero además con ejecución especulativa. Las unidades funcionales segmentadas tienen las siguientes características:

Unidad funcional	Cantidad	Latencia
FP mult	1	7 ciclos
FP add	1	5 ciclos
ALU int	1	1 ciclos

Se supone que la latencia de memoria es 1 ciclo de reloj. Considerando la misma versión del código DAXPY que en el ejercicio anterior muestra el diagrama instrucción-tiempo para las dos primeras iteraciones del bucle original. Se supone que el salto es correctamente predicho y tomado en la primera iteración.

Solución ejercicio 2:

	Issue	Exec	Write	Comm
LD F2, 0(R1)	1	2	3	4
MULTD F4, F2, F0	2	4-10$	11	12
LD F6, 0(R2)	2	3	4	13
ADDD F6, F4, F6	3	12-16$	17	18
SD 0(R2), F6	4	5	17	19
ADDI R1, R1, #8	4		6	20
ADDI R2, R2, #8	5	6	7	21
SGTI R3, R1, done	5	7$	8	22
BEQZ R3, foo	6	9$	10	23
LD F2, 0(R1)	7	8	9	24
MULTD F4, F2, F0	8	10-16$	18+	25
LD F6, 0(R2)	8	9	12+	26
ADDD F6, F4, F6	9	19-23$	24	27
SD 0(R2), F6	10	11	24	28
ADDI R1, R1, #8	10		13+	29
ADDI R2, R2, #8	11	12	14+	30
SGTI R3, R1, done	11	14$	15	31
BEQZ R3, foo	12	16$	19+	32

El $ indica dependencia LDE y + indica el uso de CDB.

Ejercicio 3. Sea un procesador segmentado con planificación dinámica mediante el algoritmo de Tomasulo.

- Los datos que se escriben en la etapa de escritura se pueden usar en la etapa de ejecución de una instrucción en el mismo ciclo.
- Las instrucciones SGTI, la BNEZ y NOP tienen tratamiento de instrucciones enteras.
- Los LOAD y STORE tienen una latencia de dos ciclos, utilizan su propia unidad funcional no segmentada.
- Hay un solo bus de datos común (CDB).
- La estructura del procesador tiene las siguientes características:

UF	CANTIDAD	LATENCIA	SEGMENTADA
FP ADD	1	2	SI
FP DIV	1	8	SI
FP MUL	1	4	SI
INT ALU	1	1	SI
MEMORIA	1	2	NO

ESTACIONES RESERVA	CANTIDAD
FP ADD	2
FP DIV	2
FP MUL	2
INT ALU	3
LOAD	2
STORE	2

Dado el siguiente fragmento de programa:

```
       ADDI R1,R0,#DIR
       LD   F0,0(R7)
LOOP:  LD   F4,0(R1)
       DIVD F8,F4,F0
       SUBI R1,R1, #4
       LD   F2,0(R1)
       MULD F8,F2,F8
       SD   0(R1),F8
       SUBI R3,R3,,#8
       SGTI R5,R3,#1000
       BEQZ R5,LOOP
       NOP
```

A. Representa el diagrama de instrucción-tiempo para todo el fragmento de programa considerando sólo la primera iteración, indicando en cada caso el tipo de parada que se produce.

B. Indica el diagrama instrucción-tiempo para todo el fragmento de programa considerando sólo la primera iteración, suponiendo un superescalar con las mismas unidades funcionales ya vistas, que lanza un par de instrucciones de cualquier tipo en un ciclo de reloj. En caso de dependencia de datos entre dos instrucciones de un par se congela el lanzamiento de la segunda. Se supone que existen dos buses comunes de datos, que el banco de registros puede realizar dos escrituras en cada ciclo de reloj, y que tiene suficientes estaciones de reserva para que no se produzcan paradas.

Solución ejercicio 3:
Apartado A:
Las ER disponibles nunca se agotan en la primera iteración.

	ISSUE	EJECUCIÓN	WRITE
ADDI R1, R0, #DIR	1	2	3
LD F0, 0(R7)	2	3-4	5
LD F4, 0(R1)	3	5-6 ESTRUC	7
DIVD F8, F4, F0	4	7-14 LDE	15
SUBI R1, R1, #4	5	6-7 BCD	8
LD F2, 0(R1)	6	8-9 LDE	10
MULD F8, F2, F8	7	15-18 LDE	19
SD 0(R1), F8	8	19-20 LDE	
SUBI R3, R3, #8	9	10	11
SGTI R5, R3, #1000	10	11	12
BEQZ R5, LOOP	11	12	
NOP	12		

Solución apartado B:
No hay límites para las ER, se supone que existen suficientes.

	ISSUE	EJECUCIÓN	WRITE
ADDI R1, R0, #DIR	**1**	**2**	**3**
LD F0, 0(R7)	**2**	**2-3**	**4**
LD F4, 0(R1)	2 LDE	4-5 ESTRUC	6
DIVD F8, F4, F0	**3**	**6-13 LDE**	**14**
SUBI R1, R1, #4	**3**	**4**	**5**
LD F2, 0(R1)	4 LDE	6-7 LDE	8
MULD F8, F2, F8	5 LDE	14-17 LDE	18
SD 0(R1), F8	**6**	**18-19 LDE**	
SUBI R3, R3, #8	**6**	**7**	**8**
SGTI R5, R3, #1000	10	11	12
BEQZ R5, LOOP	**8**	**9**	
NOP	**8**		

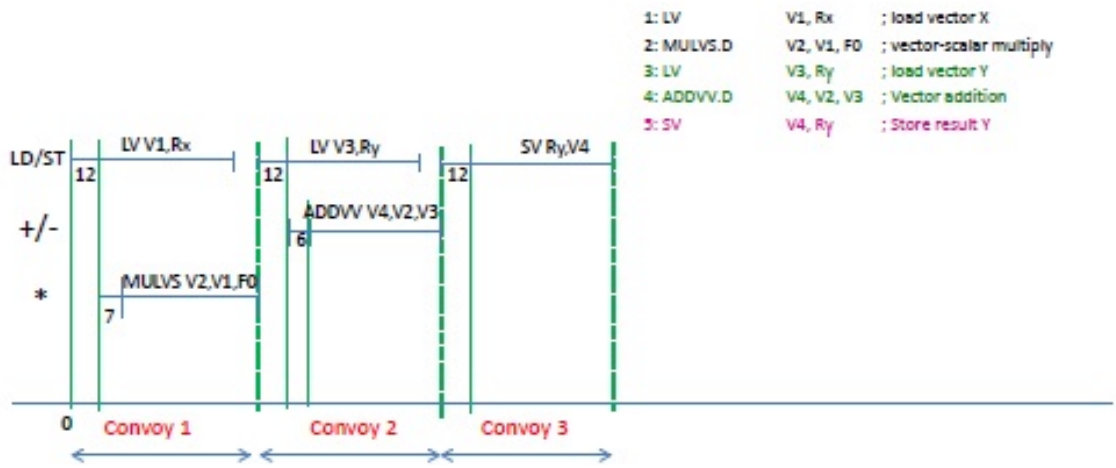
Ejercicio 4. Considera la ejecución del código vectorial para la operación DAXPY explicado en clase en el VMIPS.

A. Determina el tiempo de cálculo para vectores de 230 componentes, teniendo en cuenta la penalización de la ejecución por bloques (strip-minining).

B. Deduce, para esta operación el valor R_∞ (en MFLOPS) y de $N_{\frac{1}{2}}$. Asume un tiempo de ciclos de 1.5 ns.

Solución ejercicio 4:
Solución apartado A:

DAXPY con 23 elementos con VMIPS. $Y = a * X + Y$.



$$T_n = [n/\text{MVL}] \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} =$$

$$= [230/64] \times (15 + 12 + 7 + 12 + 6 + 12) + 230 \times 3 = 4 \times 64 + 690 = 946 \quad \text{ciclos}$$

Solución apartado B:

En el problema nos dan los MFLOPS y el $N_{\frac{1}{2}}$, y el tiempo de ciclo.

$$T_n = \left[\frac{n}{\text{MVL}}\right] \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} = \left[\frac{n}{64}\right] \times (15 + 49) + n \times 3 = 4n$$

$$R_\infty = \lim_{n\to\infty} \frac{\text{nº Ops en PF}}{T_n} = \lim_{n\to\infty} \frac{2n}{4n} = \frac{1}{2}\frac{\text{FLOP}}{\text{ciclo}} =$$

$$= 0{,}5/(1{,}5 \times 10^{-9} s) = 333{,}33 \quad \text{MFLOPS}$$

$$N_{\frac{1}{2}} = R_\infty/2 = 0{,}25 \quad \text{FLOP/ciclo}$$

$$T_n = \left[\frac{n}{64}\right] \times (15 + 49) + n \times 3$$

Suponemos que la mitad del rendimiento asintótico se obtiene en la primera iteración.

$$T_n = 1 \times (15 + 49) + n \times 3$$

$$R = \frac{2n}{T_n}$$

$$0{,}25 \quad \text{FLOP/ciclo} = 2n/64 + 3n \to n = 12 \cdot 8 = 13 \to N_{\frac{1}{2}} = 13$$

Ejercicio 5. Para estudiar el efecto que produce una adición al VMIPS de dos nuevos pipes de carga/almacenamiento, repite el problema anterior considerando que existen tres pipes de carga/almacenamiento.

Solución ejercicio 5:
Solución apartado A:
DAXPY con 230 elementos con 3 pipes de LD/ST. $Y = a * X + Y$.

$$T_n = [n/\text{MVL}] \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} = [230/64] \times (15+12+7+6+12) + 230 \times 1 =$$

$$= 4 \times 52 + 230 = 438 \quad \text{ciclos}$$

Solución apartado B:
Lo mismo que en el ejercicio anterior.

$$T_n = \left[\frac{n}{\text{MVL}}\right] \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} = \left[\frac{n}{64}\right] \times (15 + 37) + n \times 1 = 1{,}81n$$

$$R_\infty = \lim_{n\to\infty} \frac{\text{N}^\text{o}\text{Ops en PF}}{T_n} = \lim_{n\to\infty} \frac{2n}{1{,}81n} = 1{,}10 \frac{\text{FLOP}}{\text{ciclo}}$$

$$1{,}10 \quad \text{FLOP/ciclo} = 1{,}10/(1{,}5 \times 10^{-9} s) = 733 \quad \text{MFLOPS}$$

$$N_{\frac{1}{2}} = R_\infty/2 = 0{,}55 \quad \text{FLOP/ciclo}$$

$$T_n = \left[\frac{n}{64}\right] \times (15 + 37) + n \times 1$$

Suponemos que la mitad del rendimiento asintótico se obtiene en la primera iteración.

$$T_n = 1 \times (15 + 37) + n \times 1$$

$$0{,}55 \quad \text{FLOP/ciclo} = 2n/52 + n \rightarrow n = 19 \cdot 72 = 20 \rightarrow N_{\frac{1}{2}} = 20$$

Ejercicio 6. Se desea realizar el cálculo:

$$A(I) = [B(I) * C(I)] + [D(I) * S], \text{para} \quad I = 1..N$$

donde A, B, C y D son vectores de N componentes en memoria y S es un escalar almacenado en un registro, en una máquina similar al VMIPS, pero con registros vectoriales de longitud p.

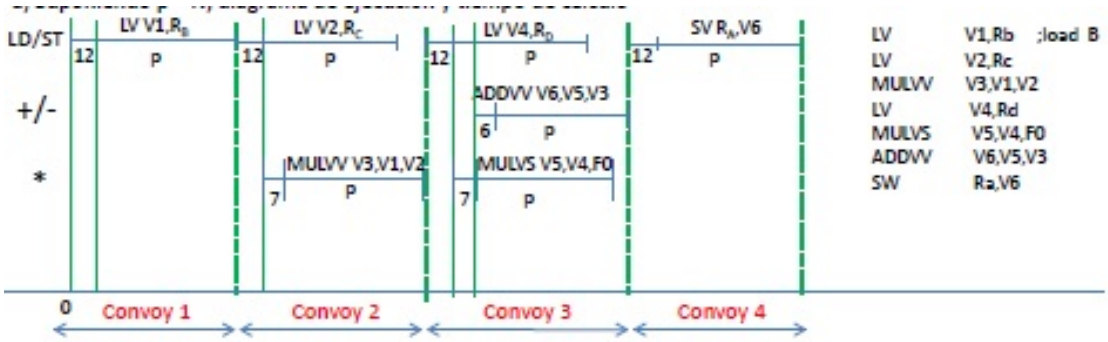
A. Suponiendo que p es igual a N, dibujar el diagrama que muestra la realización del anterior cálculo en el menor tiempo posible. Señala los instantes de inicio y final de cada operación y halla el tiempo total del cálculo.

B. Utiliza iterativamente la secuencia de operaciones realizada en el apartado A para hallar el tiempo de cálculo en el caso de que $N = 265$ y $p = 32$. Ten en cuenta la penalización producida por el procesamiento por bloques (strip-minig).

C. ¿Cuál es el rendimiento en MFLOPS para el cálculo efectuado en el apartado B, si el ciclo de reloj es 2ns?

D. ¿Para qué valor de N se alcanza el rendimiento mitad del hallado en el apartado C?

Solución ejercicio 6:
Apartado A:



Tiempo ejecución $= 12 + p + 12 + 7 + p + 12 + 7 + 6 + p + 12 + p = 68 + 4p$ ciclos

Solución apartado B:
Sabemos que $N = 265$ y $p = 32$.

$$T_n = [n/\text{MVL}] \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} = [265/32] \times (15 + 68) + 264 \times 4 =$$

$$= 9 \times 83 + 1060 = 1807 \quad \text{ciclos}$$

Solución apartado C:

$$R = \lim_{n\to\infty} \frac{\text{Nº Ops en PF}}{T} = \lim_{n\to\infty} \frac{3 \times 265 \quad \text{FLOP}}{1807 \quad \text{ciclos}} = 0{,}44 \quad \text{FLOP/ciclo}$$

$$0{,}44 \quad \text{FLOP/ciclo} = 0{,}44/(2 \times 10^{-9} s) = 220 \quad \text{MFLOPS}$$

Solución apartado D:

$$R/2 = 0{,}22 \quad \text{FLOP/ciclo}$$

$$T_n = \left[\frac{n}{32}\right] \times (15 + 68) + n \times 4$$

Suponemos que la mitad del rendimiento del apartado C se obtiene en la primera iteración.

$$T_n = 1 \times (15 + 68) + n \times 4$$

$$0{,}22 \quad \text{FLOP/ciclo} = 3n/83 + 4n \rightarrow n = 8 \cdot 61 = 9 \rightarrow n = 9$$

Ejercicio 7. Dado el fragmento de programa

```
for (i=0; i<50; i++){
    for (j=0; j<50; j++)
        c[i,j]=(a[i,j]*b[i,j])+5;

}
```

donde a, b y c son matrices 50x50, almacenadas en memoria por filas.

A. Tradúcelo al correspondiente fragmento de código máquina simbólico del VMIPS sabiendo que los cálculos en cada iteración del bucle interno se implementan mediante instrucciones vectoriales, y los del bucle externo mediante iteraciones sucesivas.

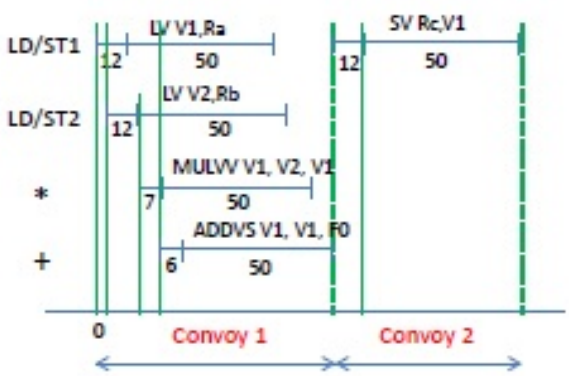
B. Suponiendo que el VMIPS tuviera dos pipes de carga/almacenamiento, determina el tiempo de ejecución del programa obtenido en el apartado anterior, teniendo en cuenta la penalización de la ejecución por bloques.

C. Halla el rendimiento del anterior logaritmo si el nº de filas crece indefinidamente y el nº de columnas se mantiene constante.

Solución ejercicio 7:
Solución apartado A:

```
        LD F0, Cte5
        ADDI R1, RO, #50      ; longitud vectorial
        MTC1 VLR, R1          ; VLR 50
Loop    LV V1 V1, Ra          ; cargar fila del vector a
        LV V2, Rb             ; cargar fila del vector b
        MULVV V1,V2, V1       ; a[i,XXX]*b[i,XXX]
        ADDVS V1, V1,FO       ; (a[i,XXX]*b[i,XXX])+5
        SV Rc, V1             ; almacenar fila de c
        ADDI Ra, Ra, #50*8
        ADDI Rb, Rb, #50*8
        SUBI R1, R1, #1
        BNEZ R1, Loop         ; si no ha finalizado el programa (50 iteraciones), volver a Loop
```

Solución apartado B:



Tenemos que adaptar la fórmula T_n porque las iteraciones se hacen 50, por lo que en lugar de [n/MVL] usamos la Cte50. Además la n se calcula de la siguiente forma: n = nº de elementos a procesar = $50 \times 50 = 2500$, $T_{start} = 1 + 12 + 7 + 6 + 12 + 12 = 38$y $T_{chime} = 2$. Con esto ya podemos calcular T_n.

$$T_n = 50 \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} = 50 \times (15 + 38) + 2500 \times 2 = 50 \times 53 + 5000 =$$

$$= 7650 \quad \text{ciclos}$$

Solución apartado C:
El nº de iteraciones es n/50 porque el número de filas crece indefinidamente.

$$T = (n/50) \times (15 + 38) + 2 \times n = 3{,}06 \times n$$

$$R = \frac{2 \times n}{3{,}06 \times n} = 0{,}654 \quad \text{FLOP/ciclo}$$

Ejercicio 8. Dado el bucle

```
for (i=1;i<=265;i++) {
    a[i]=b[i]+c[i];
    if(a[i]==b[i])
            d[i]=a[i]*3;
    b[i]=a[i]-5;
}
```

se pide:

A. Tradúcelo al correspondiente bloque de código vectorial del VMIPS.

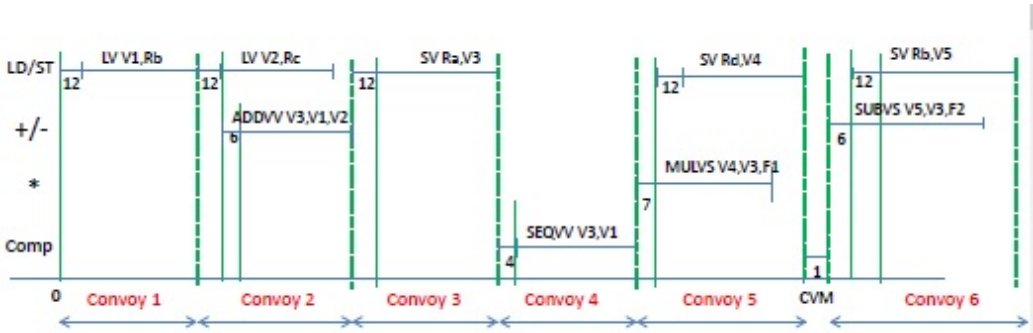
B. Determina el tiempo de ejecución, teniendo en cuenta la penalización de la ejecución por bloques. Se supone que las operaciones de comparación se ejecutan en un pipe de cuatro etapas que no puede encadenarse.

C. Si la mitad de las componentes de los vectores a y b son iguales, determina el rendimiento.

Solución ejercicio 8:
Solución apartado A:

```
        LD F1 Cte3          ; Cargamos en F1 el escalar 3
        LD F2 Cte5          ; Cargamos en F2 el escalar 5
        ADDI R2, R0, #2120  ; Cargamos en R2 la longitud total del vector en bytes (265x8=2120)
        ADDI R2, R2, Rb     ; Cargamos en R2 la dirección del final el vector B
        ADDI R1, R0, #9     ; Cargamos en R1 el número de elementos de la primera iteración (265/64=9)
        MTC1 VLR, R1        ; Cargamos en VLR el valor 9
        ADDI R1, R0, #72    ; Cargamos en R1 el número de bytes de la primera iteración (9x8=72)
        ADDI R3, R0, #64    ; Cargamos en R3 el número de elementos del resto de las iteraciones
Loop    LV V1, Rb
        LV V2, Rc
        ADDVV V3, V1, V2
        SV Ra, V3
        SEQVV V3, V1        ; Si a(i)=b(i) se pone la componente i del vector de máscara a 1
        MULVS V4, V3, F1
        SV Rd, V4
        CVM                 ; Ponemos 1s en todas las componentes del vector de máscara
        SUBVS V5, V3, F2
        SV Rb, V5
        ADDD Ra, Rb, R1     ; Cargamos en Ra la dirección del primer elemento de a de la siguiente iteración
        ADDD Rb, Rb, R1     ; Cargamos en Rb la dirección del primer elemento de b de la siguiente iteración
        ADDD Rc, Rc, R1     ; Cargamos en Rc la dirección del primer elemento de c de la siguiente iteración
        ADDD Rd, Rd, R1     ; Cargamos en Rd la dirección del primer elemento de d de la siguiente iteración
        ADD R1, R0, #512    ; Cargamos en R1 el número de bytes de la siguiente iteración
        MITC1 VLR, R3       ; Cargamos en VLR el valor 64
        SUB R4, R2, Rb      ; Hemos llegado al final del vector B
        BNEZ R4, Loop       ; Si no hemos llegado al final, volver a Loop
```

Solución apartado B:



$$T_n = [n/\text{MVL}] \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} =$$

$$= [265/64] \times (15 + 12 + 12 + 6 + 12 + 4 + 7 + 12 + 1 + 6 + 12) + 265 \times 6 =$$

$$= 5 \times 99 + 1590 = 2085 \quad \text{ciclos}$$

Solución apartado C:
La suma se ejecuta 265 veces.
La multiplicación se ejecuta 265/2 veces.
Total: 663 ops en PF en 2085 ciclos → 0318 FLOP/ciclo.

Ejercicio 9. Supongamos un VMIPS con dos pipes de carga/almacenamiento y otra máquina similar, VMIPS2, que presenta las siguientes diferencias respecto de la primera:

- Mientras que el VMIPS trabaja con un tiempo de ciclo de 1.25 ns, el VMIPS2 lo hace con un tiempo de ciclo de 1 ns.

- Para lograr la mejora del tiempo de ciclo, los pipes del VMIPS2 están segmentados en más etapas. El pipe suma tiene 8 etapas y el pipe de multiplicación tiene 10 etapas.
- El VMIPS2 tiene 2 pipes de suma y dos de multiplicación. Debido a la complejidad adicional del control de tales pipes. T_{loop} vale 45 ciclos, en lugar de los 15 del VMIPS.

Consideremos la ejecución del siguiente bucle:

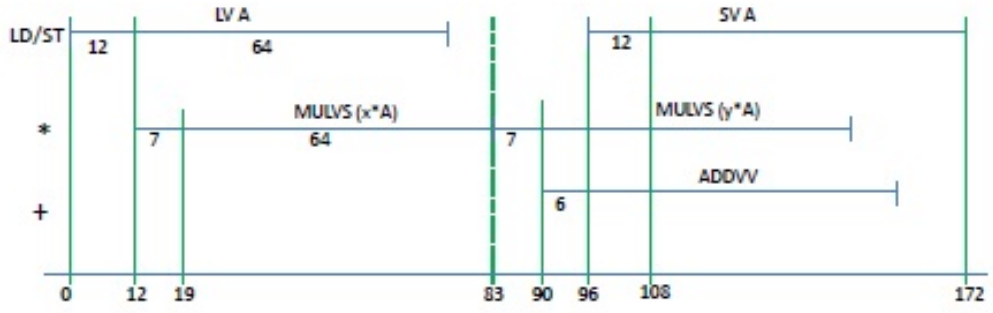
```
for (i=0;i<n;i++)
    A[i]=x*A[i]+y*A[i];
```

Se pide:

A. Halla la razón entre los rendimientos asintóticos (en MFLOPS) del VMIPS y el VMIPS2.

B. Estudia si existe algún valor de n tal que por debajo de él es más rápida una de las máquinas, mientras que por encima de él es más rápida la otra.

Solución ejercicio 9:
Solución apartado A:
Sabemos que el tc del VMIP es de 1.25 ns y los pipes del LD/ST son 2, y por otro lado, sabemos que el tc del VMIPS2 es de 1 ns, sus pipes LD/ST son 2 y el T_{loop} es de 45 ciclos.
VMIPS:



$$T_{\text{chime}} = 2; T_{\text{start}} = 172 - (2 \times 64))44; T_{\text{loop}} = 15$$

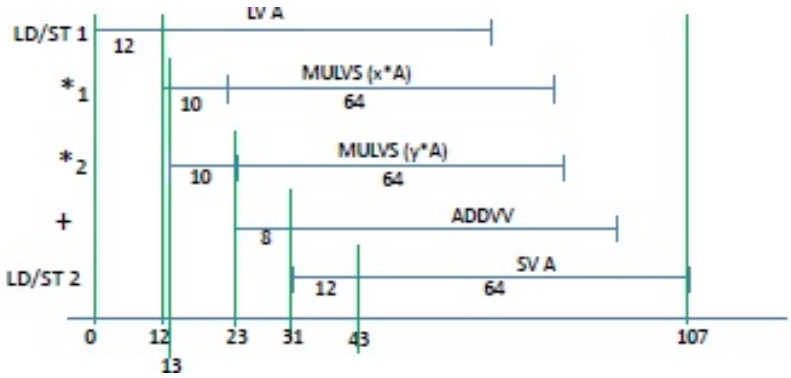
$$n \to \infty \to T_n = (n/64)(15 + 44) + 2n = 2{,}92n \quad \text{ciclos}$$

$$R_\infty = (\text{n}^\text{o} \text{ op PF})/(2{,}92n) \quad \text{FLOP/ciclo} = 3/2{,}92 \quad \text{FLOP/ciclo}$$

Lo pasamos a MFLOPS.

$$\frac{3/2{,}92}{1{,}25 \times 10^{-3}} = 822 \quad \text{MFLOPS}$$

VMIPS2:



$$T_{\text{chime}} = 1; T_{\text{start}} = 107 - 64 = 43; T_{\text{loop}} = 45$$

$$n \to \infty \to T_n = (n/64)(45 + 43) + 1n = 2{,}37n \quad \text{ciclos}$$

$$R_\infty = (\text{n}^\text{o} \text{ op PF})/(2{,}37n) \quad \text{FLOP/ciclo} = 3/2{,}37 \quad \text{FLOP/ciclo}$$

Lo pasamos a MFLOPS:

$$\frac{3/2{,}37}{1 \times 10^{-9}}\text{FLOP/s} = \frac{3/2{,}37}{1 \times 10^{-3}}\text{MFLOPS} = 1266 \quad \text{MFLOPS}$$

$$\frac{R_\infty(\text{VMIPS})}{R_\infty(\text{VMIPS2})} = \frac{822}{1266} = 0{,}6495$$

El rendimiento asintótico del VMIPS es un 64.95 % del VMIPS2.

Solución apartado B:

Suponemos que $n < 64$:

VMIPS:

$$T_n = 1(15+44)+2n = 59+2n \quad \text{ciclos} \to T_n = (59+2n) \times 1{,}25ns = (73{,}75+2{,}5n)ns$$

VMIPS2:

$$T_n = 1(45 + 43) + n = 88 + n \quad \text{ciclos} \to T_n = (88 + n) \times 1ns = (88 + n)ns$$

Para vectores pequeños, será más rápido el VMIPS. Vamos a igualarlos para saber cuál es el más rápido.

$$75 + 2n = 88 + n \to n = 9{,}5$$

A partir de los 10 elementos el VMIPS2 es más rápido.

Ejercicio 10. Supongamos dos matrices, A[29,50] y B[50,29], que están almacenadas por filas en la memoria de un computador vectorial con todas las características del VMIPS, pero que posee dos pipes del carga/almacenamiento y dos pipes de suma. Se desea realizar la operación:

```
for (i=1;i<29;i++) {
    for (j=0;j<50;j++)
        C[i,j] = A[i-1,j] + B[j,i-1] * 5;
}
```

Se pide:

A. Traduce el anterior programa al lenguaje máquina simbólico del VMIPS de tal manera que se trate de optimizar el tiempo de ejecución.

B. Dibuja el diagrama de tiempo correspondiente a una ejecución de bucle interno. Determina el tiempo de cálculo y el rendimiento en MFLOPS del programa anterior.

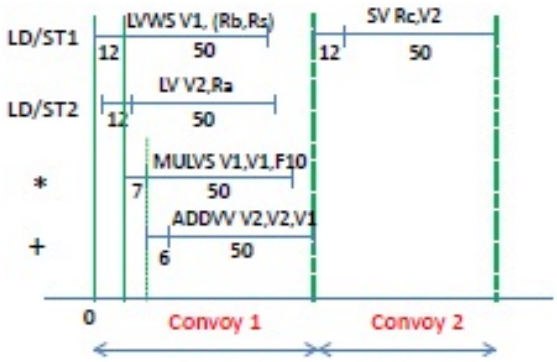
C. ¿Qué porcentaje del rendimiento asintótico se obtiene en la operación?

NOTAS: Se supone que las direcciones iniciales de las matrices A y B están almacenadas en los registros Ra, Rb y Rc, respectivamente. En el enunciado se asume que la notación X[M,N] representa una matriz X de M filas y N columnas. La memoria está entrelazada con 16 bancos. El tiempo de ciclo es 2ns.

Solución ejercicio 10:
Solución apartado A:

```
        ADDI R1, R0, #50      ; Longitud vectorial
        ADDI R2, R0, #28      ; nº de iteraciones
        LD F10, Cte5          ; Load 5 en F10
        ADDI Rc, Rc, #50*8    ; Preparar puntero inicial a Rc
        MTC1 VLR, R1          ; VLR ← 50
        ADDI Rs, R0, #29*8    ; Stride para accesos a B (29*8 bytes)
Loop    LVWS V1, (Rb, Rs)     ; V1← Columna de B
        LV V2, Ra             ; V2← Fila de A
        MULVS V1, V1, F10     ; B*5
        ADDVV V2, V2, V1      ; A+B*5
        SV Rc, V2             ; Store C
        ADDI Ra, Ra, #50*8    ; Incrementar punteros
        ADDI Rb, Rb, #8
        ADDI Rc, Rc, #50*8
        SUBI R2, R2, #1
        BNEZ R2, Loop         ; Si no se ha acabado (28 iteraciones), volver a Loop
```

Solución apartado B:
El stride en accesos a b es de 29 palabras, 29 y 16 son primos entre sí: Luego el acceso sin conflictos ya que nº bancos de memoria(16)> tiempo acceso a memoria (12 ciclos).



Se hacen 28 iteraciones del bucle vertical.

$$T_n = (\text{n}^\text{o} \text{ iteraciones}) \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} =$$

$$= 28 \times (15 + 12 + 7 + 6 + 12) + (28 \times 50) \times 2 = 28 \times 52 + 2800 = 4526 \quad \text{ciclos}$$

$$R = 2 \times 50 \times 28 \quad \text{FLOP}/4526 \quad \text{ciclos} = 0{,}658 \quad \text{FLOP/ciclo}$$

$$t_n = 2ns = 2{\times}10^{-9}s \rightarrow R = (0{,}658 \quad \text{FLOP/ciclo})/(2{\times}10^{9}s/\text{ciclo}) = 329 \quad \text{MFLOPS}$$

Solución apartado C:
Crecen indefinidamente, por lo que el nº de iteraciones siempre será n/50.

$$T_n = (\text{n}^\text{o} \text{ iteraciones}) \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} =$$

$$= (n/50) \times (15 + 12 + 7 + 6 + 12) + 2 \times n = (52 \times n/50) + 2 \times n = 3{,}04 \times n \quad \text{ciclos}$$

$$R = 2 \times n/3{,}04 \times n = 0{,}658 \quad \text{FLOP/ciclo}$$

Se obtiene el mismo rendimiento, ya que, la longitud total de los vectores siempre es múltiplo de 50.

Ejercicio 11. Consideramos el vector A, de 37 componentes, y la matriz B, de 37 filas por 29 columnas. Ambas estructuras están almacenadas en la memoria de un VMIPS, a partir de las direcciones apuntadas por los registros Ra y Rb, respectivamente. La matriz B está almacenada por filas. Supongamos que se desea realizar el cálculo:

```
for (i=0;i<37;i++) {
    B[i,2] = A[i] + (B[i,4]*9) + 2.5;
    if ( A[i]< B[i,2])
        C[i] = B[i,2];
}
```

A. Escribe un programa en el lenguaje máquina simbólico del VMIPS que ejecute el cálculo en el menor tiempo posible. Se supone que las operaciones de comparación de vectores se ejecutan en un pipe de cuatro etapas.

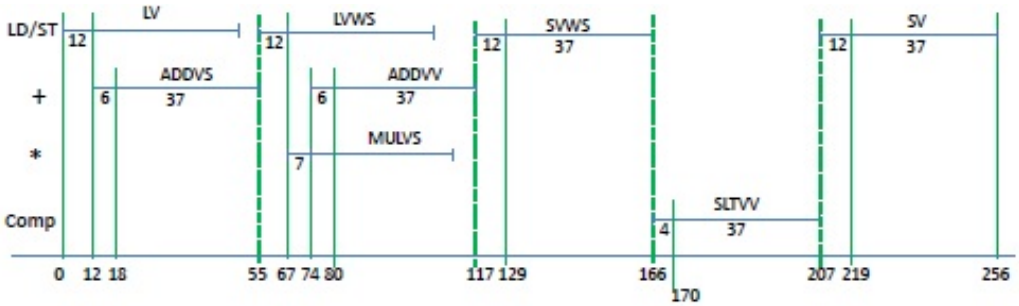
B. Dibuja el diagrama temporal, determina el tiempo de cálculo del programa y calcula el porcentaje del rendimiento asintótico que se alcanza.

C. Repite los pasos anteriores, pero considerando que en lugar de un sólo pipe de carga almacenamiento existieran dos, así como dos pipes de suma y dos de multiplicación.

Solución ejercicio 11:
Solución apartado A:
A[37] B[37,29] Ra= dir ini de A; Rb= dir ini de B, Sup: Rc dir ini de C, B almacenada en filas.

```
LD      F2, Cte9
LD      F1, Cte2_5
ADDI    R1, R0, #29*8    ; Stride
ADDI    Rb4, Rb, #4*8    ; Dir de B[0,4]
ADDI    Rb2, Rb, #2*8    ; Dir de B[0,2]
ADDI    R2, R0, #37
MTC1    VLR, R2          ; Cargar en VLR el valor 37 (long de los vectores)
LV      V1, Ra           ; Load A
ADDVS   V2, V1, F2       ; A+2.5
LVWS    V3, (Rb4, R1)    ; Load B[-,4]
MULVS   V3, V3, F2       ; B[-,4]*9
ADDVV   V3, V3, V2       ; (A+2.5)+(B[-,4]*9)
SVWS    (Rb2, R1), V3    ; Store B[-,2]
SLTVV   V1, V3           ; IF: Preparar vector de máscara
SV      Rc, V3           ; Store C
```

Solución apartado B:



$$T_{\text{chime}} = 5; T_{\text{start}} = 256 - (5 \times 37) = 71 \quad \text{ciclos}$$

$$n \to \infty \to T_n = (n/64)(15 + 71) + 5n = 6{,}34n \quad \text{ciclos}$$

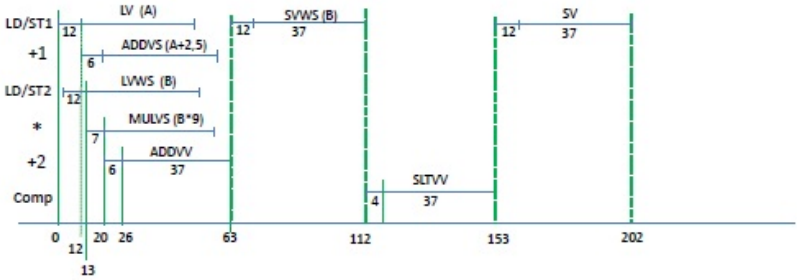
$$R_\infty = (\text{N}^\text{o} \text{ Op PF})/(6{,}34n) \quad \text{FLOP/ciclo} = 3/6{,}34 = 0{,}473 \quad \text{FLOP/ciclo}$$

$$R = (\text{N}^\text{o} \text{ Op PF})/(\text{T de cálculo}) = (3 \times 37 \quad \text{FLOP})/(256 \quad \text{ciclos}) =$$

$$= 0{,}434 \quad \text{FLOP/ciclo}$$

$$R/R_\infty = 0{,}434/0{,}473 = 0{,}9175$$

Se alcanza el 91.75 % del rendimiento asintótico.
Solución apartado C:



Tiempo de cálculo=202 ciclos.

$$T_{\text{chime}} = 4; T_{\text{start}} = 202 - (4 \times 37) = 202 - 148 = 54 \quad \text{ciclos}$$

$$n \to \infty \to T_n = (n/64)(15 + 54) + 4n = 5{,}08n \quad \text{ciclos}$$

$$R_\infty = (\text{N}^\text{o} \text{ Op PF})/(5{,}08n) \quad \text{FLOP/ciclo} = 3/5{,}08 = 0{,}591 \quad \text{FLOP/ciclo}$$

$$R = (\text{N}^\text{o} \text{ Op PF})/(\text{T de cálculo}) = (3 \times 37 \quad \text{FLOP})/(202 \quad \text{ciclos}) =$$

$$= 0{,}550 \quad \text{FLOP/ciclo}$$

$$R/R_\infty = 0{,}550/0{,}591 = 0{,}9306$$

Se alcanza el 93.06 % del rendimiento asintótico.

Capítulo 4

Tema 4: Paralelismo a nivel de datos: arquitectura vectorial, instrucciones SIMD para multimedia, GPUs

En este capítulo vamos a ver los siguientes conceptos.

- Introducción.
- Arquitectura vectorial.
  - Repertorio de instrucciones vectoriales.
  - Tiempo de ejecución, medidas de rendimiento.
  - Procesamiento selectivo de elementos.
  - Ejemplos.
- Instrucciones SIMD para procesamiento multimedia.
- Unidades para procesamiento gráfico (GPUs).
  - Modelo de programación a alto nivel: CUDA.
  - Instrucciones PTX.
  - Arquitectura del elemento de proceso.
- Paralelismo a nivel bucle: vectorización.
  - Detección de dependencias.
- Explicación de ejercicios resueltos.

4.1. Introducción

SIMD

Single Instruction steam, Multiple Data stream. Flujo de instrucción simple y flujo de datos múltiple. Esto significa que una única instrucción es aplicada sobre diferentes datos al mismo tiempo. En las máquinas de este tipo, varias unidades de procesado diferentes son invocadas por una única unidad de control. Al igual que las MISD, las SIMD soportan procesamiento vectorial (matricial) asignando cada elemento del vector a una unidad funcional diferente para procesamiento concurrente.

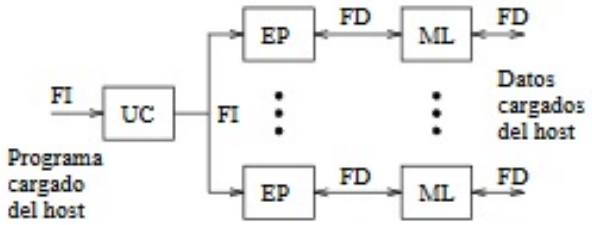


Figura 4.1: Procesador matricial

Arquitectura SIMD:

- Aplicaciones de ciencia/ingeniería con abundante cálculo matricial.
- Nuevas aplicaciones: gráficos, visión artificial, comprensión de voz, Deep Learning...

La eficiencia energética de SIMD es ventajosa frente a MIMD:

- Sólo es preciso hacer un "fetch" para operar sobre varios datos.
- Ahorro de energía atractivo en dispositivos portátiles.

Se explora el paralelismo de la SIMD:

- Arquitectura vectorial.
- Extensiones SIMD (extensiones multimedia).
- Graphics Processor Units (GPUs).

4.2. Arquitectura vectorial

IMPORTANTE

Permiten que los vectores sean tratados como números en punto flotante (FP), como enteros, o como datos lógicos. La unidad escalar no es básicamente diferente del tipo de CPU segmentada.

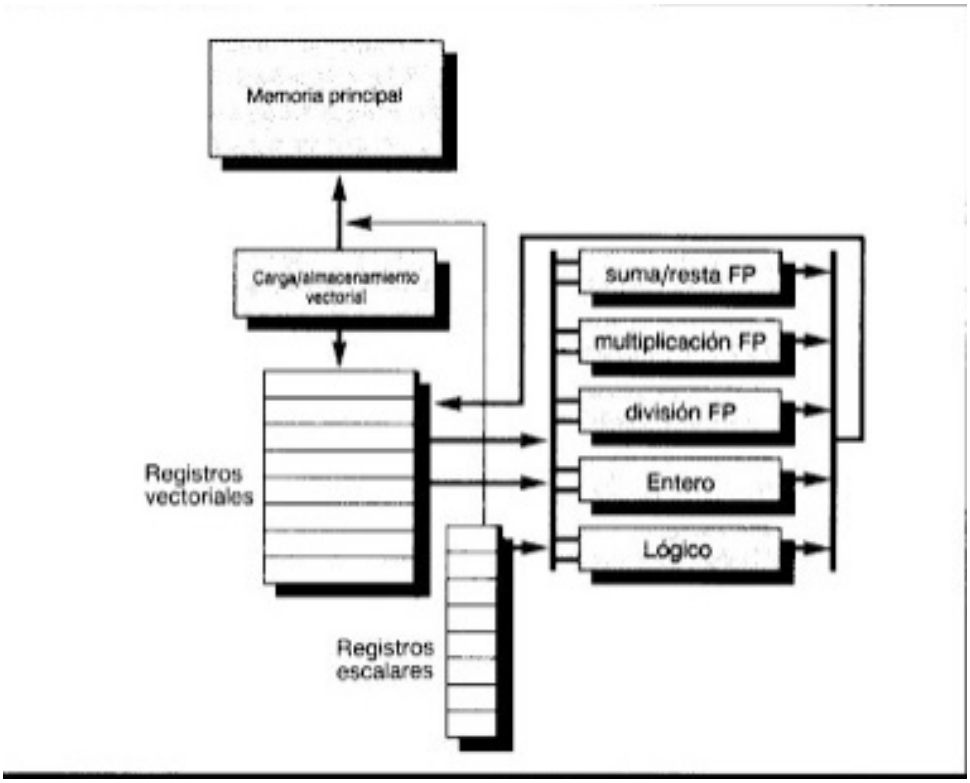


Figura 4.2: Estructura básica de una arquitectura con registros vectoriales

Vector de carga: Funcionalmente equivale a un pipe, dato por ciclo, después de latencia inicial (12 ciclos).
Registros vectoriales: 1 reg=64 elementos, 1 elemento=64 bits. 16 read, 8 write ports.
UFs segmentadas: Add 6 etapas, Mul 7 etapas y Div 20 etapas.
Funciones:

- Leer conjunto de datos sobre "registros vectoriales".
- Operar sobre el contenido de estos registros.
- Almacenar los resultados finales en memoria.

4.2.1. Operaciones vectoriales

Todas las operaciones vectoriales están segmentadas.
Las cargas y almacenamientos en memoria también están segmentadas, con un tiempo de inicio de 12 ciclos. Pueden realizarse operaciones en paralelo.
Grupo: Conjunto de instrucciones vectoriales que pueden ejecutarse en paralelo. No pueden tener ningún tipo de riesgo entre ellas.
Tiempo de grupo: Tiempo aproximado de ejecución de una instrucción vectorial. Es independiente del número de elementos del vector.
Características de las operaciones vectoriales:

- Secuencias de cálculos independientes.
- Una instrucción → Muchas operaciones.
- Patrón de acceso a memoria conocido.

- Explotación eficiente de memoria entrelazada.
- Disminución de instrucciones de salto. Reducción conflictos de control.

Ventajas de los procesadores vectoriales:

- El cálculo de los elementos de un vector suele ser independiente. Menos riesgo de datos.
- El número de instrucciones es bajo.
- Los accesos a memoria siguen un patrón fijo. Los elementos de los vectores están ordenados.
- Se eliminan bucles y dependencia de control.

4.2.2. Repertorio VMIPS

Operaciones vectoriales aritméticas sobre registros.
Instrucciones especiales de carga/almacenamiento de vectores (LV, SV).
Modos de direccionamiento especiales para vectores no contiguos. Ejemplos:

- LVWS (Load Vector With Stride): Carga elementos equiespaciados a una cierta distancia.
- LVI (Load Vector using Index): El contenido de un registro vectorial indica las posiciones de un elemento a cargar.

Registros especiales de longitud vectorial (VLR): Indica la longitud de los vectores a procesar.
Máscara (VM): Registro de 64 bits. Para las posiciones con VM a cero, la operación no se ejecuta. Ejecución selectiva de operaciones sobre componentes.

4.2.3. Código escalar vs Código vectorial

$Y = a * X + Y$(AXPY, Sup: vectores de 64 elementos)
Versión escalar:

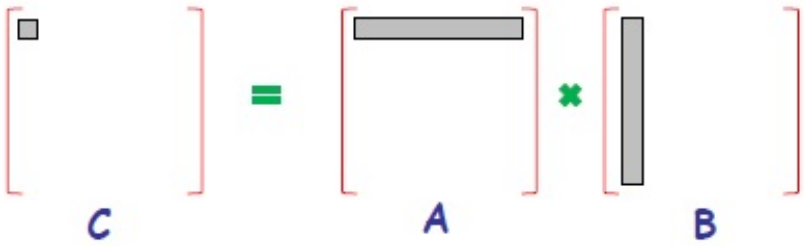
```
        L.D      F0, a            ; Load escalar a
Loop:   DADDIU   R4, Rx, #512     ; Last adress to load
        L.D      F2, 0(Rx)        ; Load X[i]
        MUL.D    F2, F2, F0       ; Ax X[i]
        L.D      F4, 0(Ry)        ; Load Y[i]
        ADD.D    F4, F4, F2       ; AxX[i]+Y[i]
        S.D      F4, 0(Ry)        ; Store Y[i]
        DADDIU   Rx, Rx, #8       ; Incremento indice X
        DADDIU   Ry, Ry, #8       ; Incremento indice Y
        DSUBU    R20, R4, Rx      ; Loop exhausted?
        BNEZ     R20, Loop
```

Versión vectorial:

```
L.D        F0, a        ; Load scalar a
LV         V1, Rx       ; Load vector X
MULVS.D    V2, V1, F0   ; Vector-scalar multiply
LV         V3, Ry       ; Load vector Y
ADDVV.D    V4, V2, V3   ; Vector addition
SV         V4, Ry       ; Store result Y
```

Las instrucciones ejecutadas pasan de ser 580 a 6.

Producto de matrices



Si realizamos un productor escalar multiplicaremos fila de A por columna de B.
Para una instrucción tipo MULVV.D haremos un producto de 2 vectores.
Por otro lado, si sumamos todos los elementos del vector estaremos realizando una ejecución secuencial (escalar).
Aplicación de AXPY: Se calculan todos los elementos de una fila C a la vez.
Operaciones por el cálculo de la primera fila de C:

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + ... + a_{1n}b_{n1}$$

Se realiza en todas las filas y columnas.
Método Paso 0: Inicializar $(c_{11}, c_{12}, c_{13}, ..., c_{1n})$ a cero.

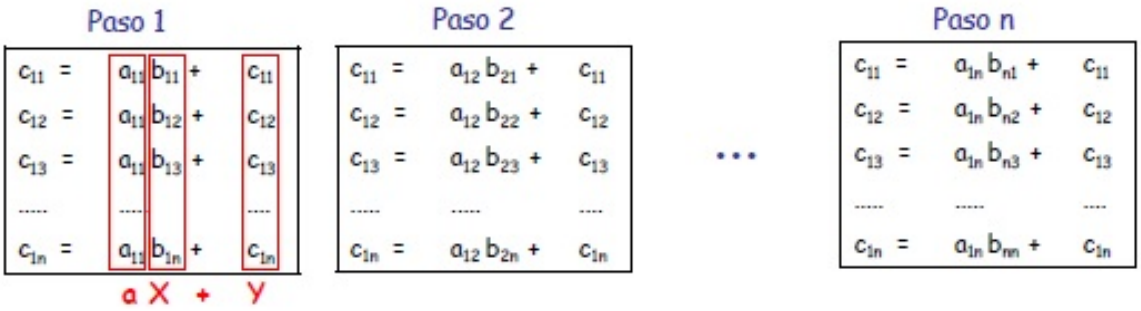


Figura 4.3: Método de paso 0

4.2.4. Tiempo de ejecución

Depende de tres factores:

- Longitud de los vectores operandos.

- Riesgos estructurales: las UF necesarias están ocupadas, no hay puertos del BR disponibles.
- Dependencias de datos.

Velocidad del procesamiento:

- Las FUs del VMIPS consumen (y producen) un elemento por ciclo de reloj.
- El tiempo de ejecución de una operación vectorial es aproximadamente igual a la longitud del vector.

Convoy: Conjunto de una o varias instrucciones vectoriales que potencialmente pueden ejecutarse juntos. Pueden tener riesgos LDE.
Paso:

- Unidad de tiempo para ejecutar un convoy.
- Para vectores de longitud requiere ciclos de reloj.

Ejemplo

```
LV        V1, Rx       ; Load vector X
MULVS.D   V2,V1, F0    ; Vector-scalar multiply
LV        V3, Ry       ; Load vector Y
ADDVV.D   V4, V2, V3   ; Vector addition
SV        V4, Ry       ; Store result Y
```

Conflictos:

- 1 y 2 no tienen conflictos estructurales.
- 3 tiene conflicto estructural con 1.
- 4 no tiene conflictos estructurales con 3.
- 5 tiene conflicto estructural con 3.

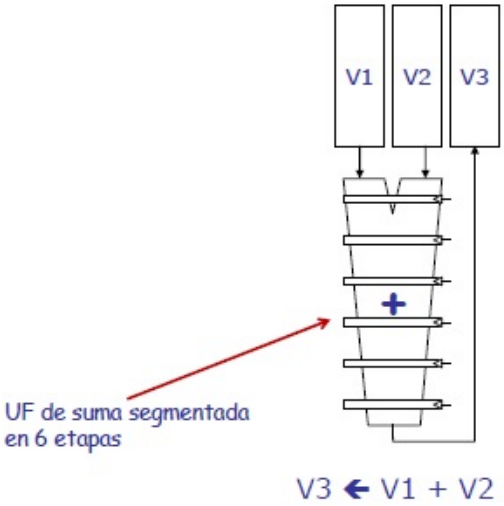
Convoyes resultantes:

- 1: Formado por LV y MULVS.D.
- 2: Formado por LV y ADDVV.D.
- 3: Formado por SV.

Tiempo de cálculo: $3 \times 64 = 192$ ciclos (Tchime=3).

4.2.5. Ejecución de operaciones aritméticas

Alta latencia: No demasiado relevante debido a la falta de dependencia entre los cálculos sobre un vector.



Ejemplo

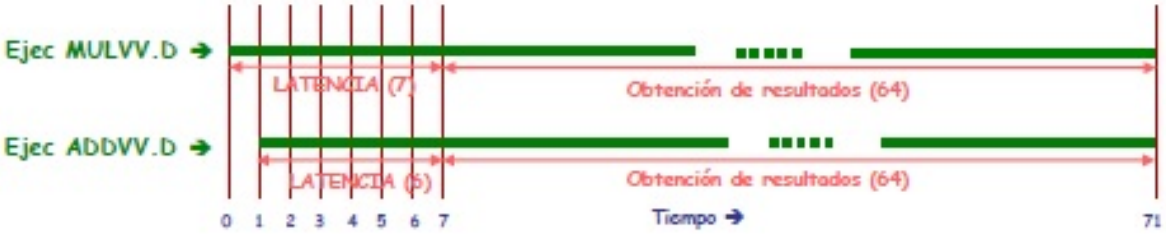
Operaciones independientes:

MULVV.D	V1, V2, V3
ADDVV.D	V4, V5, V6

Comportamiento temporal:

Operación	Inicio	Fin
MULVV.D	0	7+64=71
ADDVV.D	1	1+6+64=71

En ausencia de conflictos, **lanza una instrucción por ciclo**.
Representación:



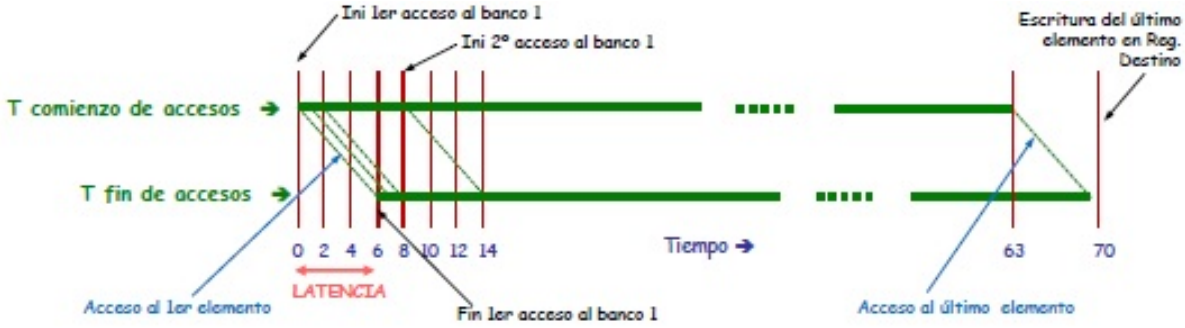
Memoria entrelazada:
Sup 8 bancos, T acceso a memoria: 6 ciclos.

- Carga de un vector de 64 componentes que comienza en la dirección 136 (cada componente 8 bytes).

- Formato de dirección: xxxx yyy 000 (siendo yyy= nºde banco).

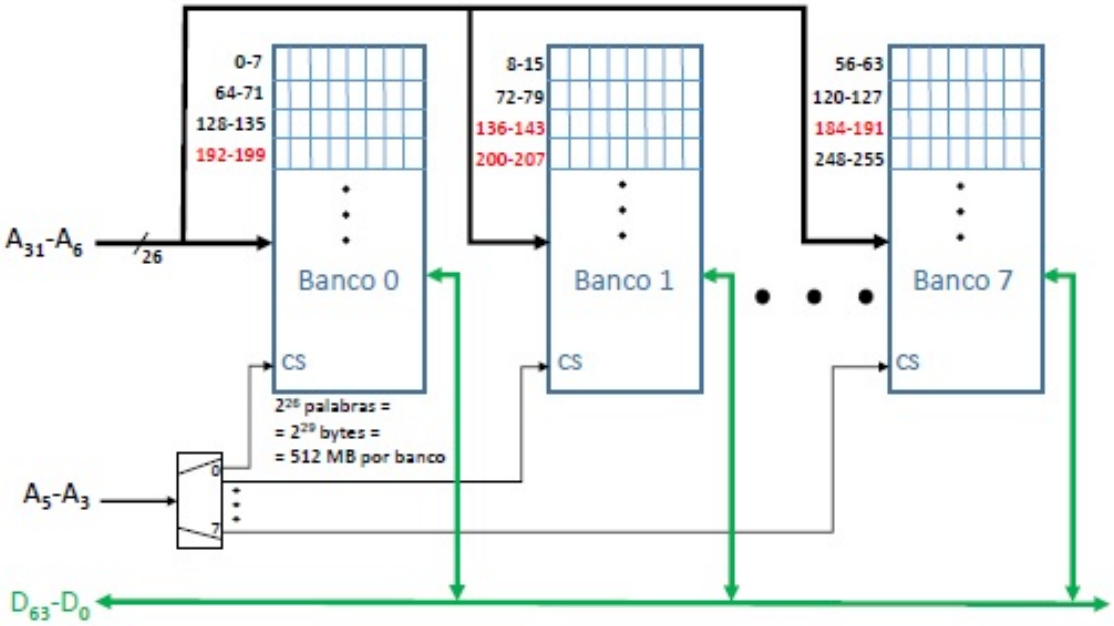
Dir	**136**	144	152	160	168	176	184	192	**200**	208	...
Banco	**1**	2	3	4	5	6	7	0	**1**	2	...
Tini	**0**	1	2	3	4	5	6	7	**8**	9	...
Tfini	**6**	7	8	9	10	11	12	13	**14**	15	...

Diagrama temporal:



Memoria entrelazada de orden bajo:

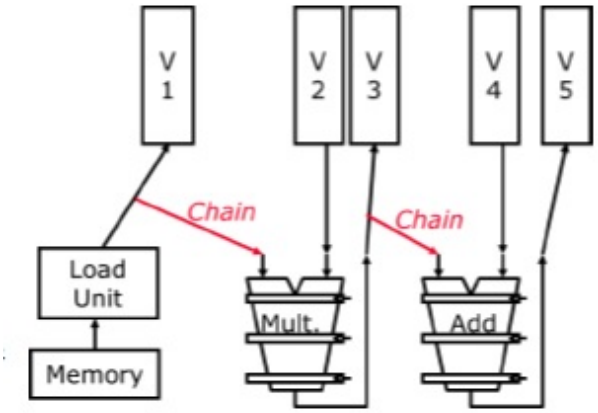
- Longitud de palabra: 8 bytes
- Palabras consecutivas están en bancos consecutivos.
- Cada 8 palabras se vuelve a usar el mismo banco.



4.2.6. Encadenamiento

Tratamiento de dependencias RAW.
Problema:

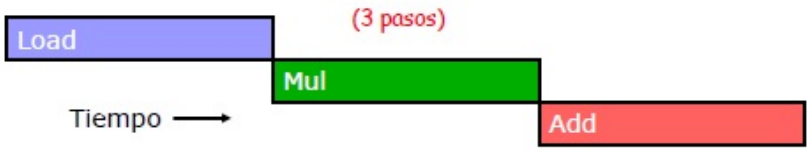
LV	V1
MULVV.D	V3, V1, V2
ADDVV.D	V5, V3, V4



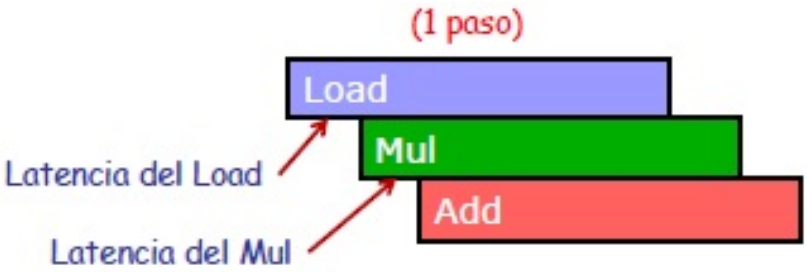
Solución:

- Cray-1. Encadenamiento de operaciones.
- 3 operaciones en 1 paso.
- Encadenamiento flexible.

Sin encadenamiento: esperar hasta que se haya calculado el último elemento de la operación anterior.



Con encadenamiento: Una instrucción puede comenzar cuando está disponible el primer elemento de la operación de la que depende.



4.2.7. Vectores de longitud arbitraria

Registro de longitud vectorial (VLR): El valor cargado en VLR determina el nº de componentes sobre los que actúa la op. vectorial lanzada.
Vectores cortos:

- Cargar registros VLR.
- Ejecutar operación vectorial.

Vectores largos. Proceso por bloques (strip mining).

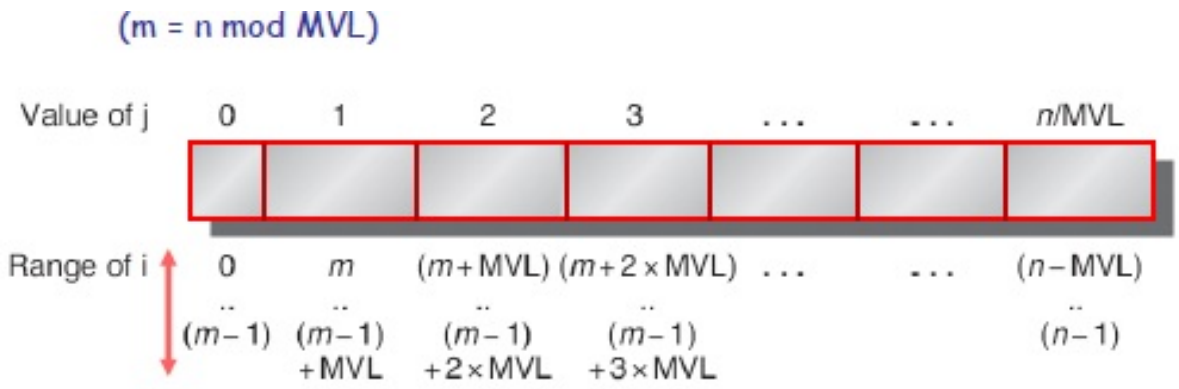
- Descomponer operación en varias suboperaciones.
- n/MVL operaciones vectoriales de longitud MVL.
- 1 operación vectorial de longitud.

Strip mining

Ejemplo: Bucles anidados para ejecución de DAXPY.

```
low = 0;
VL = (n % MVL);                    /*find odd-size piece using modulo */
for (j = 0; j <= (n/MVL); j=j+1) {  /*outer loop*/
  for (i = low; i < (low+VL); i=i+1)  /*runs for length VL*/
        Y[i] = a * X[i] + Y[i] ;     /*main operation*/
  low = low + VL;                  /*start of next vector*/
  VL = MVL;                        /*reset the length to MVL*/
}
```

Diagrama de ejecución de DAXPY:



Modelo de rendimiento para operaciones por bloques:
Inicialización de operaciones (T_{base}): Cálculo de dir iniciales, op escalares de preparación del bucle.
Penalización por inicialización y control del bucle (1 vez por cada iteración):

- T_{start}: nº ciclos para el llenado de pipes. Depende de las operaciones vectoriales incluidas en el bucle. Si las instr son independientes, el llenado de pipes se solapa.
- T_{loop}: actualización de punteros.

T_{chime}: nº convoyes en el bucle.
Tiempo total de cálculo para vectores de n elementos:

$$T_n = \left\lceil \frac{n}{\text{MVL}} \right\rceil \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}}$$

Rendimiento:

$$R(\text{FLOP/ciclo}) = \frac{\text{Nº Op en PF}}{\text{Tiempo cálculo (ciclos)}} = \frac{\text{Nº Op en PF}}{T_n}$$

Ejemplo: $A = B \times s$, para vectores de 200 componentes.

```
        DADDUI    R2,R0,#1600   ;total # bytes in vector
        DADDU     R2,R2,Ra      ;address of the end of A vector
        DADDUI    R1,R0,#8      ;loads length of 1st segment
        MTC1      VLR,R1        ;load vector length in VLR
        DADDUI    R1,R0,#64     ;length in bytes of 1st segment (8 elements)
        DADDUI    R3,R0,#64     ;vector length of other segments (64 elements)
Loop:   LV        V1,Rb         ;load B
        MULVS.D   V2,V1,Fs      ;vector * scalar
        SV        Ra,V2         ;store A (structural hazard)
        DADDU     Ra,Ra,R1      ;address of next segment of A
        DADDU     Rb,Rb,R1      ;address of next segment of B
        DADDUI    R1,R0,#512    ;load byte offset next segment
        MTC1      VLR,R3        ;set length to 64 elements
        DSUBU     R4,R2,Ra      ;at the end of A?
        BNEZ      R4,Loop       ;if not, go back
```

$$T_n = \left\lceil \frac{n}{\text{MVL}} \right\rceil \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} = \left\lceil \frac{200}{64} \right\rceil \times (15 + (12 + 7 + 12)) + 200 \times 2 =$$
$$= 584 \quad \text{ciclos}$$

Rendimiento asintótico ($\mathbf{R_\infty}$): Rendimiento obtenido para supuestos vectores de longitud infinita.

$$R_\infty = \lim_{n\to\infty} R = \lim_{n\to\infty} \frac{2n}{T_n}$$

Longitud del rendimiento mitad del asintótico ($N_{\frac{1}{2}}$): Longitud de los vectores operandos para la que se obtiene la mitad del rendimiento asintótico.

$$N_{\frac{1}{2}} = \frac{R}{2}$$

Operaciones condicionales: registro de máscara

Ejecutar

```
for (i = 0; i < 64; i=i+1)
    if (X[i] != 0)
        X[i] = X[i] – Y[i];
```

El registro de máscara vectorial permite omitir las operaciones sobre los elementos que no cumplen la condición.

```
LV        V1, Rx        ; Load vector X en V1
LV        V2, Ry        ; Load vector Y en V2
L.D       F0, #0        ; Load FP zero en F0
SNEVS.D   V1, F0        ; Sets VM(i) a 1 si V(i)!=F(0)
SUBVV.D   V1, V1, V2    ; Substract under vectr mask
SV        Rx, V1        ; Store the result in X
```

El rendimiento se reduce:

- Se consume el mismo tiempo.
- Se ejecutan menos operaciones útiles.

Bancos de memoria

El sistema de memoria debe soportar un elevado ancho de banda en la carga y almacenamiento de vectores.
Dispersar los accesos entre múltiples bancos:

- Control independiente de las direcciones en cada banco.
- Load y store de palabras no secuenciales.
- Soporte para múltiples procesadores vectoriales puedan compartir la misma memoria.

Vectores no consecutivos en memoria
Producto de matrices:

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
            A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

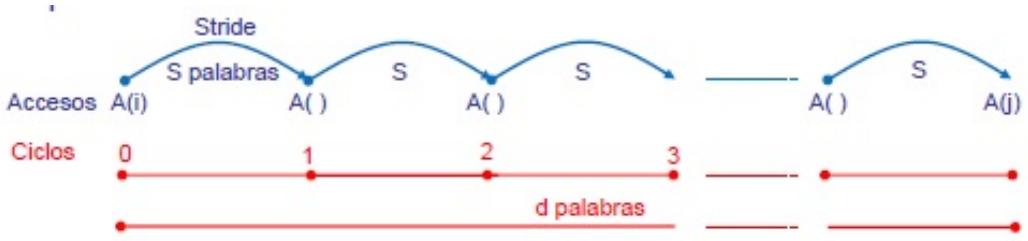
Las matrices están ordenadas por filas y para vectorizar el producto de filas de B por columnas de D:

- Acceso a elementos consecutivos de B.
- Acceso a elementos de D separados por 100 palabras de distancia.

Instrucciones de load/store con "stride" (espaciamento):

- Problema: Repetición de acceso al mismo banco de memoria antes del fin de la op anterior.
- Acceso libre de conflicto si: mcm (espaciamento, nº bancos)/espaciamento $\geq$ T acceso a banco.

Acceso a array A con espaciamento S palabras:



El objetivo es un acceso por ciclo sin conflictos.
Conflicto entre los accesos a A(i) y A(j):

- A(i) y A(j) están en el mismo banco.
- Cuando comienza A(j) todavía no ha acabado el acceso a A(i).

Vectores dispersos

Los elementos no equidistan.
Carga: Registro vectorial ← elemento dispersos de un array: LVI V1, (R1+V2). V2 indica las direcciones de los elementos a cargar en V1 como desplazamiento respecto de una dir inicial indicada por R1.
Almacenamiento: Llevar contenido de un registro vectorial a posiciones dispersas de un array: SV1 (R1+V2), V1.
Ejemplo: Código fuente; sólo se accede a algunos elementos de A y C.

```
for (i = 0; i < n; i=i+1)
        A[K[i]] = A[K[i]] + C[M[i]];
```

Código máquina: Vk y Vm usados como índices.

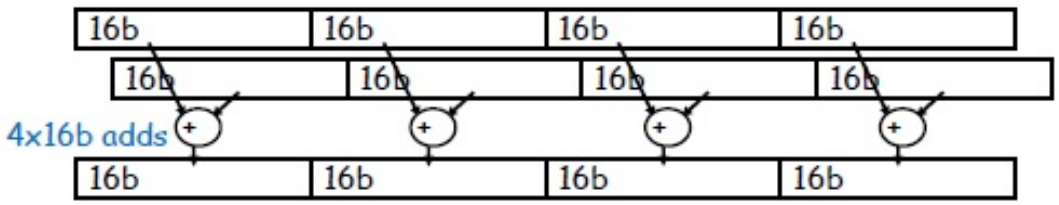
```
LV        Vk, Rk        ; Load K
LVI       Va, (Ra+Vk)   ; Load A[K[]]
LV        Vm, Rm        ; Load M
LVI       Vc, (Rc+Vm)   ; Load C[M[]]
ADDVV.D   Va, Va, Vc    ; Añadirlos
SVI       (Ra+Vk), Va   ; Store A[K[]]
```

4.3. Instrucciones SIMD para procesamiento multimedia

4.3.1. Extensiones SIMD

También son conocidas como "extensiones multimedia".
Las aplicaciones multimedia suelen operar sobre datos de menor anchura que las UFs y los registros disponibles.
Realizar varias operaciones a la vez:

- Desconectar la cadena de propagación de carries.
- Una sola instrucción de suma opera sobre varios elementos almacenados en un registro → equivale a una op vectorial, aunque con un vector de pocos elementos.



Limitaciones:

- La longitud de los vectores se codifica en el Cod_op.
- No existe direccionamiento sofisticado.
- No hay reg de máscara.

Ejemplo: código con extensiones SIMD en MIPS

Sup: Añadimos instrucciones multimedia SIMD de 256 bits al MIPS.
Código para DAXPY:

```
        L.D F0, a                ; Load scalar a
        MOV F1, F0               ; Copy a into F1 for SIMD MUL
        MOV F2, F0               ; Copy a into F2 for SIMD MUL
        MOV F3, F0               ; Copy a into F3 for SIMD MUL
        DADDIU R4, Rx, #512      ; Last adress to load 64 elements
Loop    L.4D F4, 0(Rx)           ; Load X[i], X[i+1], X[i+2], X[i+3]
        MULD.4D F4, F4, F0       ; a*X[i], a*X[i+1], a*X[i+2], a*X[i+3]
        L.4D F8, 0(Ry)           ; Load Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D F8, F8, F4        ; a*X[i]+Y[i], ..., a*X[i+3]+Y[i+3]
        S.4D F8, 0(Ry)           ; Store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU Rx, Rx, #32       ; Increment index to X
        DADDIU Ry, Ry, #32       ; Increment index to Y
        DSUBU R20, R4, Rx        ; Compute bound
        BNEZ R20, Loop           ; Check if done
```

4.4. Unidades para Procesamiento Gráfico (GPUs)

Definición

Es un coprecesador que se dedica al procesamiento de gráficos y puede aligerar la carga de información que debe ser procesada por la unidad central.

Las GPUs son económicas, accesibles y contienen una gran cantidad de elementos de cómputo.
Idea básica:

- Modelo de ejecución heterogéneo (CPU+GPU).
- Desarrollar un lenguaje de programación tipo C que permita programar la GPU.
- Unificar todo el paralelismo de la GPU bajo la abstracción denominada "CUDA Thread".
- Modelo de Programación: "Single Instruction Multiple Thread (SIMD)".

4.4.1. Arquitectura de NVIDIA GPU

Importante

Multiprocesador compuesto por un conjunto de procesadores SIMD MT.

NVIDIA vs procesadores vectoriales

Similitudes:

- Buen funcionamiento con paralelismo de datos.
- Transferencias con memoria tipo dispersar/reunir (scatter/gather).
- Registros de máscara.
- Existencias de grandes ficheros de registros.

Diferencias:

- No hay un procesador escalar.
- Utilización de multithreading para ocultar la latencia de memoria.
- Existencia de gran cantidad de UFs.

CUDA (Compute Unified Device Architecture)

Definición

CUDA es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema.

CUDA produce código C/C++ para host y dialecto de C y C++ para GPU.

- Crea un thread separado para cada elemento de los vectores.
- Los threads se agrupan en bloques de threads:
  - El número de threads por bloque puede definirlo el programador.
  - Cada bloque es ejecutado por un procesador SIMD MT de la GPU.
  - Varios bloques pueden ejecutarse en paralelo sobre varios procesadores.
- El conjunto de bloques que implementan un cálculo vectorial sobre la GPU se denomina Grid (malla).

Ejecución del cálculo:

- Nombre_función:«**dimGrid, dimBlock**».
  - **dimGrid**: nº de bloques en Grid.
  - **dimBlock**: nº de threads por bloque.

- Los bloques y las mallas pueden tener hasta 3 dimensiones.

```
// Invocar DAXPY con 256 threads por Bloque (dimBlock)
__host__                          /* código para la CPU */
int nblocks = (n+ 255) / 256;   /*  cálculo del nº total de bloques en el Grid (dimGrid) */
daxpy <<<nblocks, 256>>> (n, 2.0, x, y);

// DAXPY en CUDA (representa el cálculo ejecutado para un elemento)
__device__                        /* código para los procesadores de la GPU */
void daxpy(int n, double a, double *x, double *y)
{
// ¿Qué thread soy? Calcular i = nº elemento del vector a procesar (= nº de thread), siendo
// nº elemento = (nº de bloque x tamaño de bloque) + (nº de thread dentro del bloque)
int i = blockIdx.x*blockDim.x + threadIdx.x;

// Si el nº elemento obtenido es mayor que el tamaño del vector, ignorar operación
if (i < n) y[i] = a*x[i] + y[i];
}
```

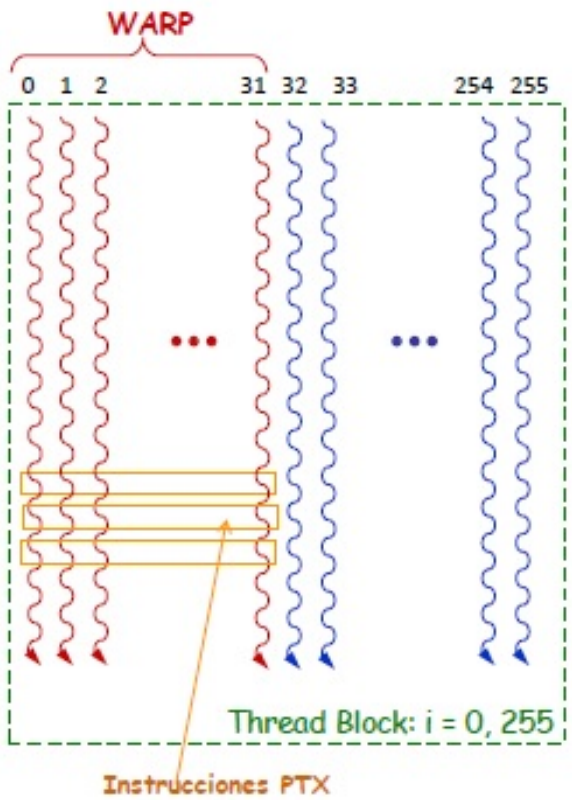
CPU

GPU

Figura 4.4: Código DAXPY en CUDA

4.4.2. Instrucciones PTX

Cada Thread Block se ejecuta en un procesador SIMD MT de la GPU.
Varios procesadores SIMD MT de la GPU pueden procesar diferentes Thread Blocks en paralelo.
Instrucción PTX: Ejecuta un mismo cálculo sobre varios datos.
WARP: Secuencia (thread) de instr PTX. El procesador ejecuta los WARP de un Thread Block en modo MT.



Ejemplo: secuencia de instrucciones PTX para una iteración del bucle DAXPY

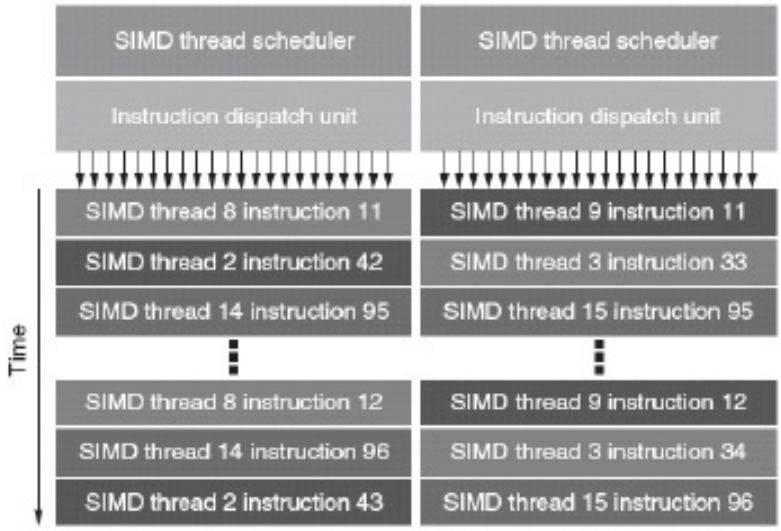
Usa reg virtuales: Ri(32 bits), RDi(64 bits).
Asigna reg físicos en el momento de la carga del programa.

```
shl.u32        R8, blockldx, 8      ; Thread Block ID*Block size
add.u32        R8, R8, threadldx    ; R8=i=my CUDA Thread ID
shl.u32        R8, R8, 3            ; Byte offset
ld.global.f64  RD0, [X+R8]          ; RD0=X[i]
ld.global.f64  RD2, [Y+R8]          ; RD2=Y[i]
mul.f64        RD0, RD0, RD4        ; Product in RD0=RD0*RD4
add.f64        RD0, RD0, RD2        ; Sum in RD0=RD0+RD2
st.global.f64  [Y+R8], RD0          ; Y[i]=sum(Y[i]*a+Y[i])
```

Cada instrucción PTX procesa 32 elementos.

4.4.3. Thread Scheduler

Cada procesador: 2 thread schedulers en paralelo.
Selecciona una instrucción de cada thread y las envía a dos conjuntos de 16 vías físicas.
Cada instrucción SIMD procesa 32 elementos → 2 ciclos por instrucción.



4.4.4. Saltos condicionales en GPUs

Gestión de registros de máscara (predicado) similar a los procesadores vectoriales.

- Para componentes enmascarados, el resultado no se guarda en el registro destino.
- Permite implementar instrucciones IF...THEN mediante una instrucción PTX.
- La parte del ELSE se complementa en el registro de máscara.
- Impacto en rendimiento.

Instrucciones PTX de salto: Permiten implementar construcciones condicionales mediante saltos verdaderos en el código.

Ejemplo

```
if (X[i] != 0)
        X[i] = X[i] – Y[i];
else X[i] = Z[i];
```

Código PTX:

```
                                        ; R8 actualizado de acuerdo con Thread Id
        Id.global.f64       RD0, [X+R8]     ; RD0=X[i]
        setp.neq.s32        P1, RD0, #0     ; P1 is predicate register 1
        @!P1 bra            ELSE1, *Push    ; Push old mask, set new mask bits
                                            ; if P1 false, go to ELSE1
        Id.global.f64       RD2, [Y+R8]     ; RD2=Y[i]
        sub.f64             RD0, RD0, RD2   ; Difference in RD0
        st.global.f64       [X+R8], RD0     ; X[i]=RD0
        @P1 bra             ENDIF, *Comp    ; Complement mask bits
                                            ; If P1 true, go to ENDIF1
ELSE1:  Id.global.f64       RD0, [Z+R8]     ; RD0=Z[I]
        st.global.f64       [X+R8], RD0     ; X[i]=RD0
ENDIF1: <next instruction>, *Pop            ; Pop to restore old mask
```

4.5. Paralelismo a nivel de bucle: vectorización

Las dependencias RAW entre setencias de una misma iteración no impiden la vectorización eficiente.
Dependencia directa: obliga a ejecutar en el orden dado.
¿Qué ocurre si los datos de una iteración son dependientes de los resultados generados en iteraciones previas?

- Dependencias en el espacio de iteraciones.
- Puede impedir la vectorización.
- Pueden existir reordenación de setencias válidas.

4.5.1. Ejemplos

```
for (i=999; i>=0; i=i-1){
        x[i] = x[i] + s;        /* S1 */
        z[i] = z[i] + x[i];     /* S2 */
}
```

Figura 4.5: Ejemplo 1

Dependencia directa: no impide la vectorización. Orden 1º S1, luego S2.

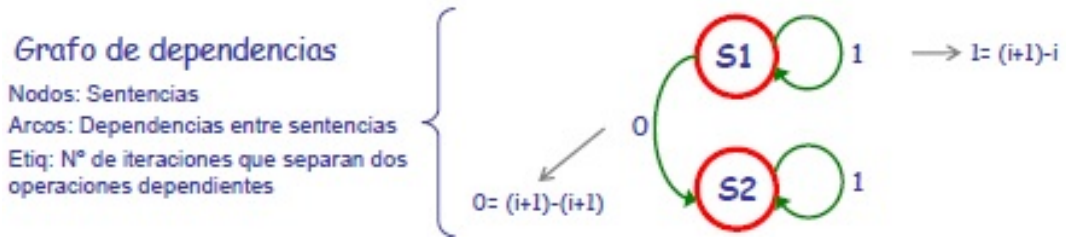
$$x[0:999] = x[0:999] + s$$
$$z[0:999] = z[0:999] + x[0:999]$$

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i];       /* S1 */
        B[i+1] = B[i] + A[i+1];     /* S2 */
}
```

Figura 4.6: Ejemplo 2

S1 y S2 usan valores calculados por ellas mismas en la iteración previa → ejecución en serie.
S2 usa resultados de S1 en la misma iteración.

- Si fuera la única dependencia, el bucle sería vectorizable.
- El orden de ejecución de S1 y S2 debe mantenerse.



4.5.2. Antidependencias y dependencias de salida

Las depedencias de nombre pueden evitarse renombrando variables.

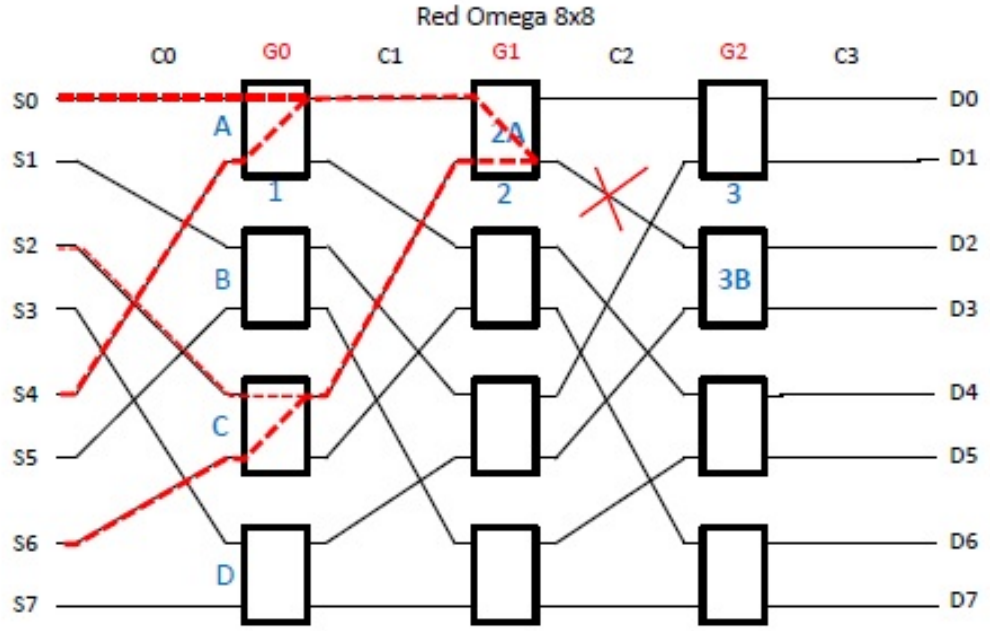
```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c;        /* S1 */
    X[i] = X[i] + c;        /* S2 */
    Z[i] = Y[i] + c;        /* S3 */
    Y[i] = c - Y[i];        /* S4 */
}
```

```
for (i=0; i<100; i=i+1 {
    T[i] = X[i] / c;    /* Y renamed to T to remove output dependence */
    X1[i] = X[i] + c; /* X renamed to X1 to remove antidependence */
    Z[i] = T[i] + c;    /* Y renamed to preserve true dependence*/
    Y[i] = c - T[i];
}
```

4.6. Explicación de ejercicios resueltos

Ejercicio 1. Suponiendo que se corta la conexión entre los conmutadores 2A (segunda etapa, primera fila) y 3B (tercera etapa, segunda fila) de una red Omega 8x8, ¿qué conexiones entre nodos fuente y destino dejarían de ser posibles?

Solución ejercicio 1:

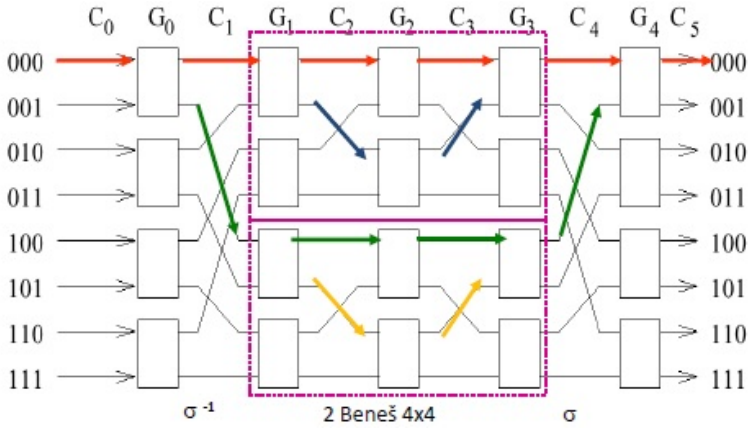


La consecuencia cortada:

- D2 queda desconectado de: S0, S4, S2 y S6.
- D3 queda desconectado de las mismas fuentes.

Ejercicio 2. Listar todos los caminos que van del nodo fuente 000 al nodo destino 000 en una red Brenes 8x8.

Solución ejercicio 2:



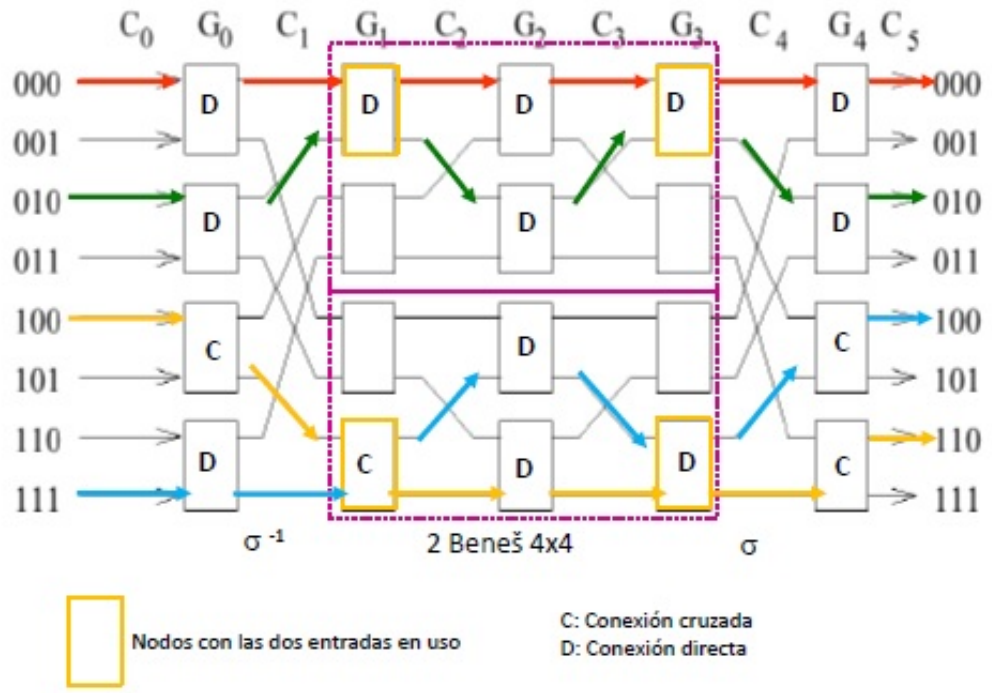
En la tabla vamos a poder ver los distintos caminos.
En la etapa G0 hay caminos: directo (D) y cruzado (C).
En la etapa G1, para cada uno de los anteriores, hay dos caminos: directo y cruzado.

Con esto ya llegamos a G2 (etapa central). A partir de ahí, los caminos divergentes tienen que converger hacia D0.

	G0	G1	G2	G3	G4
Cam 1	D	D	D	D	D
Cam 2	D	C	D	C	D
Cam 3	C	D	D	D	C
Cam 4	C	C	D	C	C

Ejercicio 3. Hallar los caminos que en una red Benes 8x8 conectan simultáneamente los siguientes pares (fuente, destino): (0,0), (2,2), (4,6) y (7,4).

Solución ejercicio 3:



Ejercicio 4. Una red de conmutación conecta 4096 procesadores a 4096 módulos de memoria "infinitamente" rápidos. Suponiendo que cada procesador lanza una instrucción por ciclo de reloj, que la frecuencia del procesador es 500 Mhz, que cada elemento de conmutación tiene un retardo de 0.2 nanosegundos, y que cada referencia a memoria implica un acceso a un módulo de memoria, indica cuántos delay slots se necesitan para instrucción de LOAD:

A. En el caso de una red Omega.

B. En el caso de una red Benes.

Solución ejercicio 4:
Solución apartado A:

Para calcular el nº de etapas tenemos en cuenta: $N = n^k$, donde N=puertos, k=base (gran conmutador)=2, n = etapas, $n = \log_2 4096 = 12$ etapas.

$$tc = 2ns; tr(\text{retardo conmutador}) = 0{,}2ns$$

El Delay slots de un Load viene dado: Procesador → Mem (envío de la dir)+Mem→ Procesador (recepción del dato).
Como 1 ciclo=2ns, esto implica que el delay slot debe ser un entero de ciclos mayor o igual que 4.8/2. Es decir un delay slot de 3 ciclos.
Solución apartado B:

$$\text{Etapas} = 2 \times 12 - 1 = 23$$

$$\text{t acceso a Mem} = 2 \times (0{,}2 \times 23) = 9{,}2ns \rightarrow \text{Delay slot} = 5 \quad \text{ciclos}$$

Ejercicio 5. Considera el uso de conmutadores 4x4 para desarrollar una red Omega 1024x1024.

A. ¿Cuántas etapas requieren la red?

B. ¿Cuántos conmutadores hacen falta?

Solución ejercicio 5:
Para calcular el nº de etapas tenemos en cuenta: $N = n^k$, donde N=puertos, k=base (gran conmutador)=2, n = etapas, $n = \log_2 4096 = 12$ etapas.
Solución apartado A:

$$N = 1024 = 2^{10}, k = 4 \rightarrow N = 4^n$$

Igualamos:

$$2^{10} = 4^n; (2^2)^5 = (2^2)^n \rightarrow n = 5 \quad \text{etapas}$$

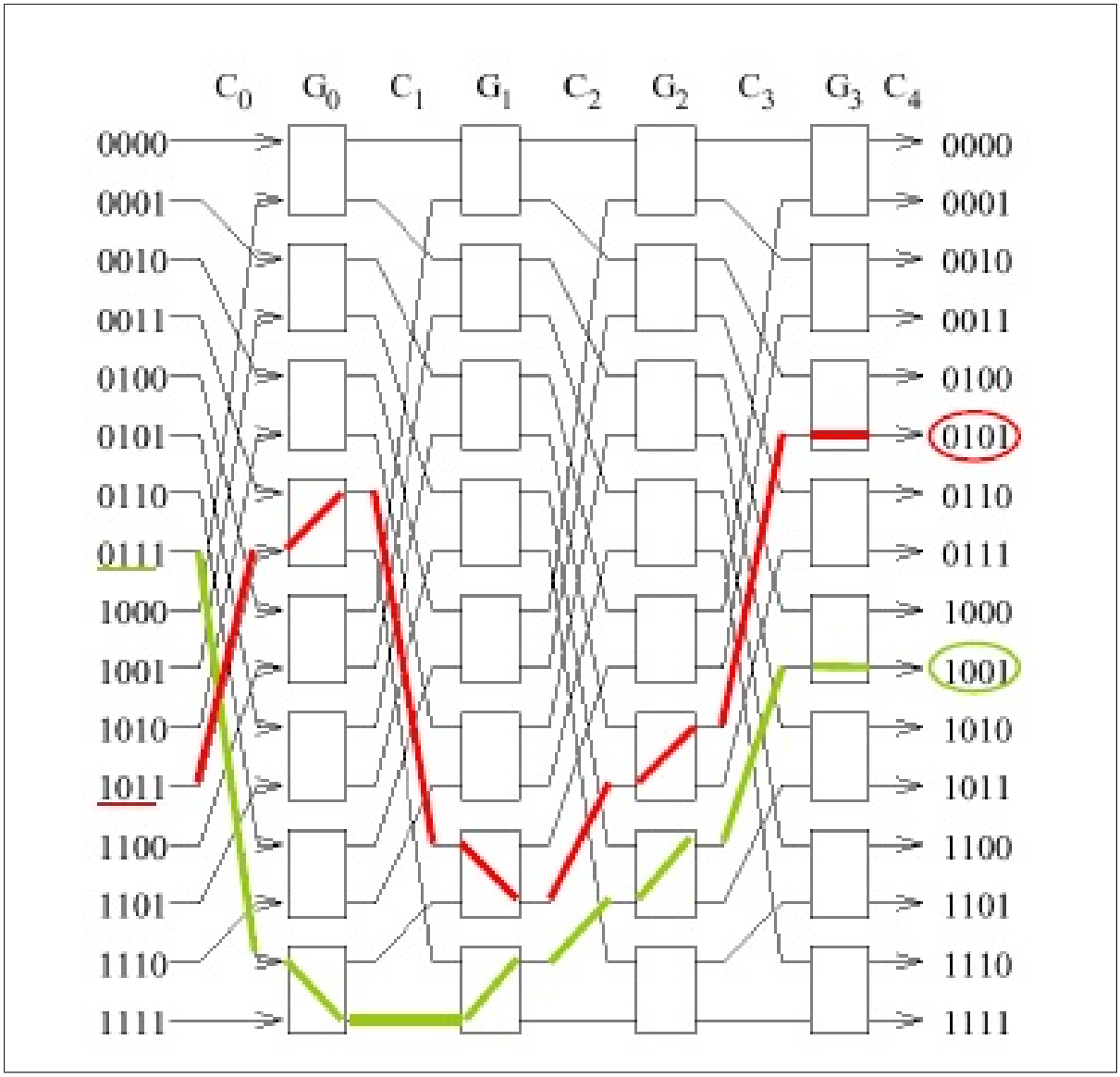
Solución apartado B:

$$\text{Nº de conmutadores} = (2^{10}/4) \times 5 = 256 \times 5 = 1280 \quad \text{conmutadores}$$

Ejercicio 6. En una red Omega de 16 entradas, usando conmutadores 2x2, determina si es posible el envío simultáneo de un mensaje desde el nodo 11 al nodo 5 y otro desde el nodo 7 al nodo 9.

Solución ejercicio 6:
Conectar 11→ 5 y 7→9.
Dir. destino: (d3,d2,d1,d0).
Selecionamos los conmutadores en la etapa $n - i - 1$:

- di=0→ Salida superior.
- di=1 → Salida inferior.



Ejercicio 7. Considera el uso de conmutadores 8x8 para construir una red Omega de 512 entradas.

A. ¿Cuántas etapas se requieren en la red?

B. ¿Cuántos conmutadores se necesitan para construir la red?

C. Si la red se expande para que tenga 4096 nodos. ¿Cuántos conmutadores adicionales 8x8 se necesitan?

Solución ejercicio 7:
Para calcular el nº de etapas tenemos en cuenta: $N = n^k$, donde N=puertos, k=base (gran conmutador)=2, n = etapas, $n = \log_2 4096 = 12$ etapas.
Solución apartado A:

$$N = 512 = 2^9; k = 8 \rightarrow N = 8; 2^9 = 8^n; 2^9 = (2^3)^n \rightarrow n = 3 \quad \text{etapas}$$

Solución apartado B:

$$\text{N}^{\text{o}} \text{ conmutadores} = (2^9/8) \times 3 = 64 \times 3 = 192 \quad \text{conmutadores}$$

Solución apartado C:

$$N = 4096 = 2^{12} \quad \text{nodos}$$

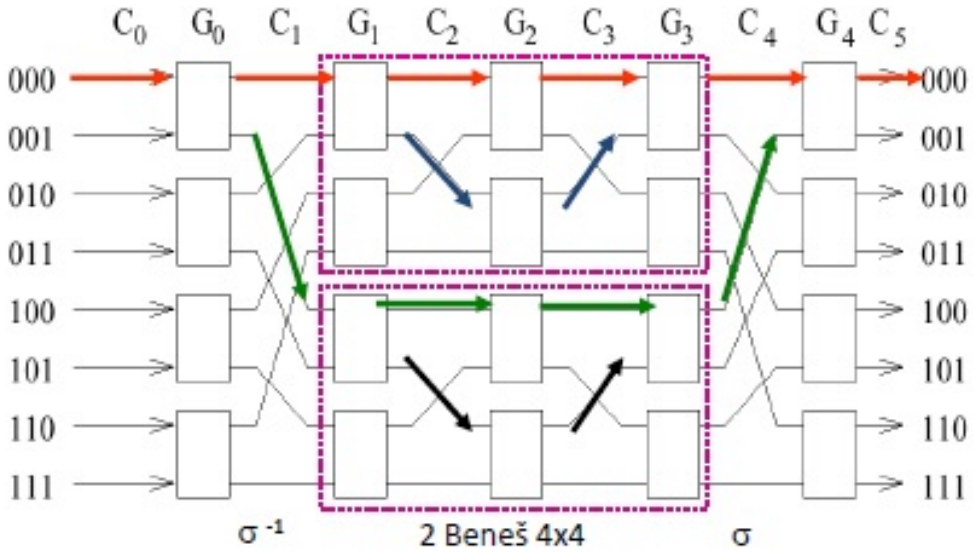
$$2^{12} = (2^3)^n \rightarrow n = 4 \quad \text{etapas}$$

$$\text{N}^{\text{o}} \text{ de conmutadores} = (2^{12}/8) \times 4 = 2^{11} = 2048 \quad \text{conmutadores}$$

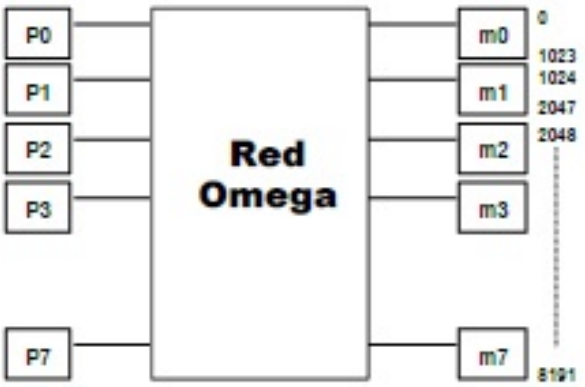
$$\text{Conmutadores adicionales} = 2048 - 192 = 1856 \quad \text{conmutadores}$$

Ejercicio 8. Considerando exclusivamente las redes de interconexión multietapa estudiadas, dibuja el esquema de un conmutador no bloqueante de 8 entradas ($k = 8$) construido a partir de conmutadores de 2 entradas e indica su latencia (expresada en función de la latencia de los conmutadores de 2 entradas).

Solución ejercicio 8:
Latencia 5 veces la del conmutador de 2 entradas.



Ejercicio 9. Se dispone de un multiprocesador de memoria compartida con 8 procesadores en el que las operaciones de acceso a memoria se realizan síncronamente. La memoria consta de 8 bancos de 1KB con un tiempo de acceso de 2 ns. La red de interconexión es una red omega basada en conmutadores de orden 2 con un retardo de 0.5 ns. Dichos conmutadores no tienen capacidad de almacenamiento temporal para secuenciar los conflictos de acceso al mismo puerto de salida. Para resolver dichos conflictos se anula la operación procedente del puerto inferior (puerto 0). Cuando la operación de acceso tiene éxito los procesadores reciben de la memoria la correspondiente confirmación: el dato leído, si se trata de una lectura, o una información específica de confirmación en caso de escritura. Si no se recibe confirmación, la operación se vuelve a planificar en el siguiente ciclo de acceso. Dada la distribución del espacio de direcciones entre los módulos que se indica en la figura y el perfil de acceso a memoria siguiente:



P0	P1	P2	P3	P4		P5	P6	P7
1025 2048 8000	100 2500 7500	3000	0 1 2	5020		7800	4000 4001	670

Calcular:

A. Tiempo de finalización de cada uno de los procesadores para el perfil de referencia especificado.

B. Ganancia de velocidad (Speedup) que se obtendría con un crossbar. Se supone que el retardo del crossbar es 4 veces el de un conmutador de orden 2.

C. ¿Qué problemas plantea la política utilizada para resolver los conflictos?¿Cómo podría resolverse?

Solución ejercicio 9:
Primero realizamos un esquema con los ciclos de acceso y con los intentos.

Ciclo	P0	P1	P2	P3	P4	P5	P6	P7
Intento 1	Dir: 1025 Con: 0->1	100 1->0	3000 2->2	0 3->0	5020 4->4	7800 5->7	4000 6->3	670 7->0
Exitos 1	✓	×	×	×	✓	✓	✓	✓
Intento 2	Dir: 2048 Con: 0->2	100 1->0	3000 2->2	0 3->0			4001 6->3	
Exitos 2	×	×	×	✓			✓	
Intento 3	Dir: 2048 Con: 0->2	100 1->0	3000 2->2	1 3->0				
Exitos 3	×	×	✓	✓				
Intento 4	Dir: 2048 Con: 0->2	100 1->0		2 3->0				
Exitos 4	✓	×		✓				
Intento 5	Dir: 8000 Con: 0->7	100 1->0						
Exitos 5	✓	✓						
Intento 6		2500 1->2						
Exitos 6		✓						
Intento 7		7500 1->7						
Exitos 7		✓						

Trazamos primero los caminos que tienen prioridad en color azul. A continuación los que no tienen prioridad hasta donde son compatibles con los casos anteriores, en rojo.

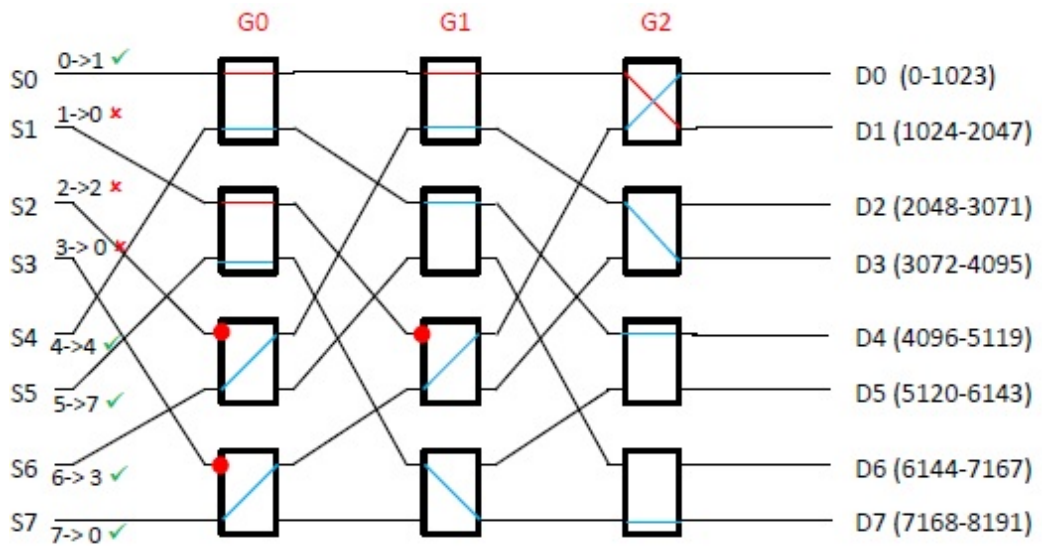


Figura 4.7: Ciclo de acceso 1

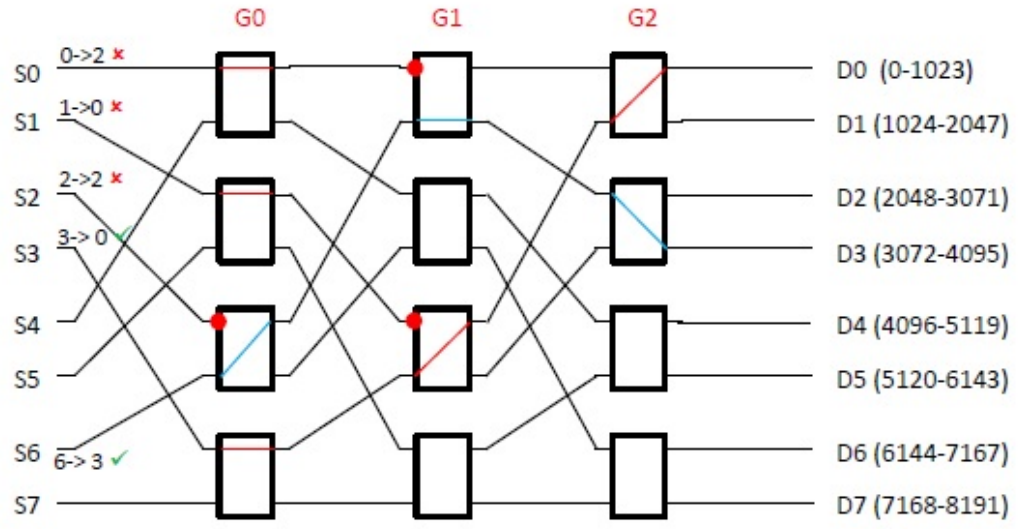


Figura 4.8: Ciclo de acceso 2

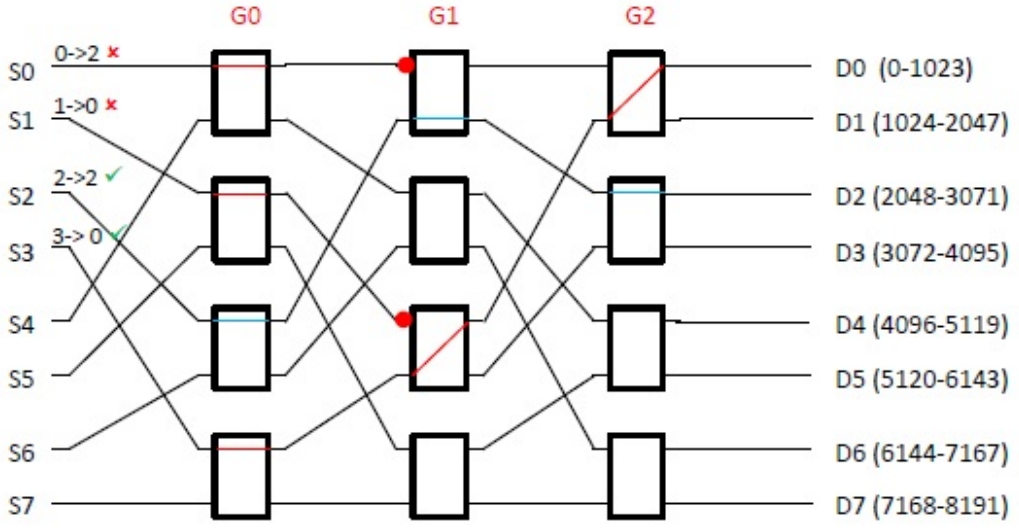


Figura 4.9: Ciclo de acceso 3

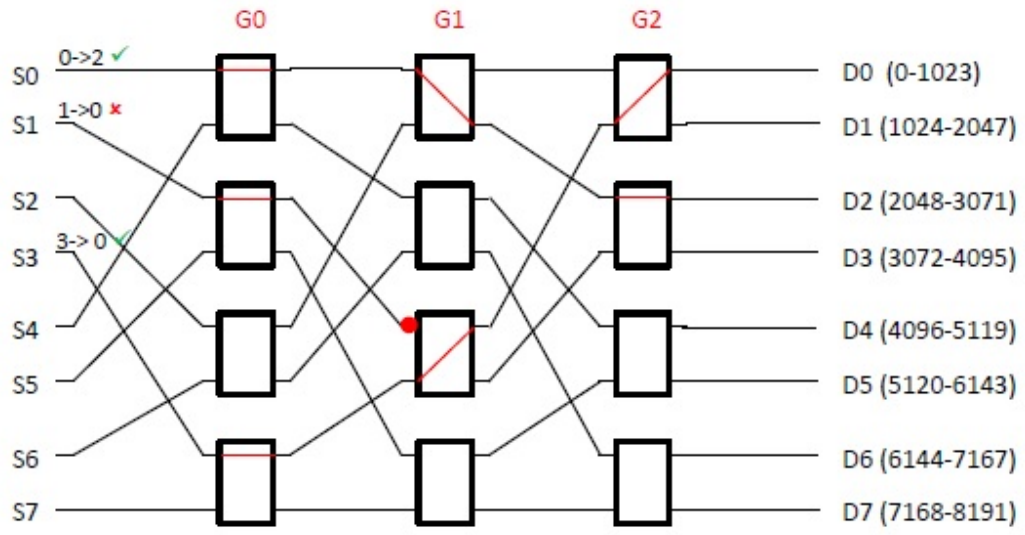


Figura 4.10: Ciclo de acceso 4

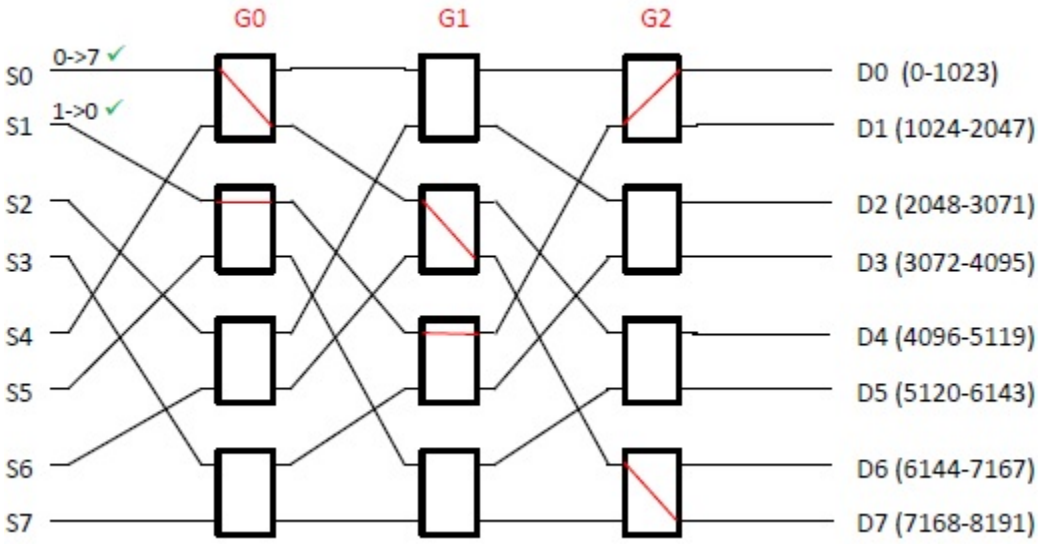


Figura 4.11: Ciclo de acceso 5

Solución apartado A:

$$T_{RC} = 0{,}5ns, T_{\text{mem}} = 2ns, T_{\text{com}} = \text{N}^{\text{o}} \text{ de etapas} \times T_{\text{RC}} = 3 \times 0{,}5 = 1{,}5ns$$

$$T_{\text{ciclo}} = 2 \times T_{\text{com}} + T_{\text{mem}} = 2 \times 1{,}5 + 2 = 5ns$$

Tiempo total para cada procesador =(nº ciclos Pi)× T_{ciclo}.

$$\text{P0} : T_{\text{total}} = 5 \times 5 = 25ns$$
$$\text{P1} : T_{\text{total}} = 7 \times 5 = 35ns$$
$$\text{P2} : T_{\text{total}} = 3 \times 5 = 15ns$$
$$\ldots$$
$$\ldots$$

Caso más lento P1: 35 ns.
Con Crossbar: En accesos a mismo destino se anulan las operaciones procedentes de puertos inferiores.

Ciclo	P0	P1	P2	P3	P4	P5	P6	P7
Intento 1	Direc: 1025 Conex: 0->1	100 1->0	3000 2->2	0 3->0	5020 4->4	7800 5->7	4000 6->3	670 7->0
Exitos 1	✓	×	✓	×	✓	✓	✓	✓
Intento 2	Direc: 2048 Conex: 0->2	100 1->0		0 3->0			4001 6->3	
Exitos 2	✓	×		✓			✓	
Intento 3	Dir: 8000 Con: 0->7	100 1->0		1 3->0				
Exitos 3	✓	×		✓				
Intento 4		100 1->0		2 3->0				
Exitos 4		×		✓				
Intento 5		100 1->0						
Exitos 5		✓						
Intento 6		2500 1->2						
Exitos 6		✓						
Intento 7		7500 1->7						
Exitos 7		✓						

Solución apartado B:
Con un Crossbar $\text{T}_{\text{ciclo}} = 2 \times T_{\text{com}} + T_{\text{mem}} = 2 \times (0{,}4 \times 0{,}5) + 2 = 6$ ns.
El Xbar tiene conectividad total, pero en un ciclo no puede haber varios procesadores accediendo a un mismo módulo de memoria. La tabla anterior muestra el secuenciamiento de accesos, suponiendo que en caso de conflicto, la política de prioridades anula, igual que en el apartado A, las peticiones procedentes de las entradas con los números inferiores.
Como muestra la tabla, con esta política de prioridades, hacen falta 7 ciclos. Con otras políticas el nº de ciclos puede variar. Luego:

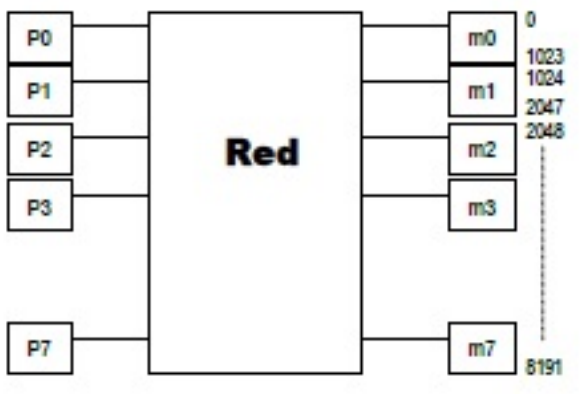
$$T_{\text{total}} = 7 \times 6 = 42ns$$

$$\text{Speedup} = 35/42 = 0{,}83$$

La velocidad no mejora, sino que empeora.

Solución apartado C:
Posterga durante mucho tiempo los accesos de algunos procesadores. Vía de solución: alternar la política de prioridades.

Ejercicio 10. Se dispone de un multiprocesador de 8 procesadores con memoria cache L1 compartida en el que las operaciones de acceso a memoria cache se realizan síncronamente. La memoria cache consta de 8 bancos de 1KB con un tiempo de aceso de 2 ns. La red de interconexión está basada en conmutadores de orden 2 con un retardo de 0.5 ns. Dichos conmutadores no tienen capacidad de almacenamiento temporal para secuenciar los conflictos de acceso al mismo puerto de salida. Para resolver dichos conflictos se anula la operación procedente del puerto superior (puerto 1). Cuando la operación de acceso tiene éxito los procesadores reciben la memoria correspondiente confirmación: el dato leído, si se trata de una lectura, o una información específica de confirmación en caso de escritura. Si no se recibe confirmación, la operación se vuelve a planificar en el siguiente ciclo de acceso a memoria. Dada la distribución del espacio de direcciones entre los módulos que se indica en la figura y el perfil de acceso a la memoria siguiente:



P0	P1	P2	P3	P4	P5	P6	P7
6050	5000	3110	2050	6200	7800	1030	670
6051	5001	3111	2051	6201	7801	1031	671
6062	5002	3112	2052	6202	7802	1032	672
6053	5003	3113	2053	6203	7803	1033	673

Calcular el número de ciclos de memoria necesarios para resolver todas las referencias, el tiempo de ciclo mínimo, y el coste de la red en función del coste por conmutador, para:

A. Interconexión por medio de red Omega.

B. Interconexión por medio de red de Benes.

C. Interconexión por medio de Crossbar. Se supone que el retardo del crossbar es 4 veces el de un conmutador de orden 2.

Solución ejercicio 10:
Solución apartado A:

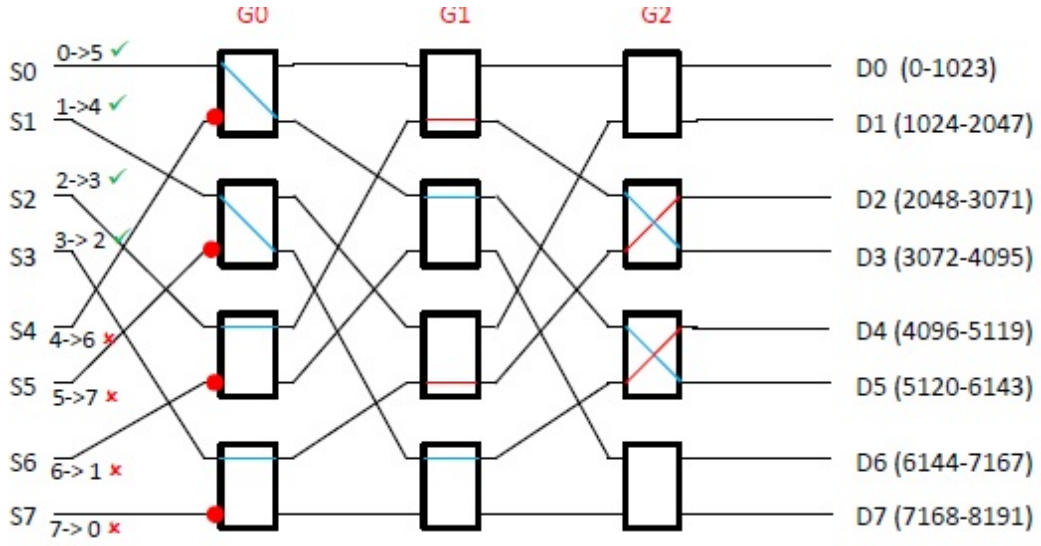


Figura 4.12: Ciclos de acceso 1-4

Solución apartado B:

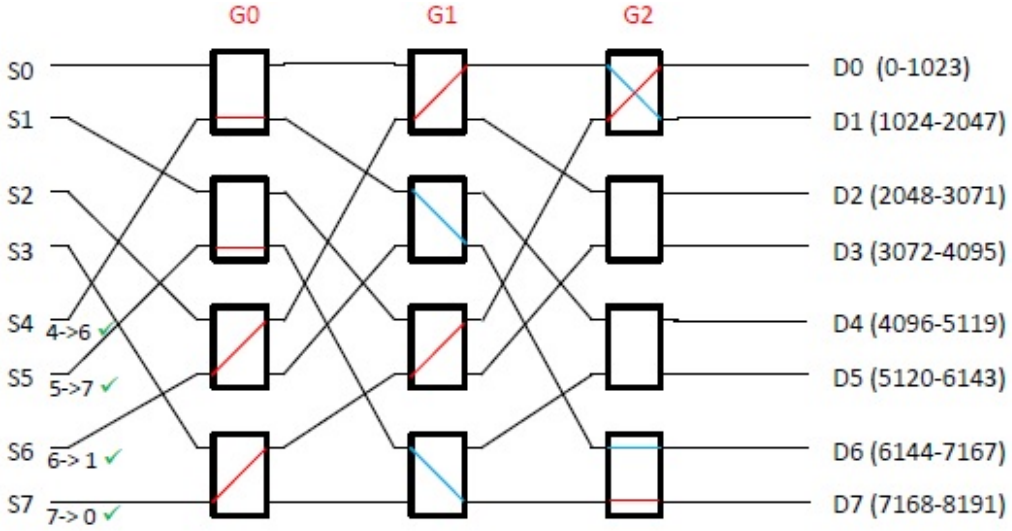
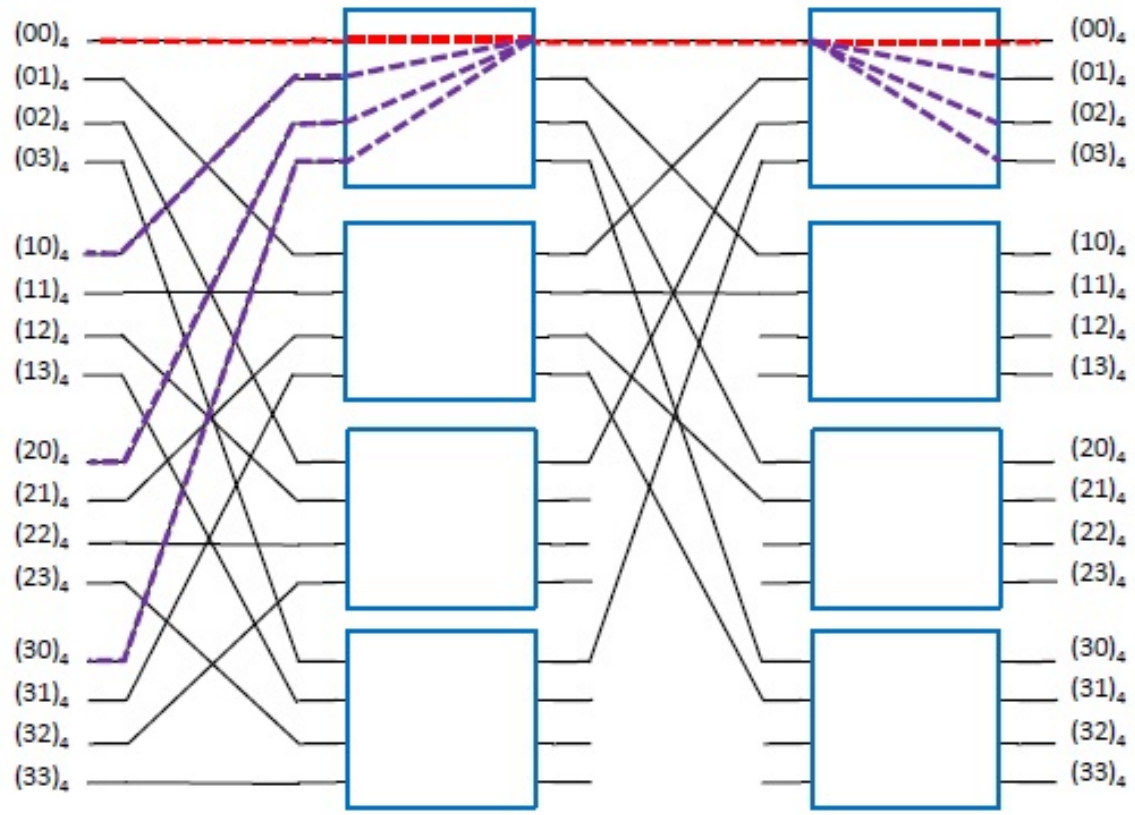


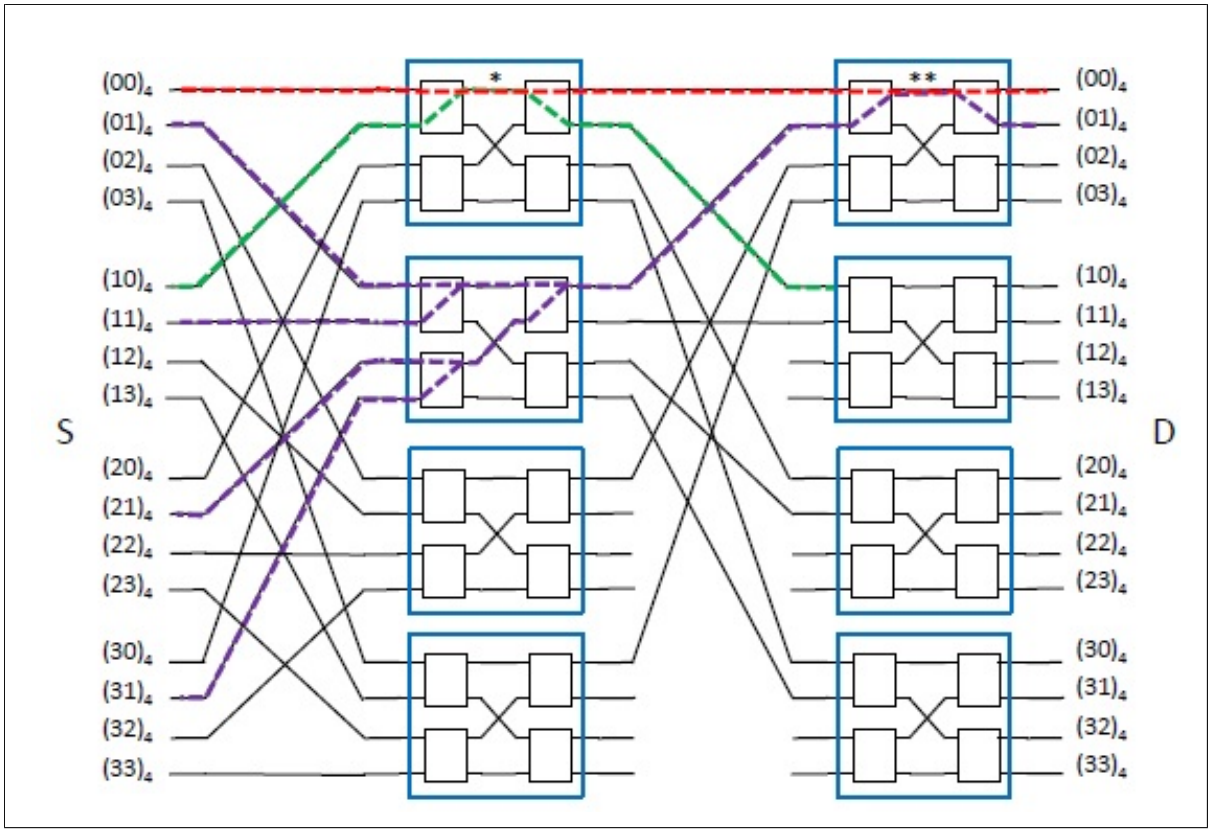
Figura 4.13: Ciclos de acceso 5-8

Ejercicio 11. Supongamos una red multietapa con topología Omega de 16 entradas formada por conmutadores de grado 4. En este caso los conmutadores de grado 4 son bloqueantes ya que suponemos que han sido implementados por medio de una red Butterfly formada por conmutadores de grado 2. Indica qué comunicaciones se bloquean debido a la contención en los canales internos de comunicación entre los conmutadores 2x2 (no debida a la contención en los canales externos entre los conmutadores 4x4), al realizar la comunicación entre la entrada 0 y la salida 0 de la red multietapa.

Solución ejercicio 11:
Preliminares: Conexiones bloqueadas al conectar 0→0.



Línea roja: Conexiones internas * y ** quedan ocupadas al conectar 0→0.
Línea verde: Conexiones bloqueadas por el uso de *. S4 con D4, D5, D6 y D7.
Línea morada: Conexiones bloqueadas por el uso de **. S1, S5, S9 y S13 con D1.



Ejercicio 12. Supongamos una red Omega de 16 entradas que puede implementarse con conmutadores de grado 2 o de grado 4, con las siguientes características:

- Para los conmutadores de grado 2, el tiempo $\triangle$, empleado en decidir la ruta y establecer la comunicación con el canal de salida adecuado es de 4 ns.
- Para los conmutadores de grado 4, el tiempo $\triangle$, empleado en decidir la ruta y establecer la comunicación con el canal de salida, adecuado es de 6 ns.
- En todos los casos el ancho de banda de los canales es de 1 GB/s y la longitud del mensaje es de 128 bits.

Determinar, en todos los casos, el tiempo de transmisión de un mensaje completo usando conmutadores de 2 o de 4 entradas y usando conmutación de paquetes con "Store-and-Forward" o "Cut-Through".

Solución ejercicio 12:
Con conmutador de grado 2 → Nº etapas: h=4.
Con cut-through:

$$T_{CT2} = h \triangle_2 + n/B = 4 \times 4 \times 10^{-9} + (128/8 \times 10^9) = 32 \times 10^{-9} s = 32ns$$

Con store-and-forward:

$$T_{SF2} = h(\triangle_2 + n/B) = 4 \times [4 \times 10^{-9} + (128/8 \times 10^9)] = 80ns$$

Con conmutadores de grado 4→ Nº etapas: h=2. Con cut-through:

$$T_{CT4} = h \triangle_4 + n/B = 2 \times 6 \times 10^{-9} + (128/8 \times 10^9) = 28ns$$

$$T_{SF4} = h(\triangle_4 + n/B) = 2 \times [6 \times 10^{-9} + (128/8 \times 10^9)] = 44ns$$

Capítulo 5

Tema 5: Multiprocesadores y redes de interconexión

En este capítulo vamos a ver los siguientes conceptos.

- Introducción.
- Arquitecturas paralelas.
  - Memoria compartida.
  - Paso de mensajes.
- Redes de interconexión.
  - Descripción estructural.
  - Topología de Red.
    - Redes estáticas.
    - Redes dinámicas.
  - Tecnologías de conmutación.
  - Encaminamiento (routing).
- Explicación de ejercicios resueltos.

5.1. Introducción

Definición

Colección de elementos de procesamiento que cooperan y se comunican para resolver problemas grandes "rápidamente".

Objetivo:

- Rendimiento/ Eficiencia.
- Replicación (tolerancia a fallos).

Factores determinantes:

- Agotamiento de ILP: Consumo de energía y coste crecen más rápido que rendimiento.
- Creciente interés en servidores de altas prestaciones.
- Aumento de las aplicaciones intensivas en datos (Big Data).
- Mejora el conocimiento sobre cómo usar eficazmente los multiprocesadores.
- Aprovechamiento de los diseños vía replicación del procesador.

5.2. Arquitecturas Paralelas

Definición

La arquitectura paralela o de líneas paralelas (pipe-line) es una técnica en la que se descomponen un proceso secuencial en suboperaciones, y cada subproceso se ejecuta en un segmento dedicado especial que opera en forma concurrente con los otros segmentos.

Cada segmento consta de un registro de entrada seguido de un circuito combinatorio. El registro contiene los datos y el circuito combinatorio ejecuta la suboperación en el segmento particular. La salida del circuito combinacional es un segmento dado se aplica al registro de entrada del siguiente segmento.

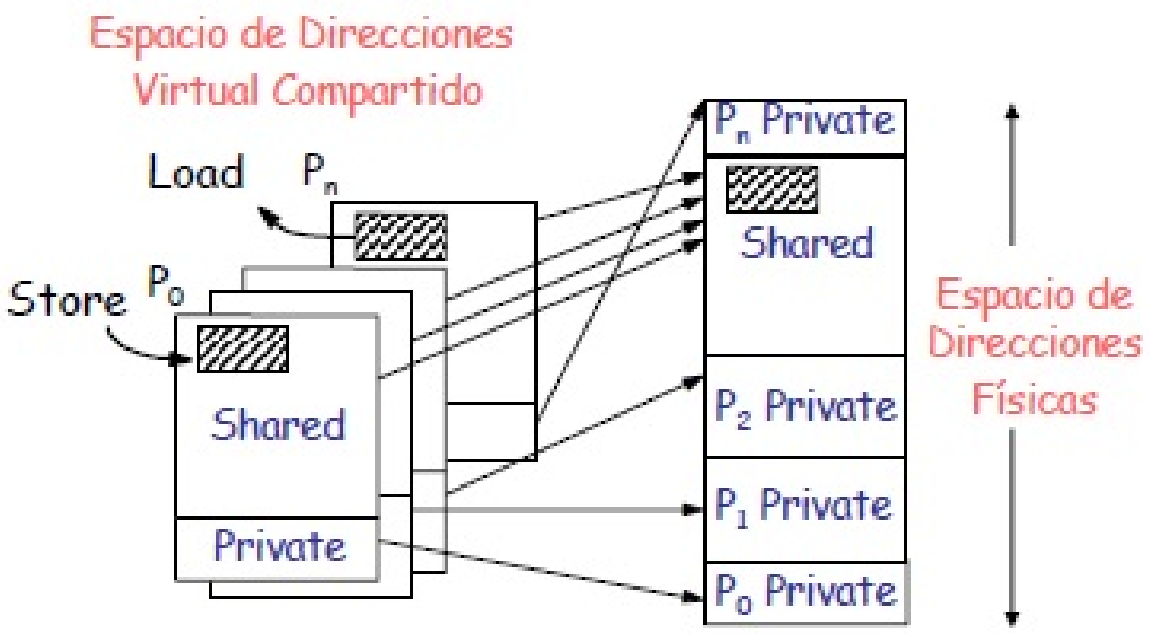
5.2.1. Arquitecturas de memoria compartida

Definición

Es un espacio de direcciones compartido.

La multiprogramación se da en monoprocesadores. Los procesos pueden configurarse de modo que parte su espacio de direcciones sea compartido:

- Comunicaciones implícitas.
- Sincronizaciones: SO, Operaciones Atómicas.



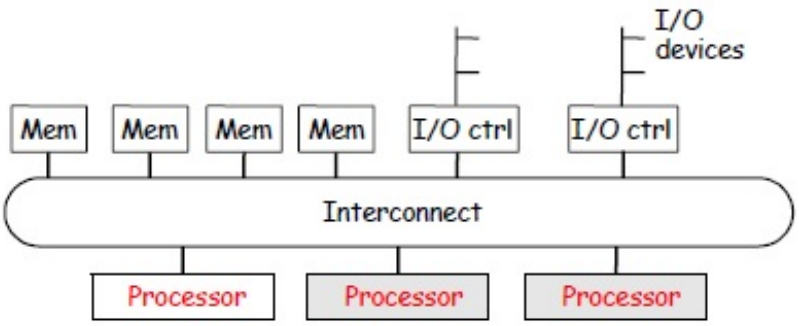
Múltiples procesos pueden tener una región de memoria compartida proyectada sobre su espacio de dirección virtual.
Las escrituras de variables residentes en esta región son visibles para el resto de los procesos.

Multiprocesador de memoria compartida

Definición

Es una extensión natural de un sistema en el que diferentes procesos pueden compartir memoria. Se puede añadir una o varios procesadores.

- Pocos procesadores: red sencilla.
- Muchos procesadores: red más compleja.



Unificada

Este es el diseño para un nº reducido de procesadores:

- Symmetric (Shared-Memory) Multiprocessor (**SMP**).
- Memoria centralizada con tiempo de acceso uniforme (**UMA**). Interconexión vía bus común. E/S compartida.
- Anchura de banda limitada. Límite en el número máximo de procesadores.
- Caches son cruciales: **Coherencia**.
- Posibilidad de usar otras redes.

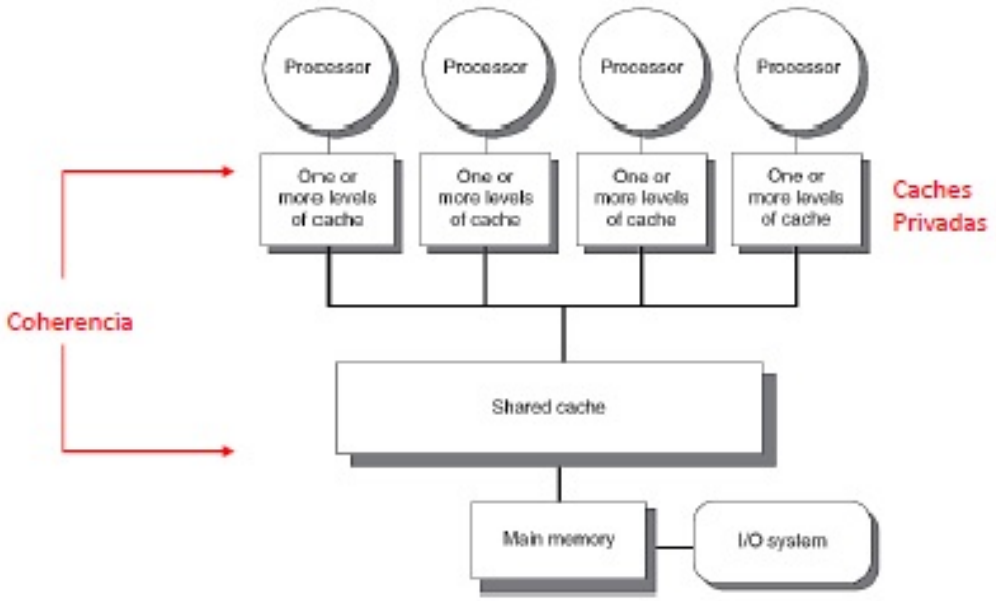


Figura 5.1: Diseño unificada

Distribuida

Diseño para un número elevado de procesadores:

- Memoria: Distribuida con tiempo de acceso no uniforme (**NUMA**).
- Interconexión escalable.
- Anchura de banda crece con el número de EP.

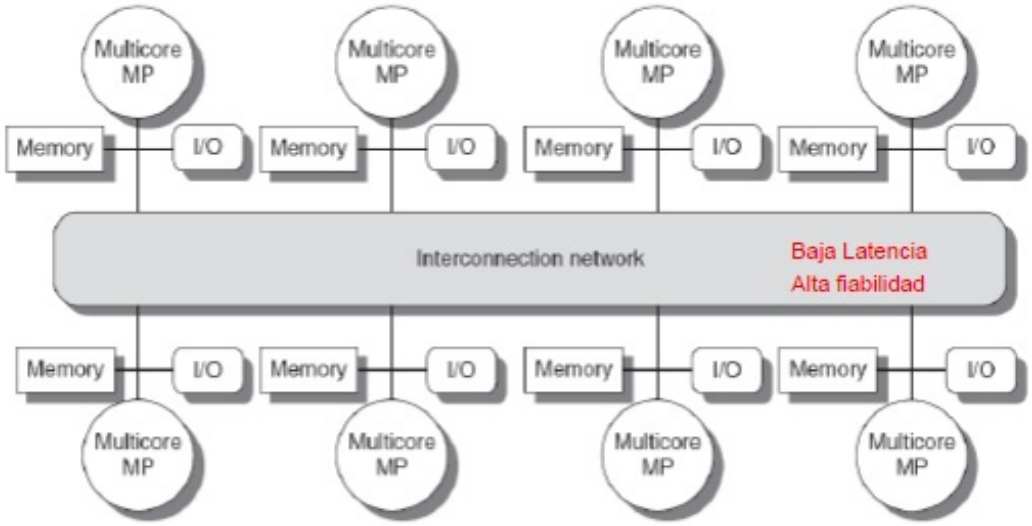
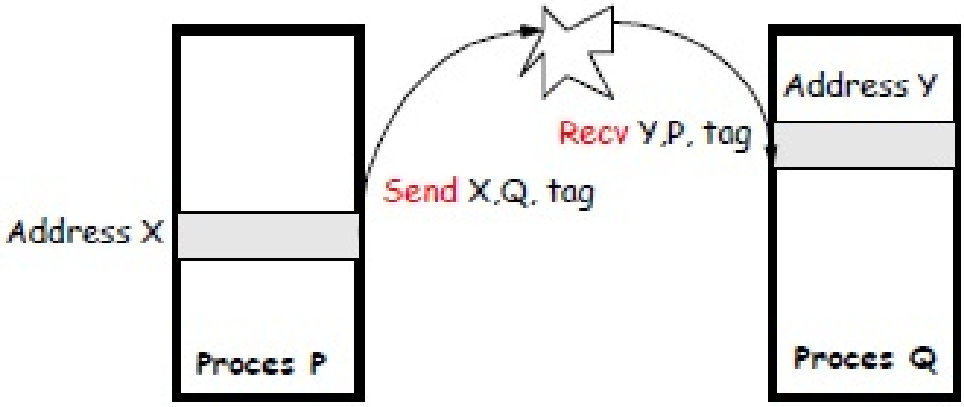


Figura 5.2: Diseño distribuida

5.2.2. Arquitecturas de Paso de Mensajes

Se requiere un computador completo.
Modelo de programación:

- Acceso directo al espacio de direcciones privado (memoria local).
- Comunicación explícita mediante operaciones I/O.



Sobrecarga: Construir la cabecera del mensaje: copiar los datos en el buffer de la red; enviar el mensaje; copiar los datos en el buffer receptor, copiar los datos del espacio sistema operativo al espacio usuario.
Diagra de Bloques Semejante a NUMA: Las comunicaciones están integradas en el sistema E/S en lugar de en el sistema de memoria.

5.3. Redes de interconexión

5.3.1. Introducción

Objetivo

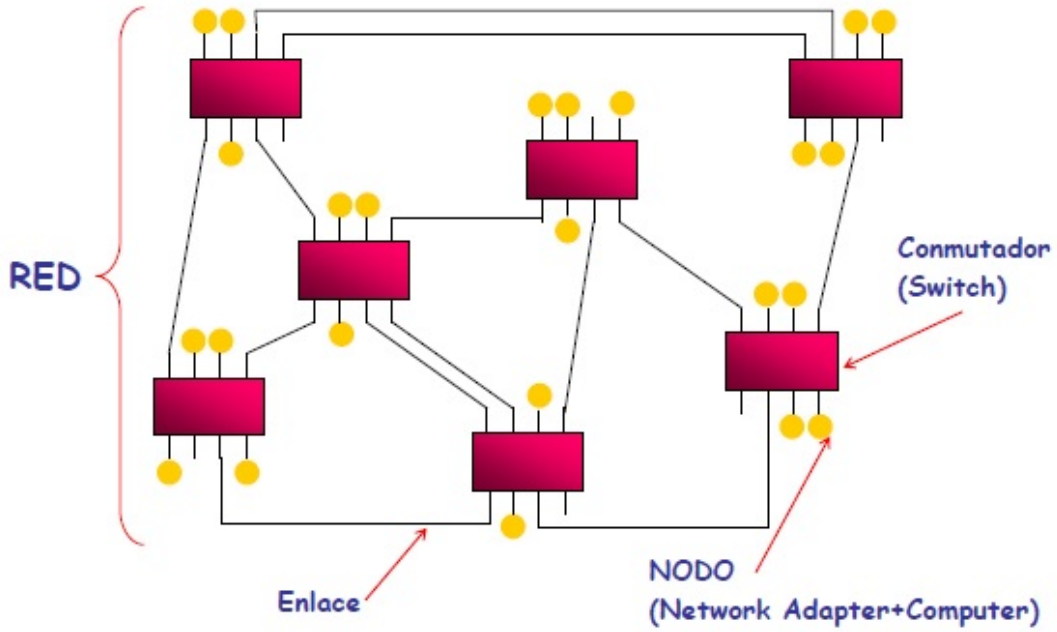
Es dar un servicio de comunicación de datos que involucre diversas redes con diferentes tecnologías de forma transparente para el usuario.

Factores de diseño:

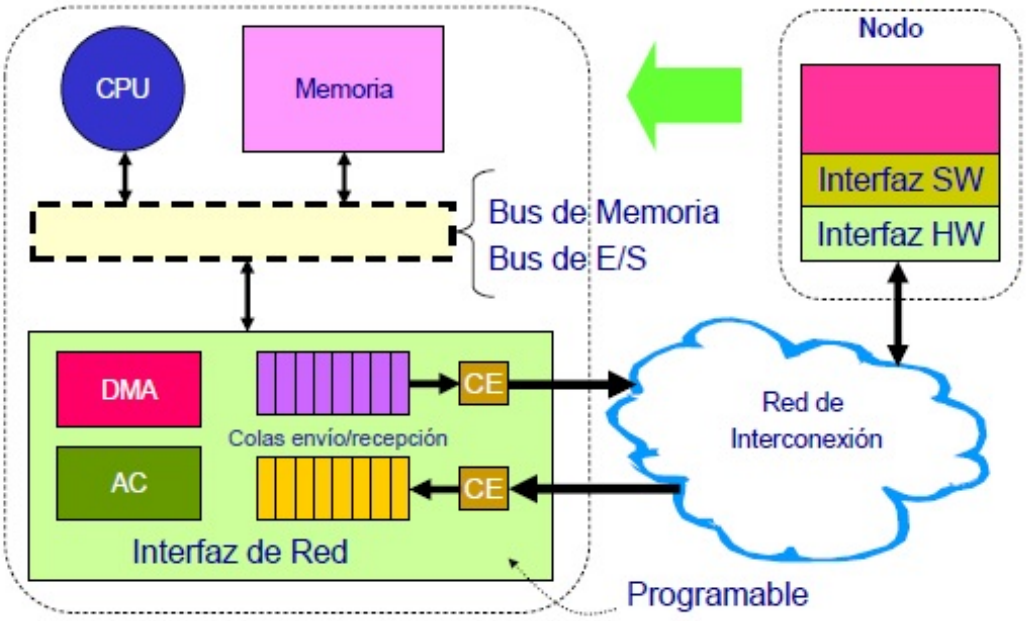
- Prestaciones de la red.
- Latencia.
- Productividad.
- Escabalidad.
- Fiabilidad.
- Expansión incremental.
- Participado y seguridad.
- Simplicidad y bajo coste.
- Distancia de expansión.
- Consumo/disipación de energía.

5.3.2. Descripción estructural

Elementos



Interfaz de red



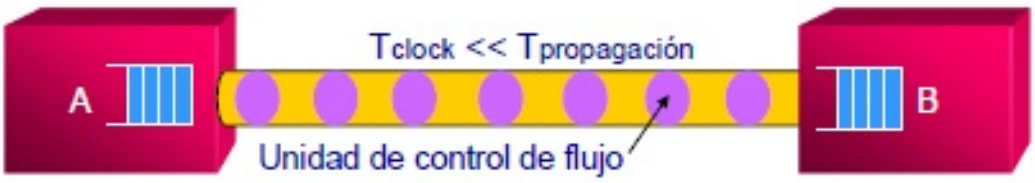
Mensaje:

- Unidad lógica de comunicación.
- Nivel de aplicación.

Paquete:

- Unidad de transferencia entre interfaces de red.
- Menor unidad que contiene información de encaminamiento.
- Mecanismo de abstracción:
  - Encapusulado.
  - Fragmentado.

Enlace

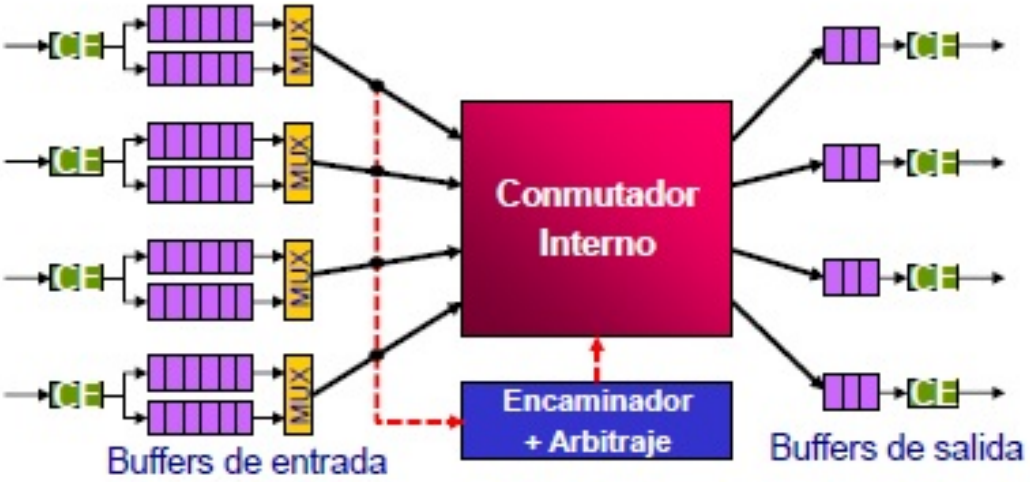


Características:

- Ancho: w.

- Velocidad de señal (frecuencia): f=1/t.
- Ancho de banda: B=wf.
- Unidad de transferencia por ciclo: Phit=w.
- Unidad de control de flujo: Flit.
- Longitud:
  - Corto: Un solo phit en el enlace.
  - Largo: Varios phit en el enlace.

Conmutador/ router



Características:

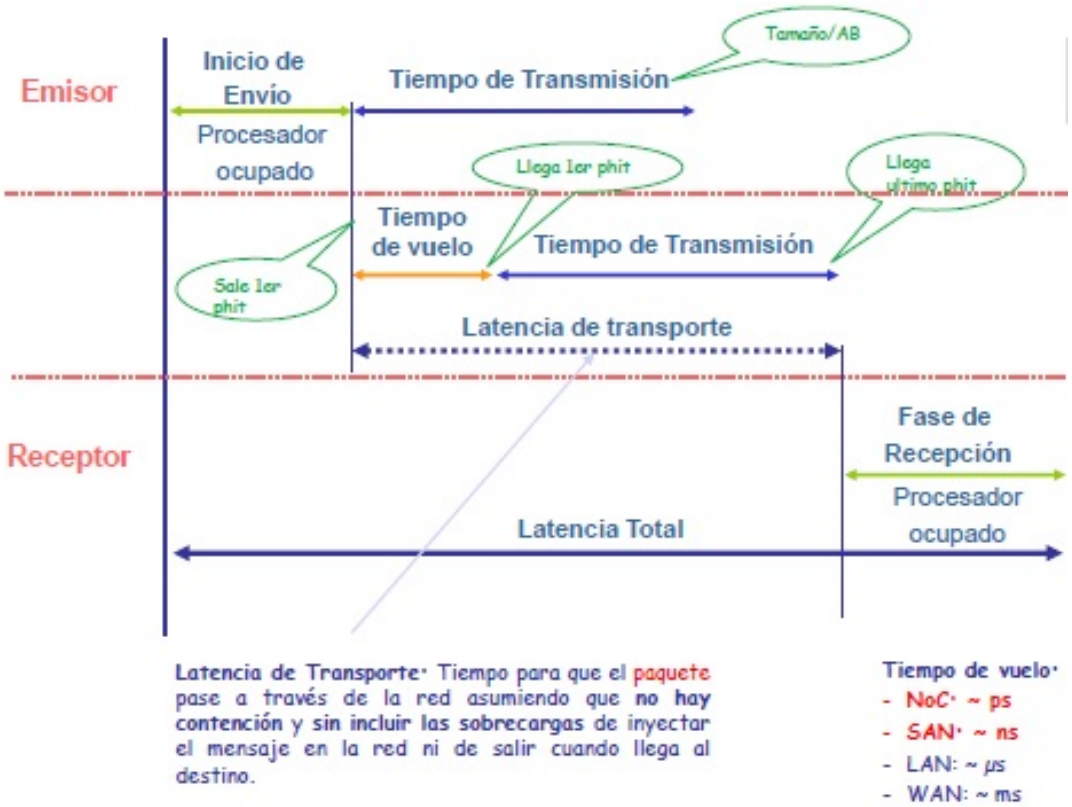
- Grado conmutador/ nodo. Número de canales de entrada.
- Implementación conmutador interno.
- Buffers de entrada/salida.
- Lógica de control: Arbitraje. Encaminamiento.

Red

Red de intersección: Grafo de V conectada por canales de comunicación.
Ruta: Secuencia de conmutadores/nodos y canales seguidos por un mensaje.
Características de la red:

- Topología: Estructura de la interconexión física de la red.
- Conmutación: Cómo atraviesan su ruta los datos de un mensaje.
- Encaminamiento: Que rutas pueden seguir los mensajes a través del grafo de la red.
- Control de flujo: Momento del mensaje a lo largo de la ruta.

Rendimiento

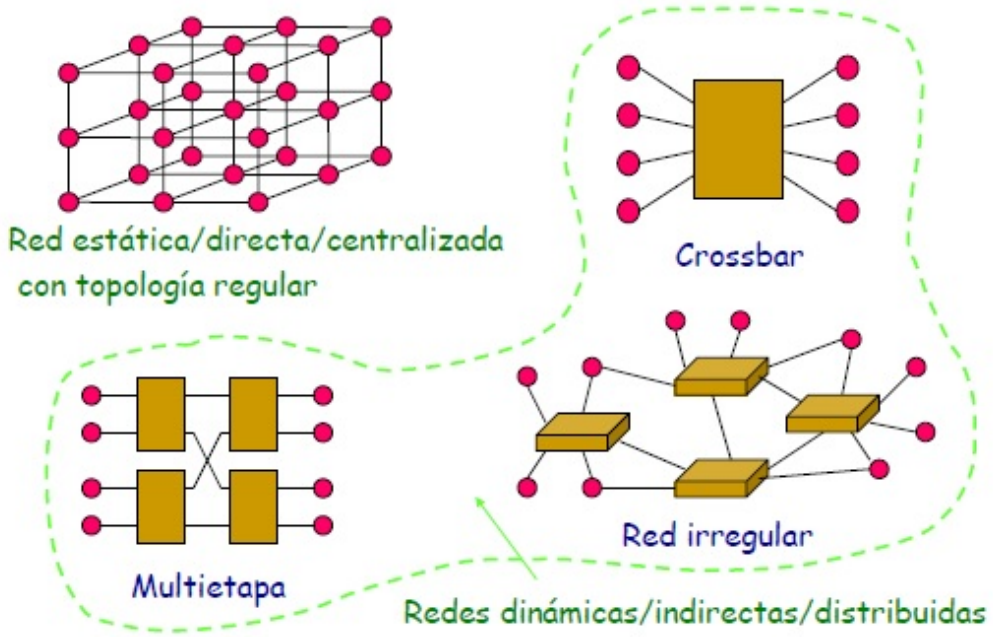


5.3.3. Topología de red

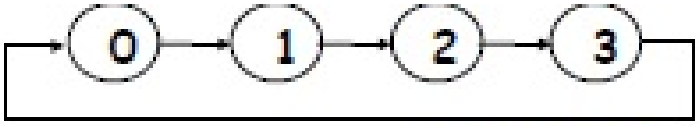
Definiciones y propiedades:

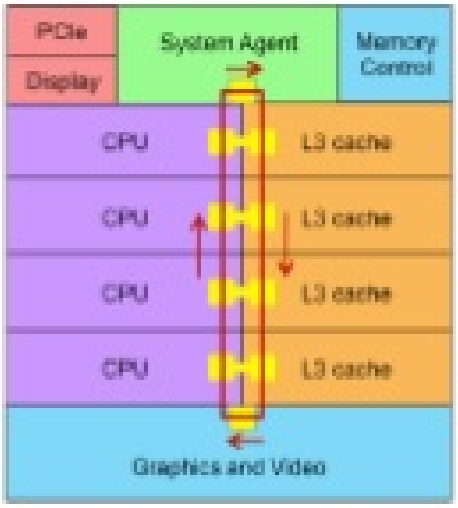
- Red de interconexión: Grafo de V conectada por canales de comunicación.
- Grado de un nodo/conmutador(d): Número de canales de E/S del nodo/conmutador.
- Tamaño de la red: Número de EP conectados.
- Número de conmutadores.
- Diámetro de la red(D): Máxima longitud de los caminos ópticos entre pares de EP.
- Simetría: Visión de la red por parte de los nodos.
- Homogeneidad: Igualdad en las características de los nodos.
- Regularidad: Los nodos tienen el mismo grado.
- Ancho de bisección: Volumen de tráfico a través de esos canales.

Clasificación topológica



Redes estáticas





Ejemplo:Sandy Brigde NoC (2010):

Anillo unidireccional conecta CPUs, caches y otros módulos.

+

Conex directa CPUi-L3i

Saltos (hops) para conectar cq CPU con cq $: entre 0 y 3

Redes dinámicas

Los EP se conectan a un conmutador.
Modelo: Grafo con conmutadores como vértices y canales como arcos.
Topologías:

- Regulares: Multiprocesadores UMA, SIMD...
- Irregulares: NOW, LAN conmutadas.

Diferencias con redes estáticas:

- Un conmutador puede estar conectado a 0 o más EP.
- Sólo los conmutadores conectados a EP pueden ser origen/destino de un mensaje.

Crossbar: Red conmutada a N entradas y M salidas que permite hasta min (M, N) interconexiones punto a punto sin contención.

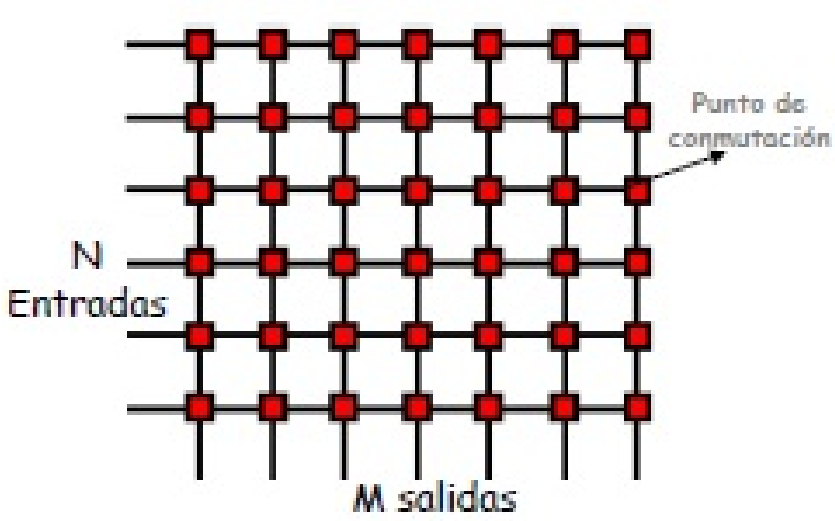


Figura 5.3: Redes dinámicas Crossbar

Multistage Interconnection Network (MIN):

- Conectan dispositivos de entrada a dispositivos de salida a través de etapas compuestas por conmutadores y patrones de interconexión.
- Cada conmutador puede conectar sus E con sus S en cualquier permutación deseada.

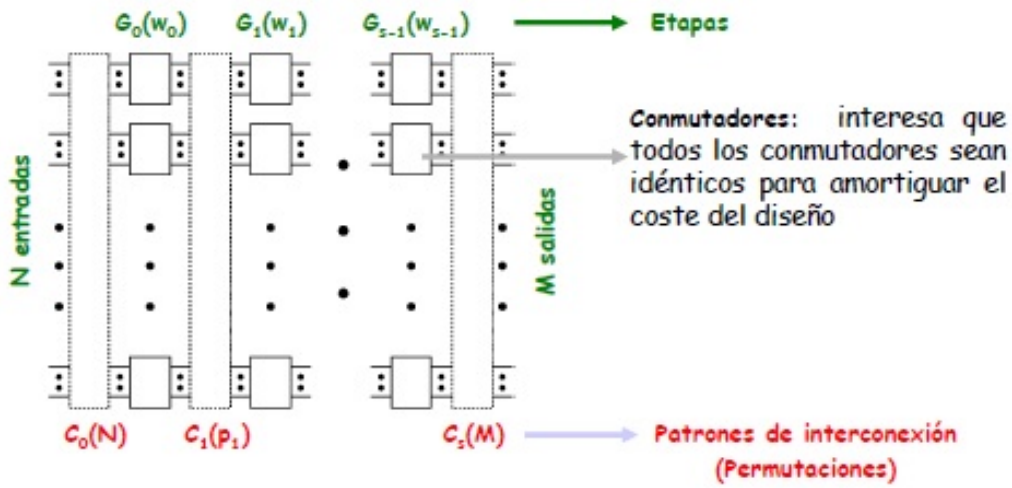


Figura 5.4: Modelo general de MIN

5.3.4. Técnicas de conmutación

Conmutación de circuitos:

- Establece cuál será la secuencia de enlaces que atravesarán los datos desde el origen al destino.
  - Un mensaje sin datos recorre el camino completo y va reservando los enlaces.
  - Cuando se enlaza el destino se devuelve un ACK al origen por el mismo camino ya establecido.
- Una vez establecido el camino todos los datos lo siguen.
- Los canales que componen el camino quedan reservados desde el comienzo hasta el final de la transmisión $\rightarrow$ durante este tiempo no puede usarse para otras comunicaciones.

Conmutación de paquetes:

- Cada decisión de rutado se va tomando según el mensaje progresa.
- Los enlaces que no estén siendo utilizados en un momento dado, pueden usarse para otra comunicación.
- Variantes:
  - Store-and-Forward.
  - Cut-Through.

Latencia

Suposiciones:

- No hay congestión en la red: latencia básica.
- Se quiere transmitir un mensaje que atravesará h enlaces.
- El tamaño del mensaje es n bits y el ancho de banda es B.
- El tiempo para decidir cuál debe ser el enlace siguiente a recorrer y establecer la conexión con ese enlace es $\triangle$.

Latencia con conmutación de circuitos

Fórmula

$$T_{cc}(n,h) = h \triangle + n/B$$

- Tiempo previo para decidir la ruta.
- Ventajas: Una vez seleccionada la ruta, la transmisión se produce a la máxima velocidad.
- Inconvenientes: Reserva de los canales.

Latencia con Store-and-Forward Latencia con conmutación de circuitos

Fórmula

$$T_{sf}(n,h) = h \triangle + h(n/B) = h(\triangle + n/B)$$

- El mensaje completo se transmite de un nodo al siguiente.
- Cuando se ha recibido completamente, se decide el siguiente enlace a seguir y se vuelve a transmitir → h transmisiones+ h decisiones.
- Menos enlaces reservados que en CC.
- Latencia mucho peor.

Latencia con Cut-Through

Fórmula

$$T_{ct}(n,h) = h \triangle + n/B$$

- El paquete está dividido en trozos de forma natural: flits.
- Cuando se recibe el primer flit decidir el siguiente enlace de la ruta y comenzar la retransmisión, sin esperar a la recepción de los restantes flits.
- La transmisión es similar a CC, pero de forma segmentada.
- No hay reserva de circuitos, pero tiempo básicamente comparable a CC.

Contención

Problema:

- Dos paquetes intentan usar el mismo canal al mismo tiempo.
- Hay arbitraje.
- El tamaño de la memoria es limitado.
- Afecta a la latencia.
- En un caso extremo, hay saturación e interbloqueos.

Store&Forward: El paquete bloqueado se recibe y se almacena completo en buffers del nodo/encaminador.
Cut-through:

- **Virtual Cut-Through**: Los flits del paquete bloqueado se van recibiendo y almacenando en buffers del encaminador.
    - En situaciones de contención → comportamiento se degrada hasta ser similar a SF.

  - Necesitamos buffers con tamaño suficiente para almacenar un paquete.

- **Wormhole**: Los buffers del encaminador sólo almacenan unos pocos flits del mensaje bloqueado. Los restantes flits permanecen en los encaminadores previos sin avanzar.

  - Buffers más pequeños.
  - Un paquete bloqueado en la red puede estar repartido entre varios encaminadores a lo largo de la ruta → Propagación hacia atrás.
  - Cuidado con interbloqueos → Ocupación extensiva de los canales.

Latencia y AB

Son dependientes del AB demandado:

- AB está demandado por debajo de la capacidad máxima de la red, la latencia se mantiene baja.
- A partir de cierto punto la latencia aumenta rápidamente y el AB entregado no se incrementa.
- Un multiprocesador es un sistema cerrado.

5.3.5. Encaminamiento

Determina la ruta que debe seguir un mensaje desde el origen hasta el destino.
Comportamiento ideal:

- Baja latencia.
- Distribución de la carga.
- Tolerancia a fallos.
- Libre de interbloqueos.

Mecanismo de encaminamiento:

- Aritmético: Determinar el siguiente tramo de la ruta mediante una op aritmética simple.
- Selección de la ruta en origen: La información de ruta acompaña al paquete. Problema: Incremento de longitud.
- Mediante tablas: Cada conmutador contiene una tabla de rutas. Cada paquete contiene un índice de acceso a la tabla de cada conmutador atravesado.

Propiedades del encaminamiento:

- Mínimo/ No mínimo.
- Determinista/ Adaptativo.

Problemas del encaminamiento

Interbloqueo(deadlock): Un paquete está a la espera de un evento que no puede ocurrir y una red bien diseñada debe estar libre de interbloqueos.

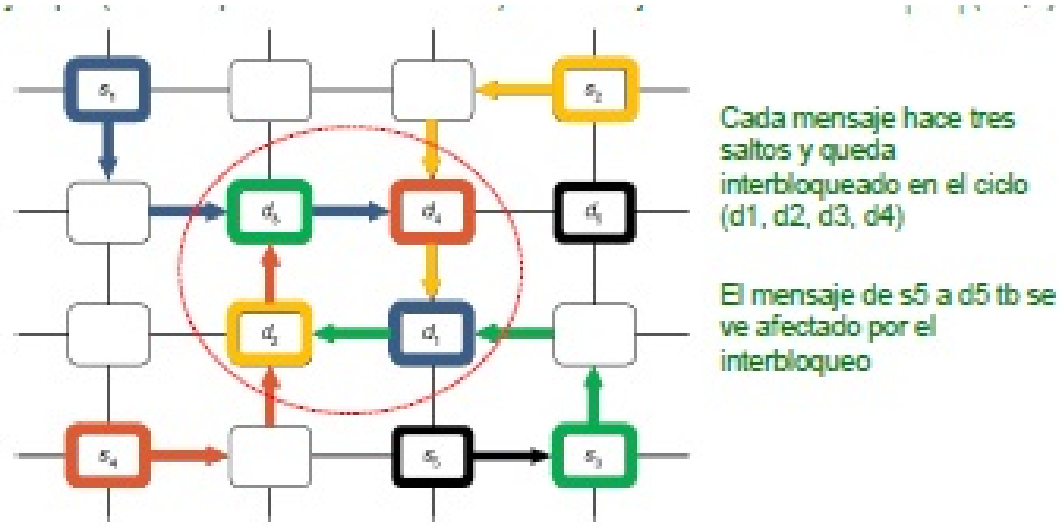


Figura 5.5: Interbloqueo en una malla 2D

Espera indefinida (starvation): Un paquete está a la espera de un evento que puede ocurrir, pero no ocurre. Puede deberse a una mala planificación del sistema (fairness).
Ciclos infinitos(livelock): Un paquete está recorriendo nodos sin llegar nunca a su destino. Sólo puede ocurrir con encaminamiento adaptativo no mínimo.

Mecanismos de encaminamiento

Orden dimensional:

- Los enlaces en una cierta dimensión no pueden ser utilizados por un paquete hasta que no ha recorrido todos los enlaces necesarios en todas las dimensiones precedentes.

Selección de ruta en origen:

- La cabecera del mensaje contiene la serie de puertos seleccionados.
- Utilizado y dividido en ruta: Cada conmutador usa el primer puntero y lo elimina de la lista.

Dirigido por tablas (distribuido):

- Cada conmutador contiene una tabla de rutado R.
- La cabecera del mensaje contiene un índice i, que determina la entrada de la tabla de rutado que específica el puerto de salida.
- Generalmente la tabla también proporciona el índice i', para el siguiente switch.

Interbloqueo: evitación

¿Cómo puede producirse?:

- Condiciones necesarias:
  - Recurso compartido.
  - Adquirido incrementalmente.
  - Sin mecanismo de expulsión.
  - Los agentes en espera forman una cola circular.
- Canales como recurso compartido adquirido incrementalmente:
  - Buffer fuente unido a buffer destino.
  - Canales forman una ruta.

¿Cómo puede evitarse?:

- Restringiendo cómo se adquieren los canales.
- Apartando recursos adicionales: Canales virtuales.

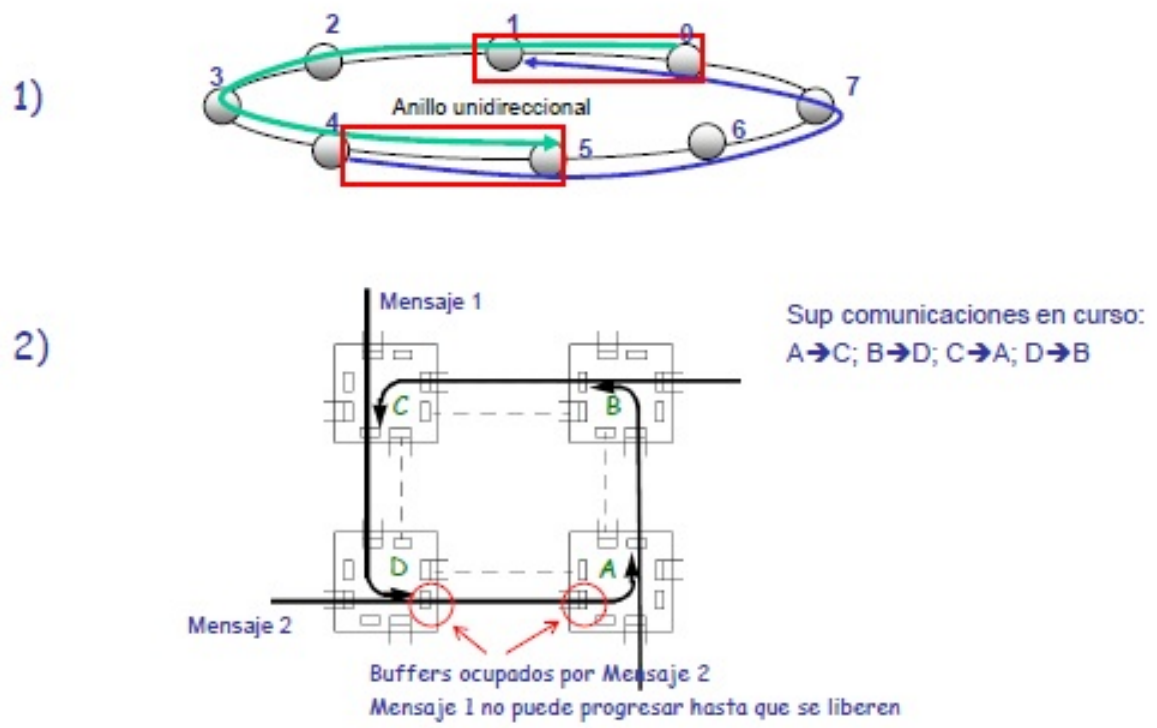


Figura 5.6: Ejemplos de situaciones de interbloqueo

Canales virtuales

Definición

Mecanismo básico para construir redes libres de interbloqueos con mecanismo worm-hole.

Técnica: Asociar con cada enlace físico más de un buffer de entrada, que se comportan cómo canales adicionales.

- Proporcionan una vía de escape en situaciones que provocarían interbloqueo.
- No aumenta el nº de enlaces en la red.
- No aumenta el tamaño del crossbar interno del conmutador: Los buffers asociados a cada canal físico, están conectados al crossbar interno mediante mux.

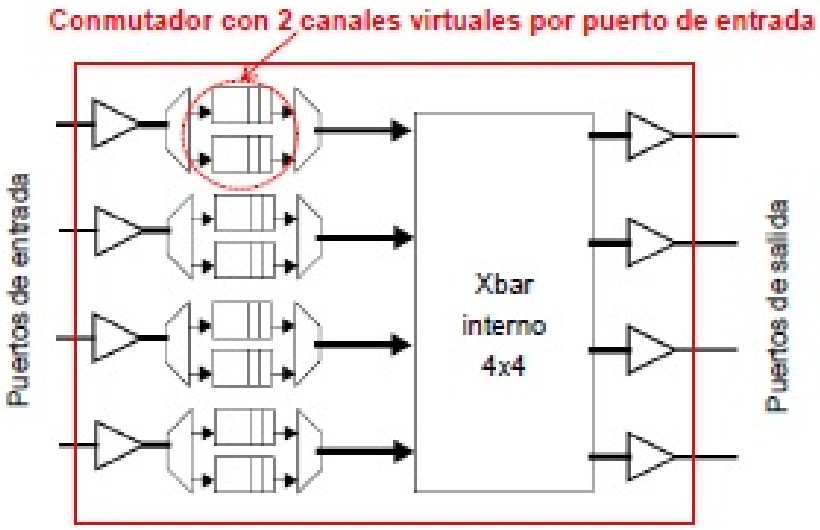


Figura 5.7: Canales virtuales

5.4. Explicación de ejercicios resueltos

Ejercicio 3. Para cada uno de las cadenas de referencias a memoria que se muestran a continuación, construye una tabla en la que se indique el estado del bloque referenciado en cada controlador de cache en cada instante y el tiempo necesario para cada referencia suponiendo un multiprocesador de bus compartido que utiliza el protocolo MESI explicado en clase.

Cadena 1: r1 w1 r1 w1 r2 w2 r2 w2 r3 w3 r3 w3
Cadena 2: r1 r2 r3 w1 w2 w3 r1 r2 r3 w1
Cadena 3: r1 r2 r3 r3 w1 w1 w1 w2 w3

Todas las referencias en las cadenas son a la misma posición de memoria, r y w indican lectura y escritura, respectivamente, y el dígito se refiere al procesador que hace referencia. Asume que todas las caches están inicialmente vacías, y usa el siguiente modelo de coste:

- Un acierto de cache en lectura/escritura (sin acceso al bus): 1 ciclo.

- Operaciones que requieren una transacción del bus simple: 60 ciclos.
- Operaciones que requieren una transferencia de un bloqueo de cache: 90 ciclos.

Solución ejercicio 3:
En cada tabla se referencia el estado del bloque referenciado en cada procesador

CADENA	r1	w1	r1	w1	r2	w2	r2	w2	r3	w3	r3	w3
P1	E	M	M	M	S	I	I	I	I	I	I	I
P2	I	I	I	I	S	M	M	M	S	I	I	I
P3	I	I	I	I	I	I	I	I	S	M	M	M
CICLOS	90	1	1	1	90	60	1	1	90	60	1	1

CADENA	r1	w1	r1	w1	r2	w2	r2	w2	r3	w3	r3	w3
P1	E	M	M	M	S	I	I	I	I	I	I	I
P2	I	I	I	I	S	M	M	M	S	I	I	I
P3	I	I	I	I	I	I	I	I	S	M	M	M
CICLOS	90	1	1	1	90	60	1	1	90	60	1	1

CADENA	r1	r2	r3	r3	w1	w1	w1	w1	w2	w3
P1	E	S	S	S	M	M	M	M	I	I
P2	I	S	S	S	I	I	I	I	M	I
P3	I	I	S	S	I	I	I	I	I	M
CICLOS	90	90	90	1	60	1	1	1	90	90

Ejercicio 4. Repetir el problema anterior para la cadena de referencias

r2 w2 r2 w1 r1 r3 w3

pero suponiendo los siguientes protocolos:

A. MSI.

B. MESI.

Indica cuál es la principal ventaja que aporta el protocolo MESI.

Solución ejercicio 4:
Solución apartado A:

CADENA	r2	w2	r2	w1	r1	r3	w3
P1	I	I	I	M	M	S	I
P2	S	M	M	I	I	I	I
P3	I	I	I	I	I	S	M
CICLOS	90	60	1	90	1	90	60

Solución apartado B:

CADENA	r2	w2	r2	w1	r1	r3	w3
P1	I	I	I	M	M	S	I
P2	E	M	M	I	I	I	I
P3	I	I	I	I	I	S	M
CICLOS	90	1	1	90	1	90	60

La principal ventaja que aporta el protocolo MESI se observa en la segunda columna, ya que al pasar del estado E al M no es necesario ordenar ninguna transacción de bus, con el consiguiente ahorro de tiempo.

Ejercicio 5. Para la cadena de referencias a memoria que se indica en la primera fila de la Tabla, calcula el coste de ejecutarla en un multiprocesador de bus compartido que soporta el protocolo **MESI** y usa **BusUpgrd**. Indica el resultado en forma de una tabla que muestre también la evolución en el tiempo del contenido de los marcos de bloque de cache, su estado y la transacción que se realiza en el bus; donde cada columna es un acceso y cada fila un marco de bloque.
La memoria principal contiene cuatro bloques y cada una de las caches dos marcos de bloque. Las referencias a memoria indican el bloque de memoria principal al que acceden; por ejemplo P1 lee 00 quiere decir que el procesador 1 lee el bloque de memoria 00.
Asume que todas las caches están inicialmente vacías, y usa el siguiente modelo de coste:

- Un acierto de cache el lectura/escritura: 1 ciclo.
- Fallos que requieren una transacción del bus simple: 60 ciclos.
- Fallos que requieren una transferencia de un bloque de cache: 90 ciclos.

Suponer que se utiliza en todos los casos caches con asignación en escritura (write allocate), post escritura, emplazamiento asociativo, política de reemplazo: se reemplaza el bloque que lleva más tiempo sin ser usado. A modo de ejemplo se han completado las dos primeras columnas.

CADENA	P1 lee 00	P2 lee 00	P1 lee 00	P2 esc 01	P1 lee 01	P2 esc 00	P1 esc 01	P1 lee 11
Bloque0 P1	00 E	00 S						
Bloque1 P1	I	I						
Bloque0 P2	I	00 S						
Bloque1 P2	I	I						
Ciclos	90	90						

Solución ejercicio 5:

CADENA	P1 lee 00	P2 lee 00	P1 lee 00	P2 esc 01	P1 lee 01	P2 esc 00	P1 esc 01	P1 lee 11
Bloque0 P1	00 E	00 S	00 S	00 S	00 S	I	I	11 E
Bloque1 P1	I	I	I	I	01 S	01 S	01 M	01 M
Bloque0 P2	I	00 S	00 S	00 S	00 S	00 M	00 M	00 M
Bloque1 P2	I	I	I	01 M	01 S	01 S	I	I
Ciclos	90	90	1	90	90	60	60	90

Ejercicio 6. Supongamos un protocolo de Directorio simple tipo MSI (información de directorio= bit de presencia por cada nodo+ bit de modificación) en un sistema de 1024 procesadores y bloques de memoria de 64 bytes.

A. Describir brevemente (a modo de algoritmo si es posible) las operaciones que se realizan en el caso de Fallo de lectura a un bloque en estado compartido.

B. ¿Cuál es la sobrecarga de memoria asociada al directorio?

Solución ejercicio 6:
Solución apartado A:
Supongamos como el nodo peticionario es el nodo i.

- Como el estado es S, entonces la Mp contiene una copia válida del bloque. Por tanto el nodo i hace una petición al nodo Home del bloque solicitado.
- El nodo Home envía el bloque al nodo i.
- El nodo Home actualiza el directorio para el bloque solicitado: Presencia[i]=1.

Solución apartado B:
Por cada bloque de memoria 1024 bits de presencia+ 1 bit sucio=1024.
Supongamos que el tamaño de la memoria es M bytes=M*8 bits.

$$\text{N}^{\text{o}} \text{ bloques mem} = M/64$$

$$\text{Tamaño del directorio}(M/64) * 1024 = M * 16\text{bits}$$

El tamaño del directorio duplica el tamaño de la memoria → La sobrecarga es del 200 %.

Ejercicio 7. Supongamos un protocolo de Directorio simple tipo MSI (información de directorio= bit de presencia por cada nodo+ bit de modificación) en un sistema de 32768 nodos (1 procesador por nodo) y bloques de memoria de 2048 bytes. Indicar la sobrecarga de memoria asociada al directorio en los siguientes casos:

A. Sin optimización.

B. Incremento de línea a 4096 bytes.

C. Reducción de la anchura del directorio usando 100 punteros por bloque de memoria.

D. Reducción de la altura del directorio mediante su organización como una cache donde solamente hay espacio para un 50 % de las líneas.

Solución ejercicio 7:
Notación:

- NP: nº de procesadores.
- NB: nº de bloques de memoria.
- TM: tamaño de la memoria.
- TL: tamaño de la línea.
- TD: tamaño del directorio.

Solución apartado A:

$$\text{TD(bits)} = \text{NP(bits)} \times \text{NB}$$

$$\text{NB} = \text{TM(bits)}/\text{TL(bits)}$$

Sustituimos:

$$\text{TD(bits)} = (\text{NP(bits)}/\text{TL(bits)}) \times \text{TM(bits)}$$

$$\text{TD(bits)} = (2^{15}/(2^{11} \times 2^{3})) \times \text{TM(bits)}$$

TD=2TM→ La sobrecarga del directorio es del 200 %.
Solución apartado B:
Con TL=2^{12}:

$$\text{TD(bits)} = (2^{15}/(2^{12} \times 2^{3})) \times \text{TM(bits)}$$

TD=TM→ La sobrecarga del directorio es del 100 %.
Solución apartado C:
100 punteros por bloque.

$$\text{TD(bits)} = 100 \times \text{Tamaño de un puntero} \times \text{NB}$$

Como hay 2^{15} procesadores → El tamaño de un puntero es de 15 bits.

$$\text{TD} = 100 \times 15 \times (\text{TM(bits)}/\text{TL(bits)}) \rightarrow \text{TL(bits)} = 2048 \times 8$$

$$\text{TD} = (1500/(2048 \times 8)) \times \text{TM} = 0{,}0916\text{TM} \rightarrow \text{La sobrecarga e del} \quad 9{,}16\,\%$$

Solución apartado D:
Dejamos sitio en el Directorio para el 50 % de los bloques, por analogía con la ecuación 1:

$$\text{TD(bits)} = ((\text{NP(bits)}/\text{TL(bits)}) \times \text{TM(bits)})/2$$

$$\text{TD(bits)} = ((2^{15}/(2^{11} \times 2^{3})) \times \text{TM(bits)}) = \text{TM} \rightarrow \text{Sobrecarga del} \quad 100\,\%$$

Capítulo 6

Tema 6: Multiprocesadores: Coherencia, sincronización y consistencia

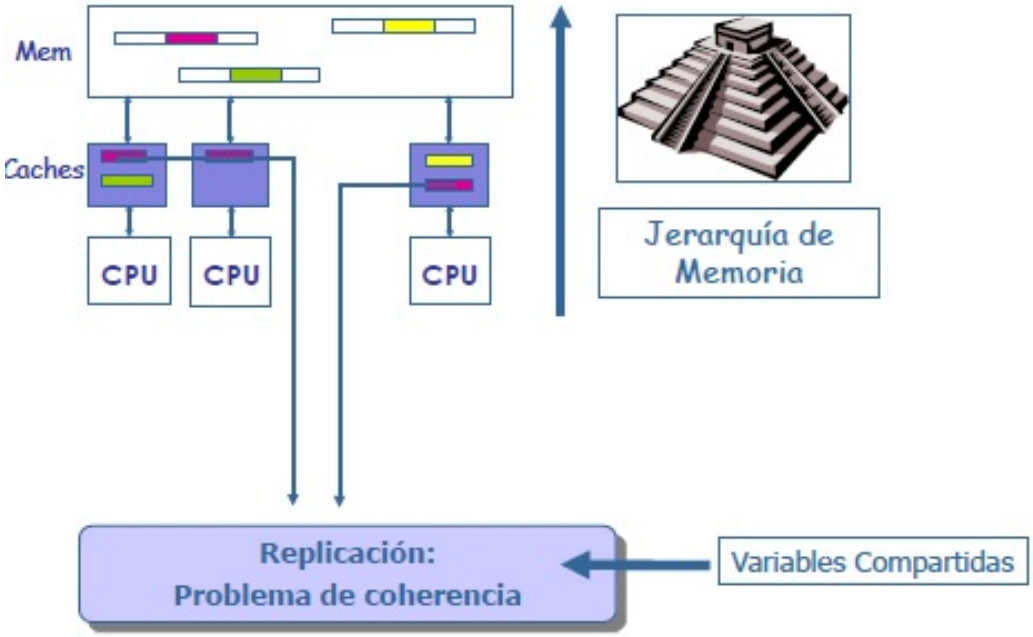
En este capítulo vamos a ver los siguientes conceptos.

- Jerarquía de memoria extendida.
- El problema de la coherencia cache.
- Protocolos de coherencia "Snoopy".
  - Protocolo Snoopy de dos estados.
  - Protocolo MSI: Invalidación de tres estados.
  - Protocolo MESI: Invalidación de cuatro estados.
- Coherencia cache basada en directorio.
  - Directorio plano basado en memoria.
  - Directorio plano basado en cache.
- Sincronización.
- Consistencia de memoria.

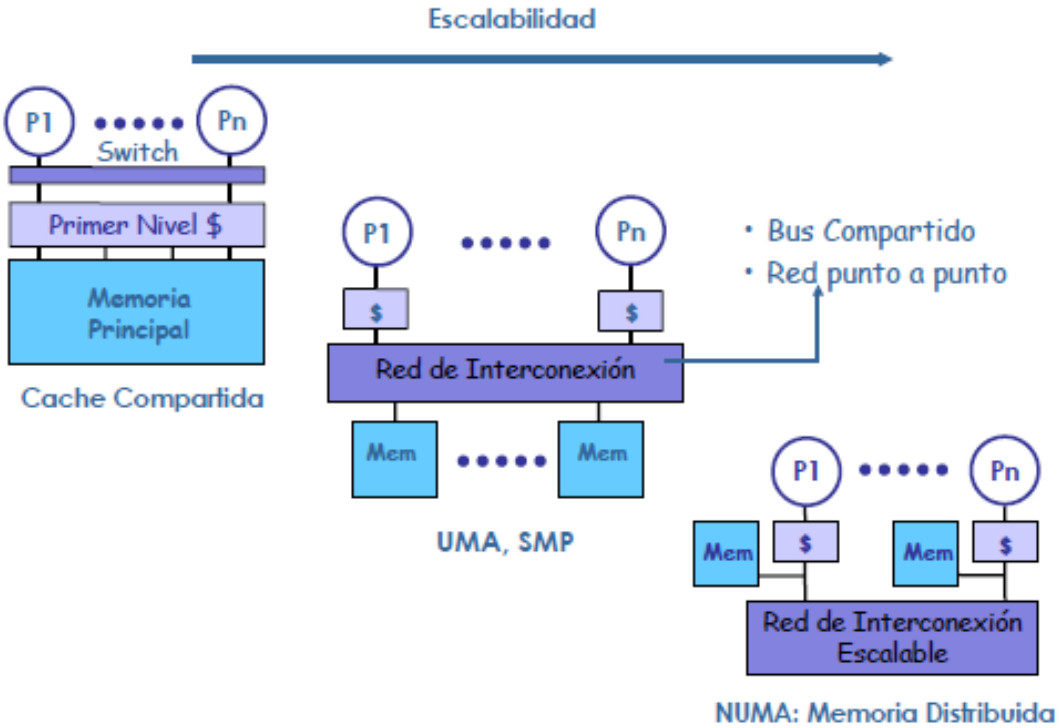
6.1. Jerarquía de memoria extendida

Las caches privadas permiten:

- Ocultar la latencia de los accesos.
- Reducir la contención.

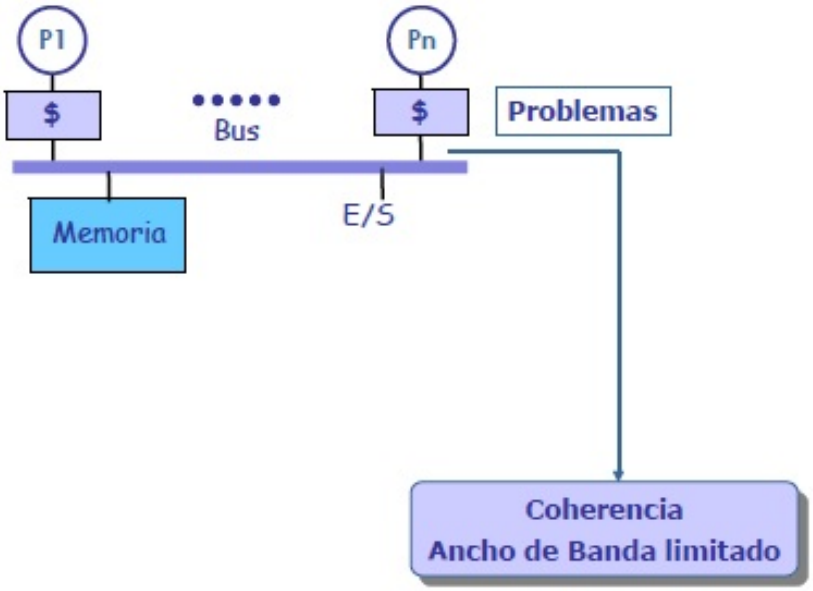


6.1.1. Alternativas de implementación



Una alternativa ampliamente usada: SMP basado en Bus Compartido.

- Servidores departamentales.
- Estaciones de trabajo.
- Bloques de construcción básicos sistemas de gran escala.
- Soporte en microprocesadores de propósito general.



6.1.2. Modelo intituivo de memoria

Programa secuencial: Las posiciones de memoria se utilizan para "comunicar valores" entre distintos puntos del programa. Cuando se lee una posición se devuelve el último valor escrito en ella. **Espacio de direcciones compartido en sistemas con un único procesador (Multiprogramación)**:

- Cuando se lee una posición se devuelve el último valor escrito en ella, independientemente del proceso que realizó la última escritura sobre dicha posición.
- Las caches no interfieren con el uso de múltiples procesos en un procesador.

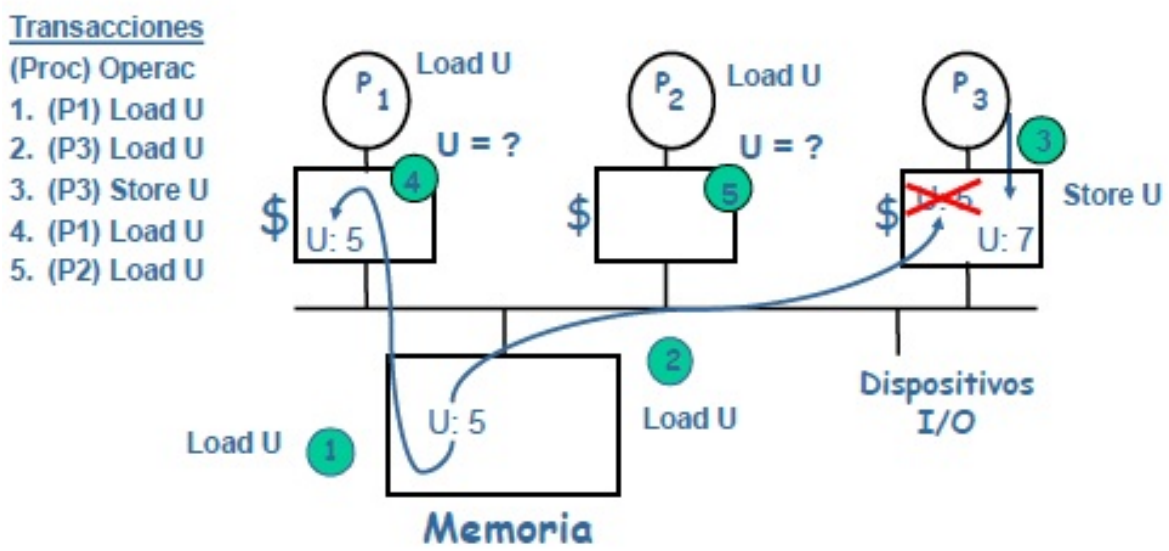
Multiprocesadores de Memoria Compartida:

- **Objetivo**: El resultado de ejecutar un programa qeu use varios procesos sea el mismo independietemente de si los procesos se ejecutan en paralelo o de forma entrelazada.
- Problema: Cuando hay más procesos que acceden a la memoria a través de caches diferentes existe el peligro de que vean valores inconsistentes.

6.2. Problemas de la coherencia cache

Protocolos de coherencia cache

Siguen la pista del estado de cualquier bloque de datos compartido para evitar incoherencias.



6.2.1. Soluciones de "Grano Grueso"

Uniprocesadores:

- Dispositivo DMA escribe en memoria: El procesador puede seguir viendo valores antiguos en la cache.
- Dispositivo DMA lee memoria: El DMA puede leer un valor antiguo en el caso de que se utilice write-back.

Operaciones E/S son mucho menos frecuentes que las operaciones de acceso a memoria:

- Evitar usar la cache: Los segmentos de memoria involucrados en operaciones de E/S se marcan como No-Cacheables o bien se utilizan operaciones load/store no-cacheables.
- Sacar de la cache antes de E/S: Las páginas de memoria involucradas en cualquier operación de E/S son eliminadas de la cache previamente por el SO.
- Usar la cache para E/S: El contenido de la cache puede corromperse.

Multiprocesadores:

- La escritura o lectura de variables compartidas es un evento frecuente.
- No es práctico deshabilitar la cache en datos compartidos o invocar al SO en cada referencia a una variable compartida.

Todos los μprocesadores actuales proporcionan mecanismos para soportar la coherencia.

6.2.2. Coherencia

Un sistema de memoria es coherente

Una operación de lectura retorna siempre el último valor que fue escrito en la posición de memoria correspondiente, independientemente del procesador que efectúa la lectura o escritura.

El sistema debe ser **coherente** y **consistente**.

- La lectura del procesador P a la posición X de memoria, que sigue a una escritura P en X, sin ninguna otra escritura debe retornar el valor escrito por P.
- Una lectura por un procesador a la posición X que sigue a una escritura de otro procesador en X, retorna el valor escrito si no hay ninguna otra escritura y están suficientemente separadas en el tiempo.
- **Señalización de escrituras**: Todas las escrituras a la misma posición deben verse en el orden correcto.

Políticas (o protocolos) para mantener la coherencia

Invalidación/ Coherencia Dinámica:

- **Múltiples lectores, un solo escritor**: Al escribir un bloque se invalidan todas las otras copias.
- Escrituras consecutivas al mismo bloque efectuadas desde el mismo procesador se realizan localmente.
- Si hay un fallo de lectura: Con write-through o con write-back.

Actualización en escritura:

- Al escribir en un bloque se actualizan todas las copias.
- Write-through.
- Escrituras consecutivas a la misma pantalla requiere múltiples actualizaciones.
- Si hay un fallo en lectura: Se busca en la memoria y siempre esta actualizado.

Los protocolos usados para la invalidación o actualización dependen de la red de interconexión utilizada.

- Si permite broadcast → Protocolo Snoopy(buses): Observación del bus. Cada controlador de cache está observando los eventos que suceden en el bus y toma decisiones en consecuencia. Un controlador de cache envía eventos al bus en función de las peticiones de acceeso a memoria que recibe su procesador → Envía órdenes de invalidación o actualización de forma simultánea a todos los controladores.
- Si no permite broadcast → Protocolos basados en Directorio: Se utiliza un directorio con una entrada por cada bloque en la que se indican en qué caches existe copia y en qué estado.

Protocolos Snoopy

Orden arbritaje: Las transacciones del Bus son visibles en el mismo orden por todos los controladores de cache.

Figura 6.1: Observar/ Espiar al Bus

Extender la funcionalidad del controlador de cache:

- El estado de cada bloque mapeado en la cache sigue estando en el directorio.
- Los cambios de estado son provocados por:
  - Operaciones LD/ST del procesador.
  - Transacciones relevantes observadas en el bus.

Convenio de representación:

- Cada transición es disparada por la observación de un evento en el procesador o el bus.
- Una transición puede implicar que el controlador ejecute alguna acción sobre su cache, u ordene una cierta acción al bus.

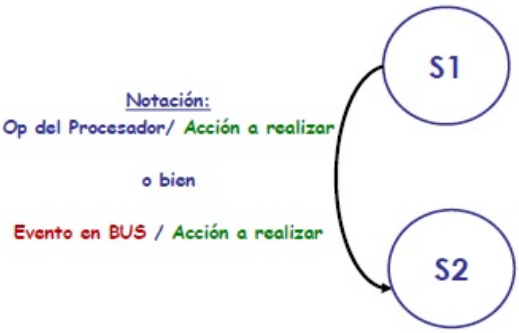
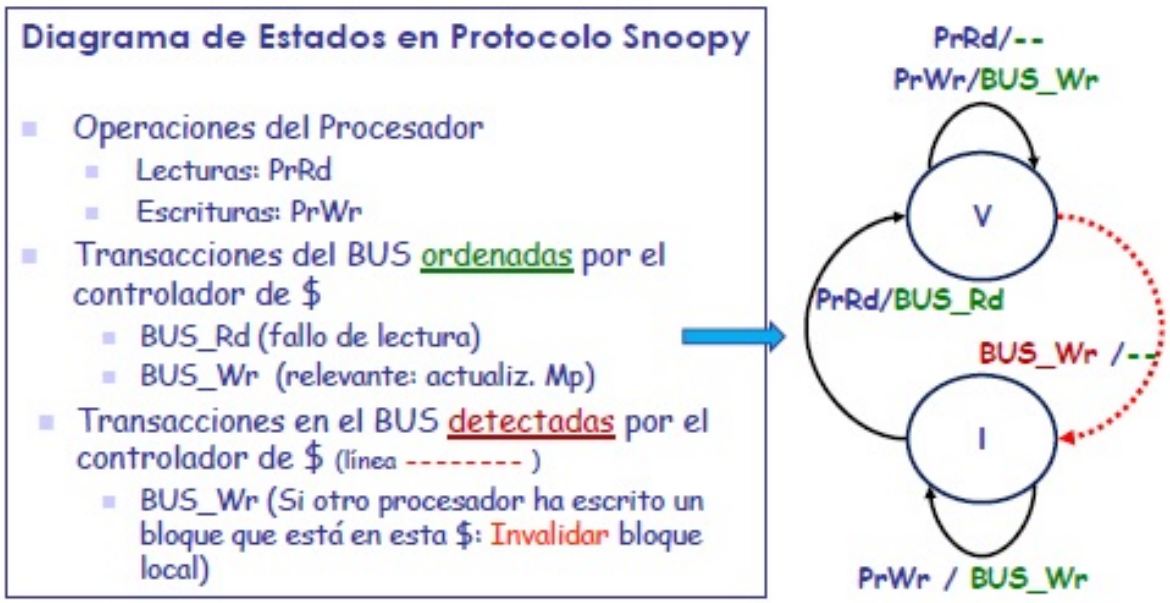


Figura 6.2: Diagrama de estados de bloque de cache

6.2.3. Protocolo Snoopy de 2 Estados

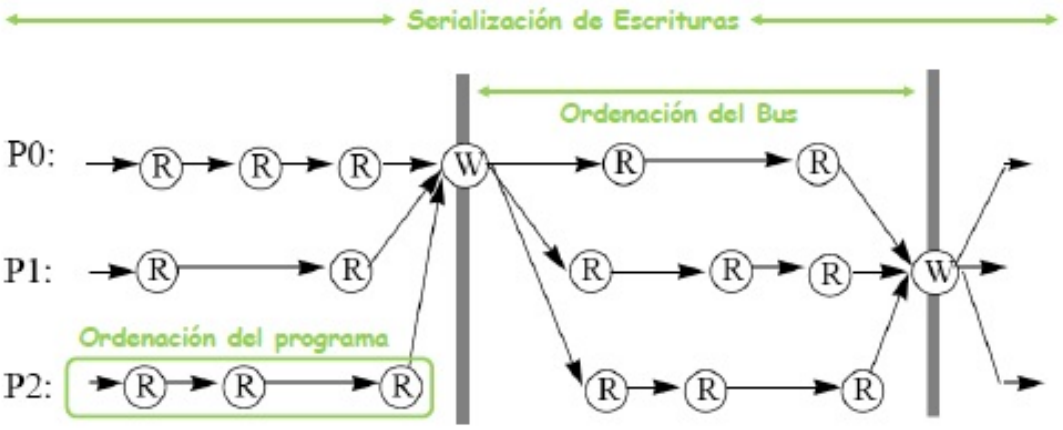


¿Es coherente el Protocolo Snoopy de 2 Estados?

- Escritura directa y 1 solo nivel de cache.
- Bus atómico.
- Operaciones atómicas.

Orden parcial de escrituras= Orden del bus.

- Propagación de escrituras: Las escrituras son visibles a todos los controladores.
- Serialización de Escrituras: El arbitraje de bus define el orden con el que se efectúan las escrituras a todas las posiciones de memoria.



Las transacciones del bus asociados con escrituras segmentan los flujos individuales de cada programa.
Entre dos transacciones de escritura el orden de las lecturas en los distintos procesadores no está restringido.
Entre escrituras, cada programa hace sus lecturas de acuerdo con el orden del programa.
Problema: Cada escritura hecha por cada procesador implica una transacción en el Bus → Consumo de Ancho de Banda.
Importante: Evitar escrituras por medio de política write-back.

6.2.4. Protocolo MSI: Invalidación de 3 estados (multiprocesador)

Tres estados para un bloque de memoria:

- Modificado: Exclusivo.
- Shared (Compartido): Válido.
- Inválido.

Estado M:

- Sólo hay 1 cache con copia válida.
- La exclusividad implica que la cache puede modificar el bloque sin notificárselo a nadie.

Estado S:

- El bloque está presente en la cache y no ha sido modificado.
- La memoria está actualizada.
- Otros caches adicionales pueden tener copia.

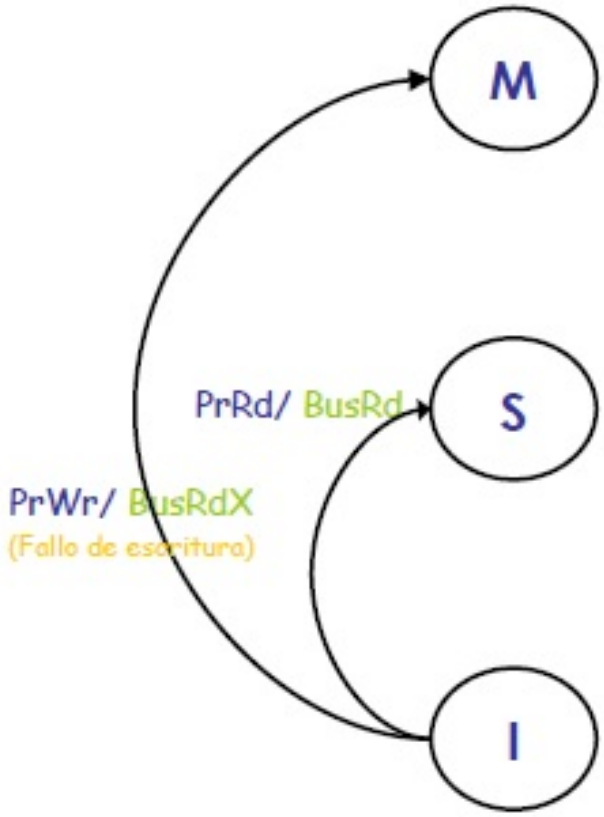


Figura 6.3: Transiciones desde el estado I

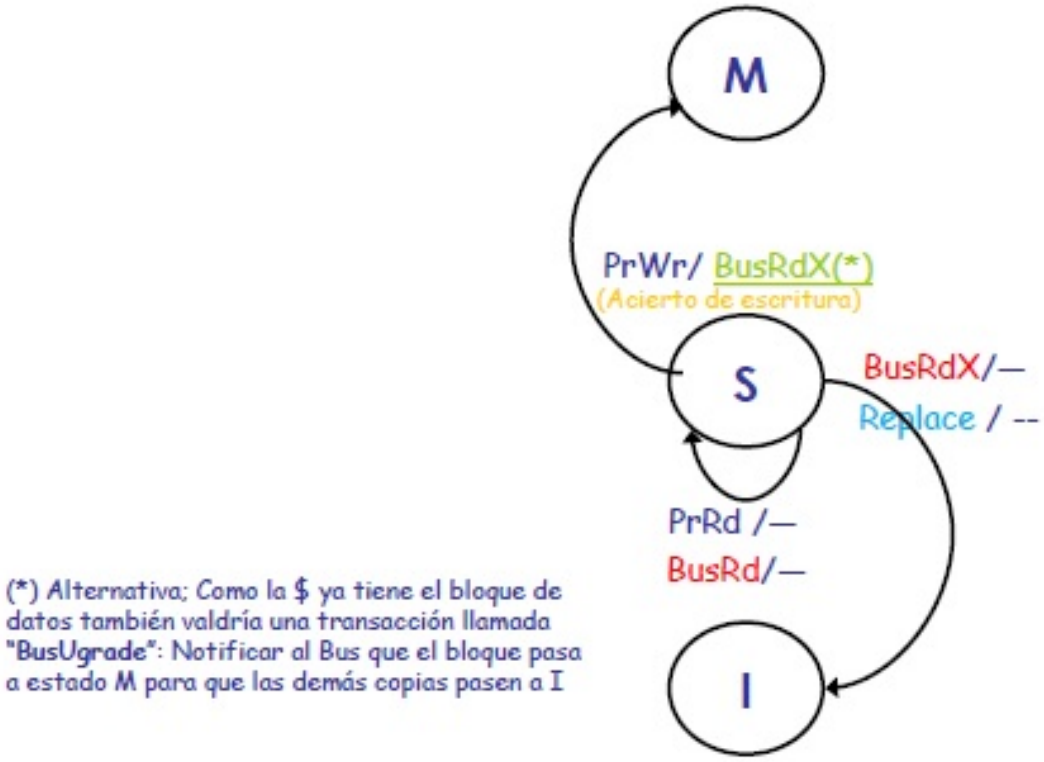


Figura 6.4: Transiciones desde el estado S

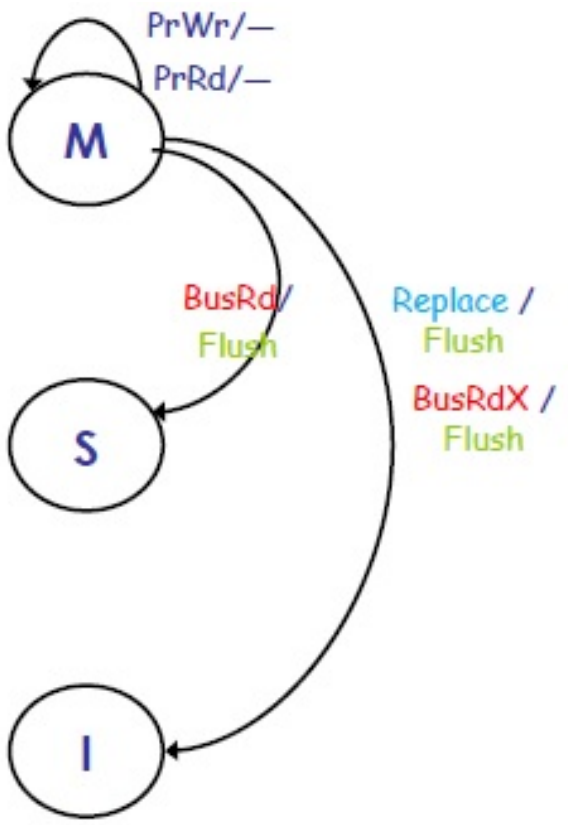


Figura 6.5: Transiciones desde el estado M

6.2.5. Implementación de protocolos Snoopy

Carreras entre escrituras:

- No puede escribirse en las cache hasta que se tiene el bus.
- Dos pasos:
  - Arbitrar el bus.
  - Poner el fallo sobre el bus y completar la operación.
- Bus de transacciones partidas: No son atómicas.

Los diferentes procesadores deben acceder al bus: datos y direcciones.
Los procesadores espían al bus: Si las direcciones coincide con algún tag invalidar o actualizar.
El bus serializa las escrituras. Solo un procesador realiza la operación con memoria.

6.2.6. Protocolo MESI: Invalidación de 4 estados

Tiene 4 estados:

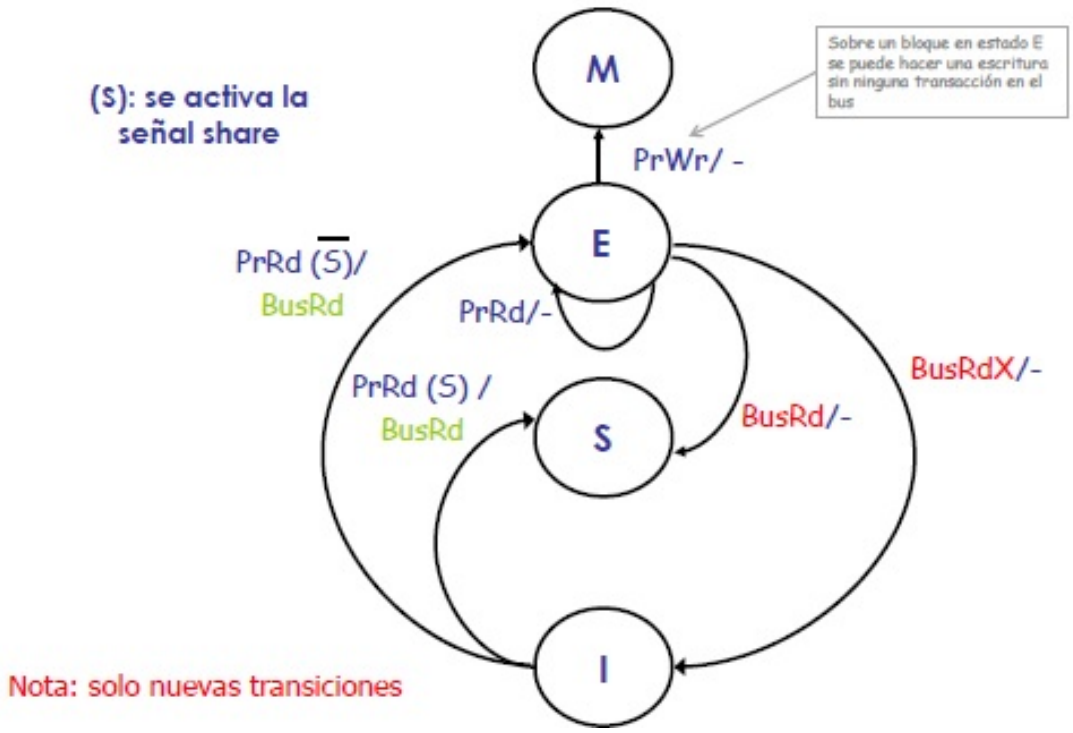
- M: Modificado.
- E: Exclusivo.

- S: Compartido.
- I: Inválido.

E: Exclusive Clean o Exclusive:

- Nivel intermedio entre compartido y modificado.
- Implica exclusividad: Puede pasarse a modificado sin realizar ninguna transacción.
- No implica pertenencia: La memoria tiene una copia válida. El controlador de cache no debe proporcionar el bloque en respuesta a una transacción de lectura de dicho bloque.

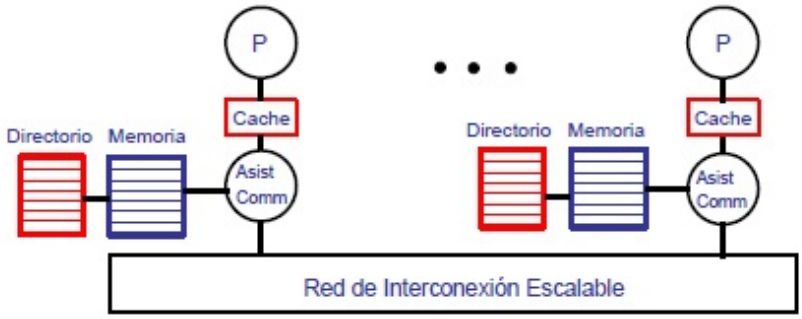
Señal S(Share): En reposo, sin ninguna cache.



6.3. Coherencia basada en directorio

Sistema de memoria distribuida:

- Una zona asociada a cada procesador. Cada procesador ve un espacio global de direcciones.
- Los accesos locales o remotos los maneja el Asistente de Comunicaciones del nodo.
- Problema: Coherencia de caches.
  - No hay mecanismo para observar las transacciones de los otros nodos.
  - Es demasiado costoso mandar información a todos los demás nodos.



Se puede utilizar un esquema similar a snoopy pero los cambios de estado son provocados por mensajes a través de la red.
Estados de un bloque en la cache de un procesador P:

- **M**: Exclusivo. La cache P tiene la única copia válida del bloque. La Mp no está actualizada.
- **S**: Compartida. La cache de P tiene una copia del bloque. Puede haber más copias en otros caches. La Mp está actualizada.
- **I**: Inválido. El bloque no está en la cache de P.

Permite determinar para cada bloque de memoria:

- Si el bloque no se encuentra en la cache de ningún procesador.
- Si existe una o varias copias del bloque sin modificar en los procesadores.
- Si el procesador tiene una copia modificada del bloque y cuál es el procesador.

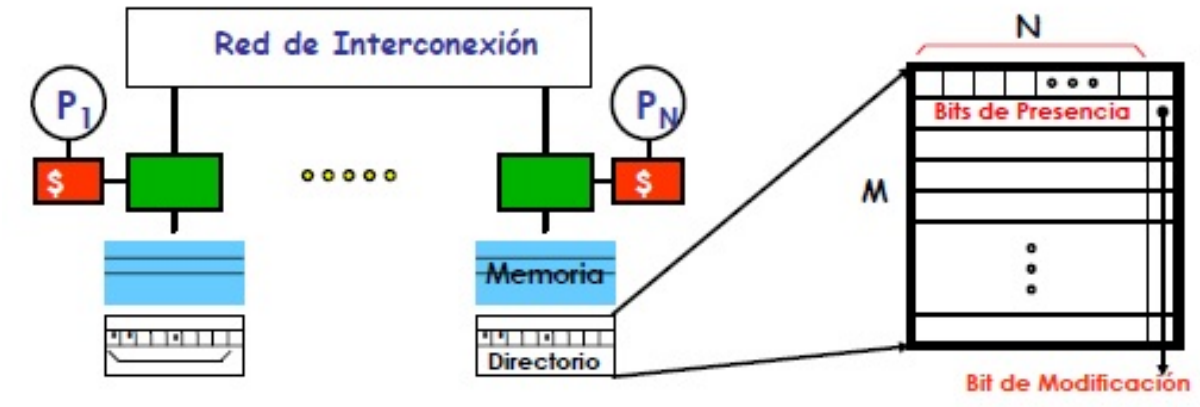
Tres tipos de procesadores implicados en una transacción:

- **Peticionario** (local): El procesador inicia la petición.
- **Home**: El procesador donde reside la zona de memoria que contiene la dirección solicitada.
- **Remoto**: Procesador cuya cache contiene copias válidas de la dirección solicitada en estado S o M.

Organización del directorio:

- Esquema Plano: El origen de la información de directorio para un bloque se encuentra en un lugar fijo, que viene determinado directamente por la dirección del bloque. Ante un fallo de cache se envía una transacción de red directamente al nodo origen para consultar directorio.
- Esquema Plano Basado en Memoria: La información de directorio acerca de un determinado bloque esta almacenado en el nodo origen de dicho bloque.
- Esquema Plano Basado en Cache: La información de directorio acerca de un determinado bloque no está totalmente almacenada en el nodo origen sino distribuido entre las propias copias.

6.3.1. Directorio plano basado en memoria



N nodos, M bloques de memoria por nodo.
Estado: uno o más bits.

- No es necesario saber estado concreto del protocolo.
- Más simple: un único bit.

Funcionamiento

Definiciones previas:

- **Nodo Peticionario (local)**: Contiene el procesador que emite una petición del bloque.
- **Nodo Origen (home)**: Nodo en cuya porción de memoria principal se encuentra el bloque.
- **Nodo Sucio (dirty)**: Tiene una copia del bloque en su cache en estado modificado.
- **Nodo Exclusivo (exclusive)**: Tiene una copia del bloque en estado exclusivo.
- **Nodo Propietario (owner)**: Nodo que mantiene en la actualidad la copia válida de un bloque y debe suplir los datos cuando sean necesarios.

Rendimiento en las Escrituras:

- Número de mensajes (BW): Proporcional al número de nodos que comparten el bloque.
- Número de mensajes en el camino crítico.

Sobrecarga de almacenamiento:

$$\text{M bloques de memoria} = \text{Sobrecarga promocional a Nproc} * M$$

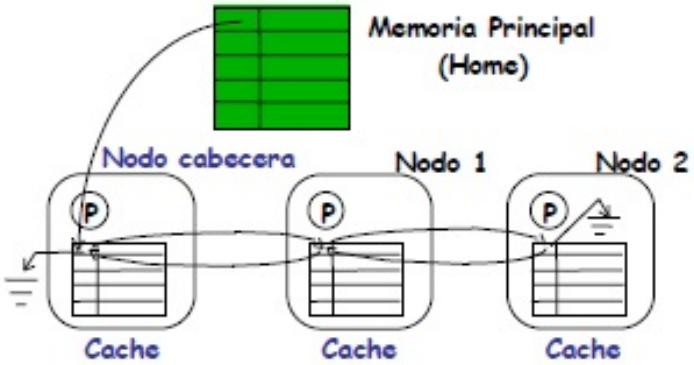
- Con 64 nodos: Sobrecarga del directorio 12.5 %.
- Con 256 nodos: Sobrecarga del directorio 50 %.

- Con 1024 nodos: Sobrecarga del directorio 200 %.

Soluciones:

- Utilizar nodos multiprocesador.
- En lugar de almacenar 1 bit por nodo cada entrada del directorio, se almacenan un determinado número de punteros a los nodos que comparten dicho bloque.
- Organizar el directorio como una cache, en lugar de tener una entrada por cada bloque.

6.3.2. Directorio plano basado en cache



Memoria principal origen de cada bloque:

- La entrada del directorio en el nodo origen tiene un puntero al primero de los nodos que tiene copia de dicho bloque.
- Bits estado.

El resto de los nodos se encuentra unido mediantte una lista entrelanzada distribuida.
Fallo de lectura:

- El nodo peticionario envía una petición al nodo origen de ese bloque para determinar la identidad del nodo cabecera.
- El nodo origen responde con la identidad del nodo cabecera.
- El nodo peticionario envía solicitud al nodo cabecera para que le inserte a la cabeza de la lista. Incrementa latencia.
- Los datos son enviados por nodo origen is tiene copia o por el nodo cabecera en caso contrario.

Eliminación (Reemplazamiento) e inserción en la lista: Añade bastante complejidad al protocolo.
Escritura:

- El nodo peticionario puede encontrarse ya en la lista de nodos que comparten el bloque.
- Se recorre el resto de la lista para invalidar las sucesivas copias. Las acuse de recibo de las invalidaciones se envían al nodo que realiza escritura.

Rendimiento de Escritura:

- Número de mensajes (BW): Proporcional al número de nodos que comparten el bloque, pero están distribuidos.
- Número de mensajes en el camino crítico (latencia): También es proporcional al número de nodos que comparten el bloque.

Ventajas:

- Menor sobrecarga de directorio:
  - Punteros a nodo cabecera: Proporcional al número de bloques de memoria de la máquina.
  - Punteros siguiente y anterior: Proporcional al número de bloques de cache en la máquina.
- El trabajo realizado por los controladores para enviar las invalidaciones no está centralizado en el nodo origen sino distribuido entre los nodos que comparten el bloque.
- La lista enlazada guarda el orden en el que se han realizado los accesos.

6.4. Sincronización

6.4.1. Introducción

Importante

Necesidad de garantizar acceso seguro de un proceso a variables compartidas.

Problemas: Exclusión mutua, secciones críticas.
Aspectos revelantes:

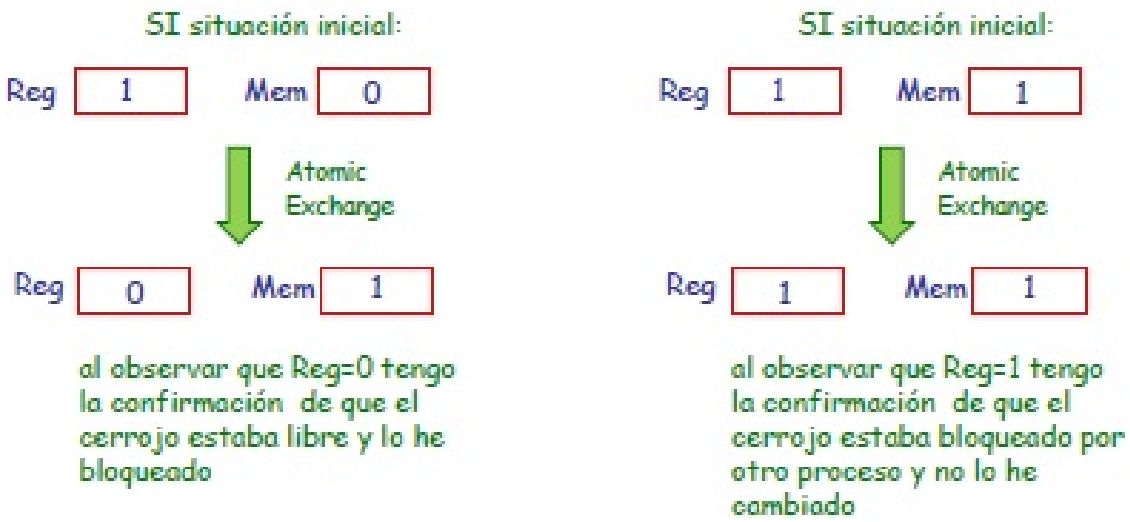
- hw básica: Una instrucción no interrumpible que lee y actualiza una cierta posición de memoria.
- Las operaciones de sincronización a nivel de usuario se construyen usando esta primitiva hw.
- Sistemas MPP: Las operaciones de sincronización pueden ser el cuello de botella del sistema.
  - Hay un conjunto de procesos compitiendo por leer-y-modificar el cerrojo.
  - Todos los procesos pueden estar corriendo simultáneamente en los diferentes procesadores.

- Cada procesador puede hacer las lecturas de cerrojo sobre su cache local, sin generar tráfico en el bus.
- Al escribir genera tráfico en el sistema de comunicaciones incluso aunque el intento de modificar el cerrojo sea fallido porque el cerrojo ya haya sido adquirido por otro procesador.
- El procesador puede permanecer en un bucle generando tráfico inútil en el bus hasta que logra adquirir el cerrojo.

6.4.2. Primitivas hw básicas

Convenio de interpretación de los valores del cerrojo:

- 0 → El cerrojo está libre.
- 1 → El cerrojo está bloqueado y no es accesible.



Importante

Si dos procesadores intentaran ejecutar el Exchange a la vez, el mecanismo de serialización de escrituras garantiza que uno de los dos obtendrá un 0 en el Reg y el otro obtendrá un 1.

Otras primitivas:

- Test-and-set: Comprueba el valor de una variable y si está a cero, la pone a 1.
- Fetch-and-increment: Incrementa el valor de una variable en memoria.

Operaciones atómicas en multiprocesadores:

- Se requiere una lectura y una escritura en una sola instrucción no interrumpible.
- Complica la implementación de la coherencia: el hw no puede permitir otra op entre la lectura y la escritura, y además debe evitar antibloqueo.

Sincronización con un par de operaciones:

- **Load Linken (LL)**: Lee una dir de memoria, pero además guarda un registro especial la dirección leída. Si ocurre una interrupción o si el bloque de cache que contiene la dir es invalidado por una escritura, el link register se borra.
- **Store Conditional (SC)**: Almacena el contenido de un registro R en una dir de memoria, siempre que ésta coincida con la dir almacenada en el link register. En caso contrario no se hace el almacenamiento. Si el almacenamiento tuvo éxito devuelve R=1 y en caso contrario R=0.

Implementación de spin locks

Spin lock: Bucle en el que un proceso permanece intentando conseguir un cerrojo hasta que lo consigue.
Problema: Si varios procesadores → Aumento de tráfico.
Implementación simple sin coherencia cache: Mantener las variables cerrojo en memoria.
Sup la variable cerrojo está en M[0+(R1)]:

```
        DADDIU  R2, R0, #1
lockit: EXCH    R2, 0(R1)     ; Atomic exchange
        BNEZ    R2, lockit    ; Already locked?
```

- Los accesos a mem implican tráfico en el sistema de comunicaciones → Cuello de botella cuando varios procesadores compiten por el cerrojo.
- Si tuviéramos un sistema con coherencia cache, la situación no sería mucho mejor. Varios procesadores ejecutando EXCH → Fallo de escritura → Invalidaciones.

Implementación con coherencia cache:

- Hacer una consulta preliminar del cerrojo.
- Un procesador permanece leyendo su copia local mientras detecta que el cerrojo está bloqueado.
- Cuando lee en su cache que el cerrojo está libre, ejecuta un EXCH y compite con los demás procesadores que estén haciendo el mismo spin lock. Uno de ellos consigue el bloqueo del cerrojo.
- El ganador verá un 0 en el registro de la instr EXCH, los demás verán un 1.
- Evita invalidaciones inútiles cuando un procesador está ejecutando repetidamente el bucle de espera.

```
lockit: LD      R2, 0(R1)     ; Preliminary load of hock
        BNEZ    R2, lockit    ; Test 1: if not avaible →Spin
        DADDUI  R2, R0, #1    ; R2=1
        EXCH    R2, 0(R1)     ; M[0+(R1)]⟺R2
        BNEZ    R2, lockit    ; Test 2: if lock wasn't 0→ Spin
```

Cuando el procesador ganador libera el cerrojo, la carrera entre los perdedores empieza de nuevo.
Implementación usando LL+SC

```
lockit: LL      R2, 0(R1)    ; Load linked of lock (1)
        BNEZ    R2, lockit   ; Not available-spin (2)
        DADDUI  R2, R0, #1   ; R2=1
        SC      R2, 0(R1)    ; Store conditional
        BEQZ    R2, lockit   ; Branch if store fails (3)
```

- El LL no tiene por qué generar tráfico en el sistema de comunicaciones.
- Si R2≠0 → El cerrojo está ocupado.
- Si R2≠0 → El cerrojo estaba libre cuando se ejecutó el LL, pero el SC ha fallado. Ha habido una interrupción, ha habido otra escritura entre LL y SC → Seguir intentando.

6.5. Consistencia de memoria

¿Qué aporta?

Las escrituras sobre una posición se hacen visibles a los lectores en el mismo orden. No dice nada sobre el momento en que una escritura particular se hará visible para un lector concreto.

Para garantizar las dependencias se recurre a operaciones de sincronización en las que suelen estar involucrados más de una posición de memoria.

6.5.1. Sincronización mediante Flag

P1	P2
/*Valor inicial de A y de flag=0*/	
A=1;	while (flag==0); **/*espera activa*/**
flag=1;	print A;

P1 almacena un nuevo valor de A, P2 debe leer el nuevo valor e imprimirlo.
El programador está asumiendo que el resultado de "print A" debe ser 1:

- La escritura de la variable A se hace visible al P2 antes de la escritura de flag.
- La lectura de flag hace que salgamos del bucle de espera se ha completado antes de la lectura de A.
- Esta ordenación entre los accesos de P1 y P2 no está garantizada por la coherencia.

La coherencia exige que el valor de A se haga visible a P2, pero no necesariamente antes que se haga visible el nuevo valor de flag.
La coherencia no dice nada sobre el orden con el que se hacen visibles las escrituras a

diferentes posiciones de memoria.
Esperamos más de un sistema de Memoria que simplemente "devolver el último valor escrito" para cada posición.
No es preciso que exista sincronización explícita de procesos para tener comportamiento que resulten extraños desde el punto de vista del programador.
Sea el siguiente código:

P1	P2
/*Valor inicial de A y de flag=0*/	
A=1;	while (flag==0); **/*espera activa*/**
flag=1;	print A;

- Los accesos a Memoria hechos por los procesos P1 y P2 son escrituras normales.
- A y B no se usan como variables de sincronización.
- La intuición nos hace esperar que si "print B" imprime un 2, entonces "print A" tiene que imprimir un 1.
- 2a y 2b leen diferentes direcciones de memoria. Y la coherencia no dice nada sobre el orden en que las escrituras de P1 sobre esas variables se hace visible a P2.
- Si P1 ejecuta 1a y continua con 1b, sin asegurarse antes de que P2 ha completado la invalidación de A en su cache local.

6.5.2. Consistencia de memoria

Es necesario un **modelo de ordenación** que los programadores puedan usar para razonar acerca de la corrección de los programas.
Modelo de Consistencia de la Memoria: Especifica las restricciones en el orden en el que las operaciones de memoria deben hacerse visibles a los procesadores. Incluye operaciones a las mismas o a diferentes posiciones de memoria → La Consistencia incluye a la Coherencia.
Contrato entre el programador y el diseño del sistema:

- Los programadores se basan en el modelo de consistencia para razonar acerca de los posibles resultados.
- Para el diseñador del compilador o del hardware el modelo impone limitaciones a posibles reordenaciones.

6.5.3. Consistencia secuencial

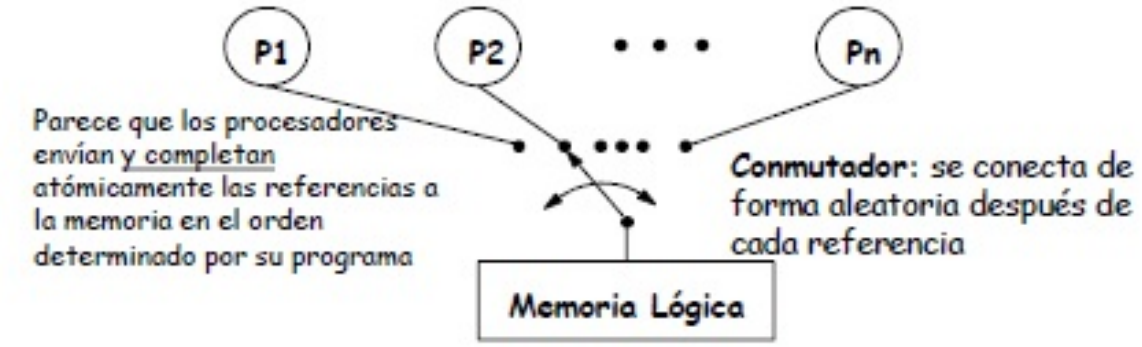
Objetivo

El resultado de ejecutar un programa que usa varios procesos sea el mismo independientemente de si los procesos se ejecutan en paralelo o de forma entrelazada.

Consistencia Secuencial:

- Un multiprocesador es secuencialmente consistente si el resultado de la ejecución de cualquier programa es el mismo que el que se obtendría si:
  - Las operaciones de todos los procesadores se realizan según algún orden secuencial → Cada entrelanzamiento define un orden total.
  - Las operaciones de cada procesador individual ocurren en esta secuencia en el orden especificado por el programa.

Abstracción para el programador del subsistema de memoria bajo el model de consistencia secuencial: Parece que los procesadores comparten una única memoria.



No es realmente importante en qué orden se emitan de hecho las operaciones a la memoria o incluso se completen.

Ejemplo

P1	P2
/*Valor inicial de A y B=0*/	
(la) A=1;	(2a) print B;
(lb) B=2;	(2b) print A;

Resultados posibles bajo **CS**:

- Orden 2a, 2b, 1a, 1b produce (A,B)=(0,0).
- Orden 1a, 2a, 1b, 2b produce (A,B)=(1,0).
- Orden 1a, 1b, 2a, 2b produce (A,B)=(1,2).

(A,B)=(0,2):

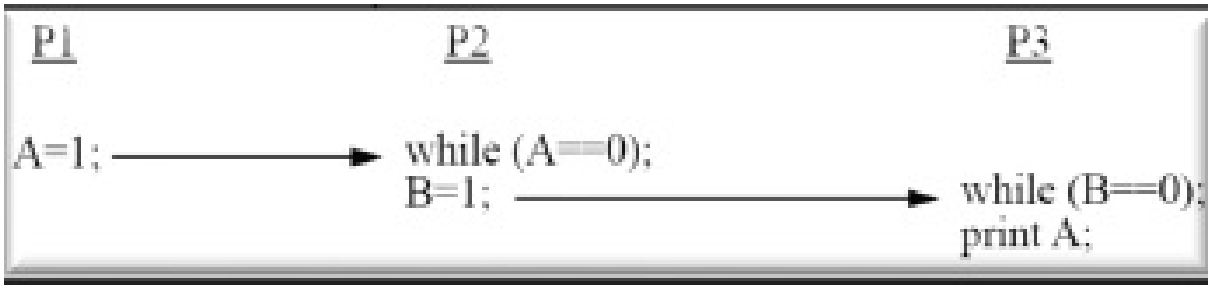
- El orden del programa implica 1a→ 1b y 2a→2b.
- A=0 implica 2b→1a, lo que implica 2a→1b.
- B=2 implica 1b→2a, contradicción.

Dos restricciones:

- **Orden del programa**: Las operaciones de memoria de un proceso deben hacerse visibles en el orden del programa.

- **Atromicidad**: Antes de emitirse la siguiente operación de memoria de acuerdo con el hipótetico orden total, es necesario que la operación en curso se haya completado respecto a todos los procesos.
- **Dificultad**: Atomicidad de escrituras.

Coherencia: Propagación+ serialización de escrituras.
Serialización: Todas las escrituras a una posición de memoria deben verse en el mismo orden por todos los procesadores.
Consistencia secuencial: Atomicidad de escrituras.
Atomicidad: Todas las escrituras deben verse en el mismo orden por todos los procesadores.
La atomicidad de las escrituras es la clave que garantiza que ninguna operación realizada por procesador P2 después de haber visto el nuevo valor producido por una escritura hecha por P1, es visible para otro procesador P3 hasta que P3 ha visto también la escritura hecha por P1.
Ejemplo: Si no se preserva la atomicidad de escrituras de puede violar la consistencia secuencial. Sup: Inicialmente A=B=0.



- P2 espera hasta que A vale 1; luego asigna el valor 1 a B.
- Como P3 espera hasta que B vale 1 y luego lee A y lo imprime, el valor impreso de A debería ser 1.
- Si la escritura de P1 en A no es atómico, P2 podría seguir su ejecución y asignar a B el valor 1 y P3 ver el nuevo valor de B, antes que P3 haya visto el nuevo valor de A. Por tanto P3 podría leer el nuevo valor de B, pero hallar el viejo valor de A en su cache e imprimirlo.

Condiciones Suficientes:

- **Condición Local**: Todo proceso emite las peticiones a memoria en el orden especificado por el programa.
- **Condición Local**: Después de emitir una operación de escritura, el proceso que la emitió espera antes de emitir su siguiente operación a que se haya completado dicha escritura.
- **Condición Glogal**: Después de emitir una operación de lectura, el proceso que la emitió espera antes de emitir su siguiente operación a que se haya completado:
  - Dicha lectura.
  - La escritura que generó el valor devuelto por la lectura.

Índice de figuras

1.1. Rendimiento del procesador 6
1.2. TOP500: Actores principales 7
1.3. Escalado de la Ley de Moore 8
1.4. Predicción Moore Intel 9
1.5. Esquema rendimiento 11
1.6. Esquema Ley de Amdahl 14
1.7. En resumen 15
1.8. Proporción de las fracciones tras obtener B 16
1.9. Nueva fracción 17
1.10. Esquema problema 4 24
1.11. Esquema problema 5 25

2.1. Ejemplo de LDE 34
2.2. Antidependencia WAR 34
2.3. Dependencia de salida WAW 34
2.4. S1 depende de P1: S2 depende de S2 35
2.5. Modelo computador 37
2.6. Ejemplo de programa 38
2.7. Ejemplo 40
2.8. Ejemplo 41
2.9. Ejecución SW pipelined: R1=1000 41
2.10. Paso 1 48
2.11. Paso 2A 48
2.12. Paso 2B 49
2.13. Paso 3 49
2.14. BTA 70
2.15. BHT 71
2.16. Predictor hibrido 74
2.17. Alpha 21264 75
2.18. Sin especulación 79
2.19. Con especulación 80

3.1. Decodifican 4 instrucciones en paralelo, que se pasan a la siguiente etapa 115
3.2. Ejemplo 117
3.3. Chequeos necesarios para renombrar un grupo de instrucciones 118
3.4. Se asume un fichero de registros unificado 125

4.1. Procesador matricial 154
4.2. Estructura básica de una arquitectura con registros vectoriales 155

4.3. Método de paso 0 157
4.4. Código DAXPY en CUDA 168
4.5. Ejemplo 1 171
4.6. Ejemplo 2 171
4.7. Ciclo de acceso 1 179
4.8. Ciclo de acceso 2 180
4.9. Ciclo de acceso 3 180
4.10. Ciclo de acceso 4 181
4.11. Ciclo de acceso 5 181
4.12. Ciclos de acceso 1-4 184
4.13. Ciclos de acceso 5-8 184

5.1. Diseño unificada 192
5.2. Diseño distribuida 193
5.3. Redes dinámicas Crossbar 199
5.4. Modelo general de MIN 199
5.5. Interbloqueo en una malla 2D 203
5.6. Ejemplos de situaciones de interbloqueo 204
5.7. Canales virtuales 205

6.1. Observar/ Espiar al Bus 216
6.2. Diagrama de estados de bloque de cache 216
6.3. Transiciones desde el estado I 219
6.4. Transiciones desde el estado S 219
6.5. Transiciones desde el estado M 220

Índice de cuadros

1.1. Para un cierto programa 12
1.2. Ejemplo para calcular rendimiento con máquina de referencia 13

2.1. Estaciones de Reserva 90
2.2. Registros 90
2.3. Load Buffers 90
2.4. Store Buffers 90
2.5. Estaciones de Reserva 91
2.6. Registros 91
2.7. Load Buffers 91
2.8. Store Buffers 91
2.9. Estaciones de Reserva 91
2.10. Registros 92
2.11. Load Buffers 92
2.12. Store Buffers 92
2.13. Estaciones de Reserva 92
2.14. Registros 92
2.15. Load Buffers 92
2.16. Store Buffers 93
2.17. Estaciones de Reserva 93
2.18. Registros 93
2.19. Load Buffers 93
2.20. Store Buffers 93
2.21. Estacionesde Reserva 94
2.22. Registros 94
2.23. Load Buffers 94
2.24. Store Buffers 94
2.25. Estaciones de Reserva 94
2.26. Registros 94
2.27. Load Buffers 95
2.28. Store Buffers 95
2.29. Estaciones de Reserva 95
2.30. Registros 95
2.31. Load Buffers 95
2.32. Store Buffers 95

PASCAL
Invierno 2018
Ult. actualización 31 de Diciembre de 2018

LaTeX lic.LPPL & powered by TEFLON CC-ZERO

Este documento esta realizado bajo licencia Creative Commons “CC0 1.0 Universal”.