Arquitectura de Procesadores

Alejandro Furfaro

Agosto de 2012

Agenda

1. Tecnología de Integracion
2. Arquitectura de Computadores
   - Instruction Set Architecture (ISA)
   - Organización y Hardware
3. El sistema de Memoria
   - Tecnologías de Memoria
   - Memorias y velocidad del Procesador
   - Memoria Cache
   - Coherencia de un cache
4. Conceptos Básicos
5. Conceptos avanzados
   - branch prediction
   - Superscalar
   - Scheduling Dinámico
   - Especulación por Hardware
   - Casos prácticos que mezclan todo lo visto
6. Embedded
   - Downsizing para consumir poco
   - CPU
   - Subsistema de memoria
   - Subsistema de Video
7. Media Applications
   - Modelo de ejeccución SIMD

Evolución

Figura: Vista microscópica de una célula del virus de la gripe

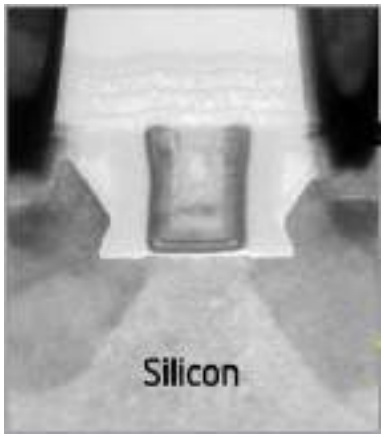


Figura: Vista en un Microscopio termoeléctrico de un CMOS de 32 nm high-K gate

En 2013 se espera tener lista la primer implementación con transistores MOS tri-gate en 22nm.

Evolución

Figura: Evolución de las tecnologías de integración.©Cortesía Intel

Tendencias tecnológicas

La tarea de un diseñador es permanente e inevitablemente moldeada por el rumbo de las tecnologías

1. La densidad de transistores por unidad de superficie aumenta 35 % por año en promedio. (Otra forma de la ley de Moore). Además el tamaño del die aumenta 10 a 20 % por año. Esto deriva en un crecimiento en la cantidad de transistores de entre 40 % y 55 % de un año a otro.
2. La velocidad de clock actualmente no crece a este ritmo. Parecería haber alcanzado un techo.
3. La capacidad de almacenamiento de las memorias DRAM crece a razón de un 40 % por año.
4. Los discos rígidos aumentan su capacidad 25 a 30 % por año. Su costo por bit de almacenamiento se mantiene entre 50 y 100 veces por debajo del costo de un bit de memoria DRAM

Scaling

- El proceso de un circuito integrado está caracterizado por un solo parámetro: *tamaño*, que es el mínimo valor en la dimensión de un transistor en las dimensiones *x* o *y*.
- El tamaño de un transistor en 1971 era de 10 microns. Actualmente es de 0,032 microns: 1250 veces mas pequeño...
- Los transistores se cuentan por mm^2 de silicio, de modo que podemos esperar una función de incremento del tipo cuadrática.
- Otro parámetro importante en un transistor es su rendimiento. Este aspecto es mas complejo. Al disminuir el tamaño en sentido vertical un transistor requiere reducir su tensión de alimentación. De otro modo su rendimiento decae o puede dañarse. Como no es posible cambiar la tensión de operación cada vez que se reduce la escala, la mejora en el rendimiento con cada avance en scaling no es cuadrática, sino que tiende a ser lineal.

Scaling

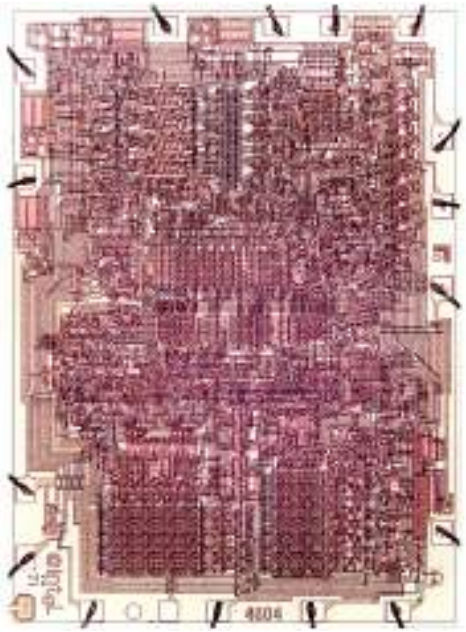


Figura: Procesador 4004, 2300 transistores 10 micrones. 1971 ©Cortesía Intel

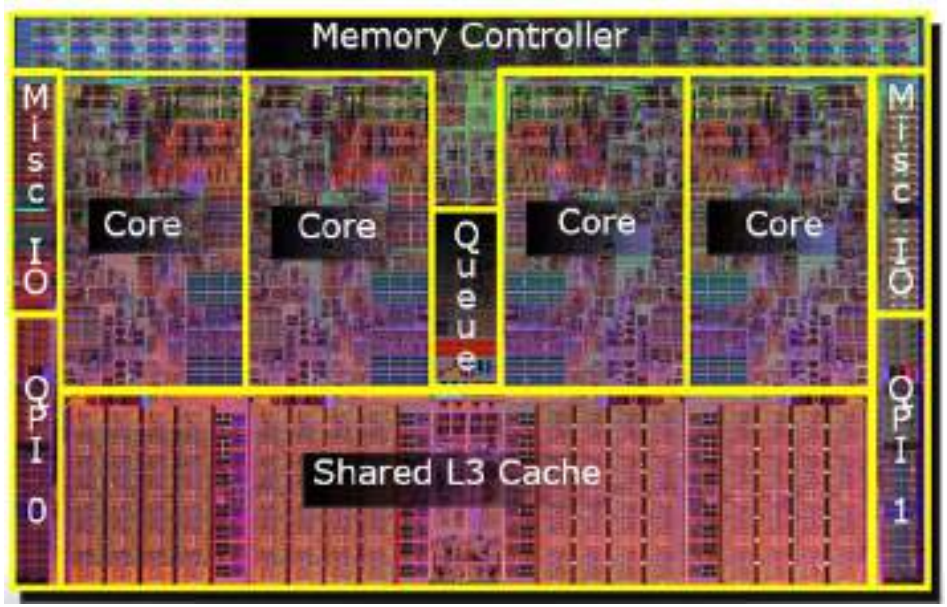


Figura: Procesador Core i7, 2.000.000.000 transistores 22 nm. 2012 ©Cortesía Intel

Scaling

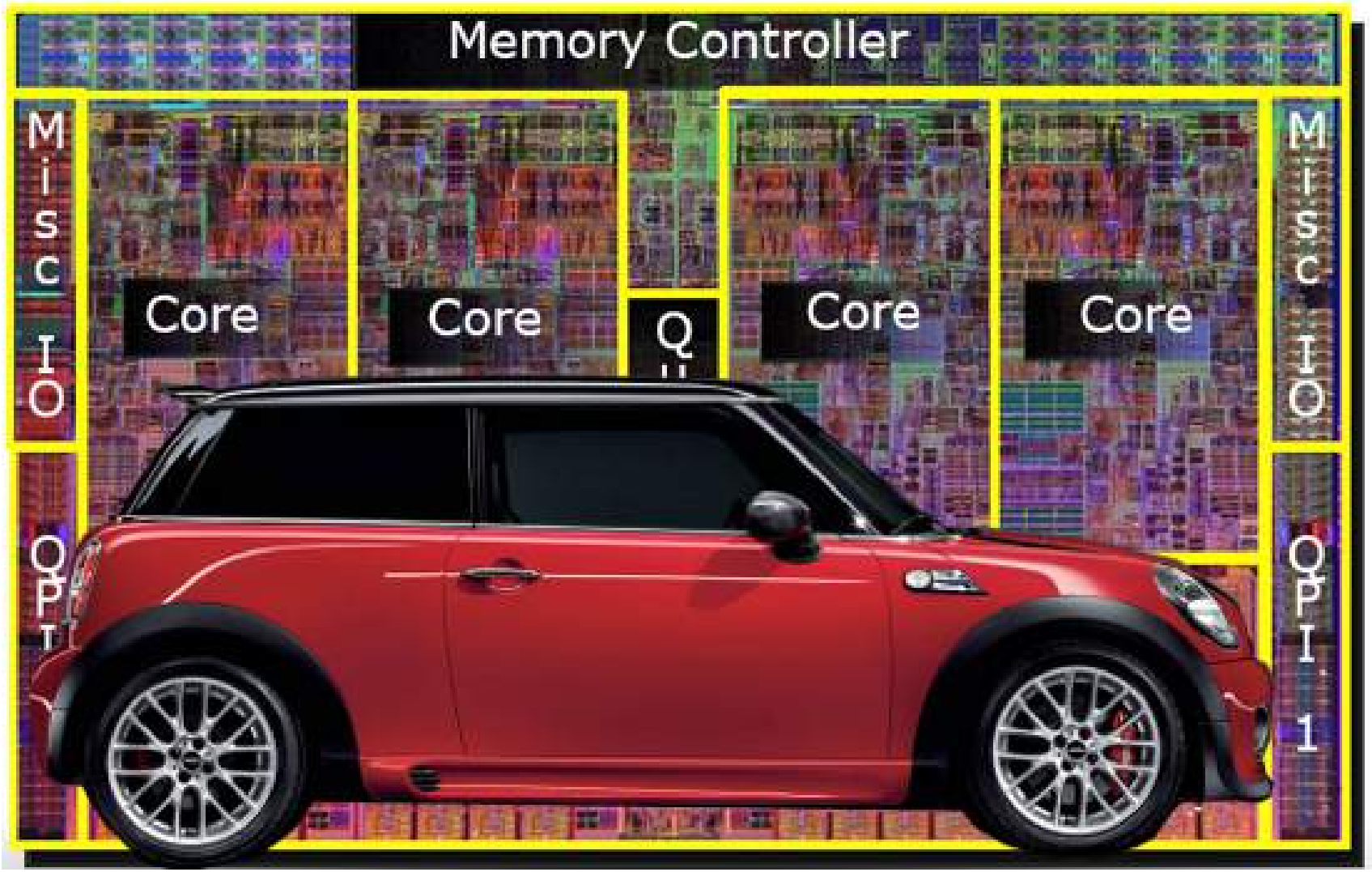


Figura: Dimensiones de un core i7 construido con la tecnología del 4004

Scaling

funcionamiento. . .

- A medida que disminuye el parámetro *tamaño*, los transistores ganan linealmente en rendimiento, pero los “alambres” que conectan los diferentes componentes que conforman los buses internos se transforman en un problema.
- Los caminos de señal, en las escalas actuales y a las frecuencias de trabajo actuales generan:
  1. Delays. Debido a que finalmente son un medio de transmisión con una resistencia y capacidad de distribuida, responsables de la constante de tiempo RC que hace que la señal inyectada en un extremo se propague al otro con demoras.
  2. picoJoules de energía disipados en cada “cable”. Cientos de millones de transistores requieren cientos de millones de cables conectores, que consumen solo algunos pocos Pico Joules cada uno. Hagan cuentas. . .

Scaling

- A pesar de que los “alambres” también se acortan a medida que reducimos las dimensiones, sus efectos no se reducen en consecuencia. Es un aspecto muy complejo en el diseño, por donde pasar un bus. Su cercanía con otros componentes afecta sus propios parámetros RC.
- Los delays consumen fracciones de un ciclo de clock para transportar la señal eléctrica entre sus extremos.
- La disminución en la escala ha transformado a el delay en uno de los factores mas críticos en el diseño de Microprocesadores.
- Dos de las mas de 20 etapas del pipeline del Procesador Pentium IV desarrollado en 2001 tienen como objeto, compensar delays.

Consumo

- La alimentación en un circuito integrado moderno es un tema de consideración por varios factores.
- Se debe distribuir la tensión de alimentación a todo el circuito integrado. Esto motiva desde hace mas de una década que los circuitos integrados dediquen una buena cantidad de terminales de conexión a V_{cc} y Tierra.
- La potencia que se disipa en un transistor CMOS en conmutación, es proporcional a la Capacidad de carga del dispositivo, al cuadrado de la tensión de alimentación y a la frecuencia de conmutación. Su expresión viene dada por:

$$P_d = \frac{1}{2} C_c V^2 f_c \tag{1}$$

Para procesadores destinados a dispositivos portátiles, para dimensionar la capacidad de una batería y su tiempo de duración, mas que la potencia, interesa la energía en Joules, que viene dada por:

$$E_d = C_c V^2 \tag{2}$$

Consumo

Conclusiones de (1) y (2)

1. La tensión de alimentación se redujo en los últimos 30 años de 5V a 0,85V. Esto por sí solo es una reducción drástica en el consumo de un transistor.
2. La capacidad de carga depende de la cantidad de dispositivos que se conecten a la salida de un transistor y de la tecnología de integración empleada.
3. Para una tarea fija, reducir la frecuencia de clock disminuye la potencia disipada pero no tiene efecto con la energía consumida.

Entonces: ¿Por que razón es un problema el Consumo?

- El incremento en la cantidad de transistores CMOS por mm^2 de superficie tiene preminencia por sobre los ahorros de energía individuales de cada transistor debidos al cambio de tecnología.
- Por lo tanto cada vez es mas crítico el manejo del consumo.
- El procesador 4004 de Intel en 1971 tenía poco mas de 2300 transistores y su consumo era de algunas décimas de Watts.Su clock era de 108 KHz (si...leyó bien...Kilo Hertz)
- El procesador Pentium IV Extreme Edition desarrollado en 2001 (30 años después), llegó a consumir 135 Watts. Claro. Tenía cerca de 40 millones de transistores y un clock de 3GHz
- Por lo tanto cada vez existen mas limitaciones tanto con la distribución de la alimentación como con el ahorro de potencia y energía.

Tendencias en reducción del Consumo

- La mayoría de los procesadores actuales contiene bloques de hardware para control de consumo, que se encargan de mantener alimentados solo los bloques funcionales que se necesitan en cada momento. Por ejemplo: Si el procesador no está ejecutando operaciones de Punto Flotante, entonces la Unidad de Punto Flotante se mantiene apagada.
- A pesar de que un transistor esté al corte, existe una corriente de fuga (likage) que circula de todos modos, de modo que interesa determinar la Potencia estática, relacionada con esta corriente:

$$E_e = I_e V^2 \tag{3}$$

- V = Tensión de Alimentación, I_e = corriente de fuga (likage). Cada transistor tiene así una componente adicional de potencia cuando está en corte.
- A medida que aumenta la cantidad de transistores esta corriente se hace mas significativa.
- En 2006 los principales diseñadores establecieron como meta que esta corriente represente solo el 25 % del consumo total del chip. Aún así los modelos de mas alto rendimiento no lograron esta marca.

Arquitectura vs Microarquitectura

Arquitectura

Es el conjunto de recursos accesibles para el programador, que por lo general se mantienen a lo largo de los diferentes modelos de procesadores de esa arquitectura (puede evolucionar pero la tendencia es mantener compatibilidad hacia los modelos anteriores).

- Registros
- Set de instrucciones
- Estructuras de memoria (descriptores de segmento y de página p. ej.)

Micro Arquitectura

Es la implementación en el silicio de la arquitectura. Lo que está detrás del set de registros y del modelo de programación. Puede ser muy simple o sumamente robusta y poderosa. Camba de un modelo a otro dentro de una misma familia.

Arquitectura vs Microarquitectura

Ejemplo

La arquitectura IA-32 se inicia con el procesador 80386 en 1985, y llega hasta los procesadores Intel Core i7, i5, i3, ATOM y Xeon actuales.
En el camino han pasado diferentes generaciones de Micro-Arquitectura para mas de 25 modelos de procesadores.

Definición de la Arquitectura de un Computador

- Es sin dudas la tarea mas ardua de un diseñador.
- Se trata de definir los aspectos mas relevantes en la arquitectura de un computador que maximicen su rendimiento, sin dejar de satisfacer otras limitaciones impuestas por los usuarios, como un costo económico que lo haga accesible, o un consumo de energía moderado.
- Comprende el diseño del set de instrucciones, el manejo de la memoria y sus modos de direccionamiento, los restantes bloques funcionales que componen el CPU, el diseño lógico, y el proceso de implementación
- La implementación comprende el diseño del circuito integrado, su encapsulado, montaje, alimentación y refrigeración.

Definición de la Arquitectura de un Computador

skills necesarios

No es posible realizar esta tarea con éxito sin tener manejar de manera solvente varias tecnologías diferentes:

- Diseño lógico.
- Tecnología de encapsulado
- Funcionamiento y diseño de compiladores y de Sistemas Operativos.

Definiendo un ISA

Nos referimos a *Instruction Set Architecture*, como el set de instrucciones visibles por el programador. Es además el límite entre el software y el hardware.

Clases de ISA. ISAs con Registros de Propósito general vs. Registros dedicados. ISAs registro-memoria vs. ISAs Load Store.

Direccionamiento de Memoria. Alineación obligatoria de datos vs. administración de a bytes.

Modos de Direccionamiento. Como se especifican los operandos.

Tipos y tamaños de operandos. Enteros, Punto Flotante, Punto Fijo.Diferentes tamaños y precisiones.

Operaciones. Pocas Simples (RISC) o Muchas Complejas (CISC).

Instrucciones de control de flujo Saltos condicionales, calls.

Longitud del código Instrucciones de tamaño fijo vs. variable.

Microarquitectura = Organización + Hardware

Organización

Se refiere a los detalles de implementación de la ISA. Es decir

- Organización e interconexión de la memoria.
- Diseño de los diferentes bloques de la CPU y para implementar el set de instrucciones.
- Implementación de paralelismo a nivel de Instrucciones, y/o de datos.

Podemos así encontrar procesadores que poseen la misma ISA pero una organización muy diferente. Ejemplo: Los procesadores AMD FX y los Intel Core i7, tienen la misma ISA, sin embargo organizan su caché y su motor de ejecución de maneras diferentes.

Hardware

Se refiere a los detalles de diseño lógico y tecnología de fabricación. Existen procesadores con las mismas ISA y organización, pero que a nivel de hardware y diseño lógico de detalle son muy diferentes. Ejemplo: el Pentium 4, diseñado para desktop, y el Pentium 4 M para notebooks. Este tenia una cantidad de lógica para control del consumo de energía que hacía su hardware muy diferente del otro.

Evolución

Pioneros:

Maurice Wilkes en 1947 con la primer memoria de tanque de mercurio para la computadora EDSAC. Capacidad 2 bytes.

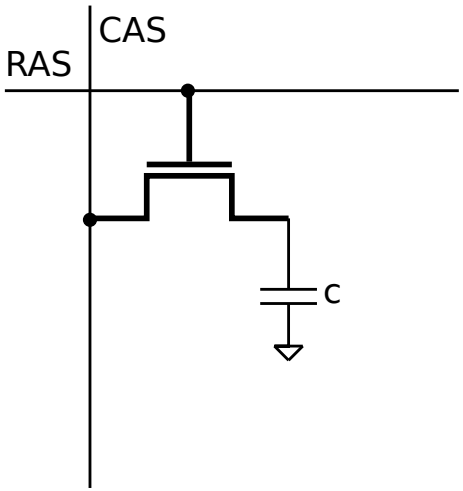
Visionarios:

“640K debe ser suficiente memoria para cualquiera....”
Bill Gates. 1981

Clasificación

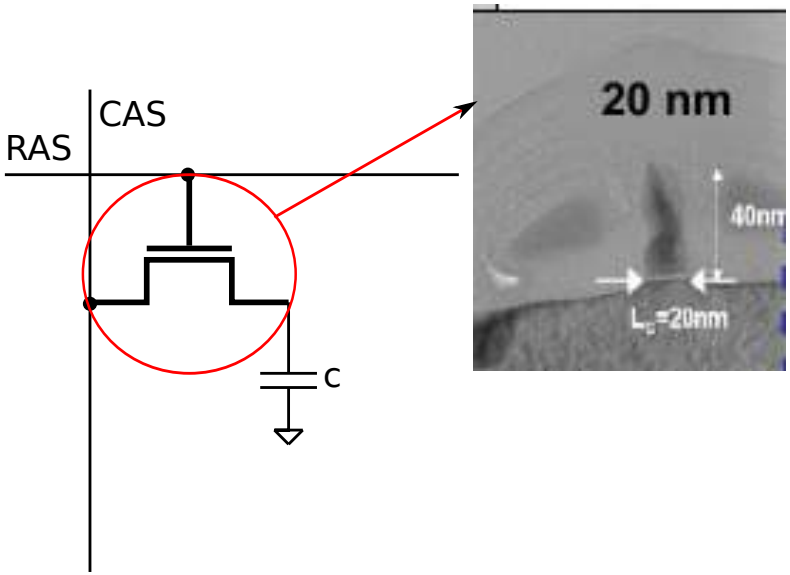
- Memorias Solo lectura.
  Se trata de memorias capaces de retener la información almacenada cuando se les desconecta la alimentación. De acuerdo con las diferentes tecnologías han evolucionado desde ser memorias denominadas ***ROM*** (por Read Only Memory), que en sus primeras implementaciones debían ser grabadas por el fabricante, hasta las actuales flash memories que pueden ser grabadas por algoritmos de escritura on the fly por el usuario, y cuyo ejemplo mas habitual es el pen drive o las tarjetas microsd de las cámaras fotográficas.
- Memorias Lectura Escritura
  Se trata de memorias volátiles en lo que respecta a la permanencia de la información una vez interrumpida la alimentación eléctrica. Sin embargo estas memorias pueden almacenar mayores cantidades de información y modificarla en tiempo real a gran velocidad en comparación con las ROM.
  Se clasifican de acuerdo con la tecnología y su diseño interno en dinámicas y estáticas.

Memorias dinámicas



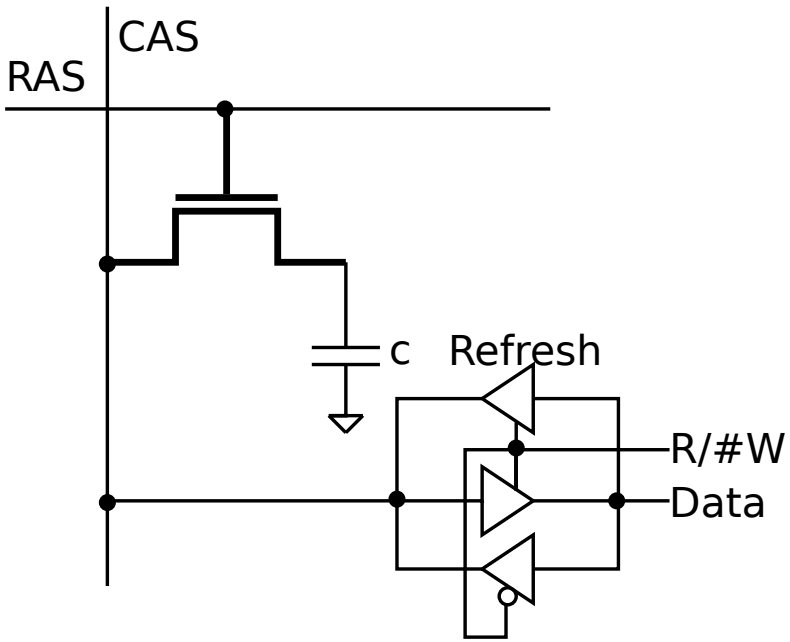
- Almacena la información como una carga en un capacitor y la sostiene con la ayuda de un transistor.
- Una celda (un bit) se implementa con un solo transistor $=>$ máxima capacidad de almacenamiento por chip.
- Ese transistor está generalmente en estado de Corte. Consume mínima energía.

Memorias dinámicas



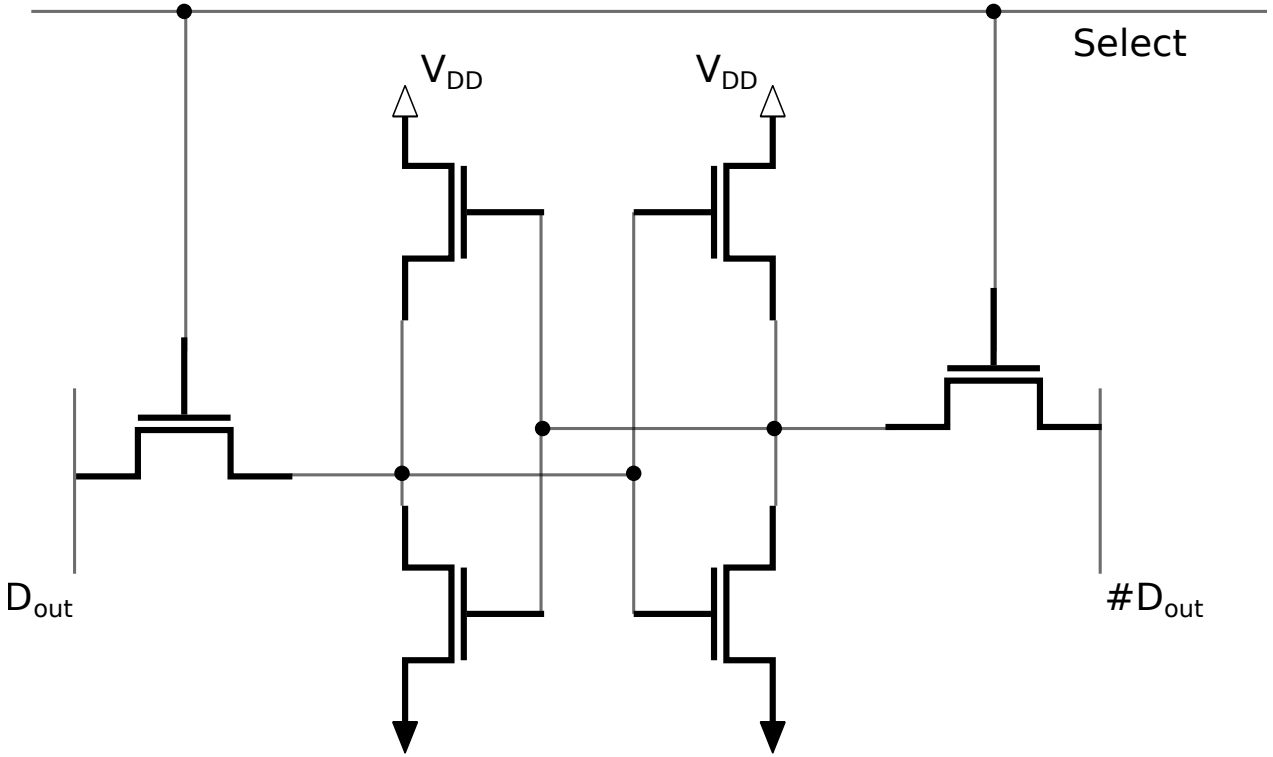
- Almacena la información como una carga en un capacitor y la sostiene con la ayuda de un transistor.
- Una celda (un bit) se implementa con un solo transistor $=>$ máxima capacidad de almacenamiento por chip.
- Ese transistor está generalmente en estado de Corte. Consume mínima energía.

Memorias dinámicas



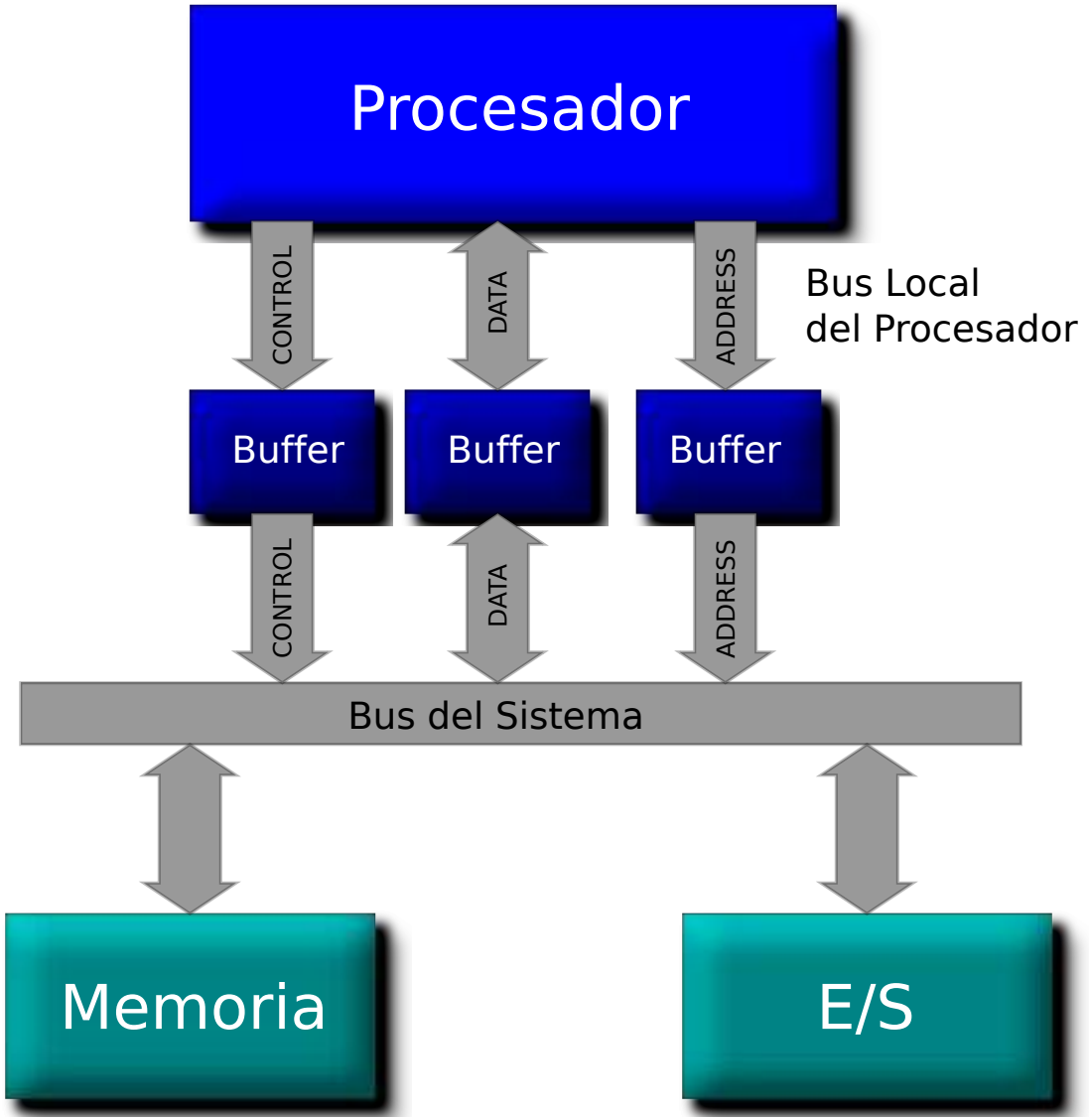
- Al leer el bit, se descarga la capacidad.
- Una celda de DRMA **Necesita regenerar la carga** cada vez que se la lee.
- Esta operación se realiza por realimentación mediante buffers
- Aumenta entonces el tiempo total que demanda el acceso de la celda, ya que no libera la operación hasta no haber repuesto el estado de carga del capacitor.

Memorias estáticas



- Almacena la información en un biestable.
- Una celda (un bit) se compone de seis transistores. Por lo tanto tiene menor capacidad de almacenamiento por chip.
- Tres de los seis transistores están en saturación (consumen energía máxima en forma permanente) y los otros tres al corte (consumen mínima energía). Esto genera mayor consumo de energía por celda.
- La lectura es directa y no destructiva por lo cual el tiempo de acceso es muy bajo en comparación con las memorias dinámicas. De hecho son luego de los registros del procesador el medio de almacenamiento más rápido para un procesador.

Conexión básica (Según Von Newmann)



- Desde fines de los años 80, los procesadores desarrollaban velocidades muy superiores a los tiempos de acceso a memoria.
- En este escenario, el procesador necesita generar wait states para esperar que la memoria esté lista (“READY”) para el acceso.
- ¿Tiene sentido lograr altos clocks en los procesadores si no puede aprovecharlos por tener que esperar (wait) a la memoria?

El problema...

El problema consiste en decidir que tipo de RAM usar en el sistema. Hay dos opciones...

- **RAM dinámica (DRAM)**
  - Consumo mínimo.
  - Capacidad de almacenamiento comparativamente alta.
  - Costo por bit bajo.
  - Tiempo de acceso alto (lento), debido al circuito de regeneración de carga.
- **RAM estática (SRAM)**
  - Alto consumo relativo.
  - Capacidad de almacenamiento comparativamente baja.
  - Costo por bit alto.
  - Tiempo de acceso bajo (es mas rápida).

Conclusión:

Si construimos el banco de memoria utilizando RAM estática, el costo y el consumo de la computadora son altos. Si construimos el banco de memoria utilizando RAM dinámica, no aprovechamos la velocidad del procesador.

La solución. . . Memoria Cache

- Se trata de un banco de SRAM de muy alta velocidad, que contiene una copia de los datos e instrucciones que están en memoria principal.
- El arte consiste en que esta copia esté disponible justo cuando el procesador la necesita permitiéndole acceder a esos ítems sin recurrir a wait states.
- Combinada con una gran cantidad de memoria DRAM, para almacenar el resto de códigos y datos, resuelve el problema mediante una solución de compromiso típica.
- Requiere de hardware adicional que asegure que este pequeño banco de memoria cache contenga los datos e instrucciones mas frecuentemente utilizados por el procesador.

Características y métricas

El tamaño del banco de memoria cache debe ser:

1. Suficientemente grande para que el procesador resuelva la mayor cantidad posible de búsquedas de código y datos en esta memoria asegurando una alta performance.
2. Suficientemente pequeña para no afectar el consumo ni el costo del sistema.

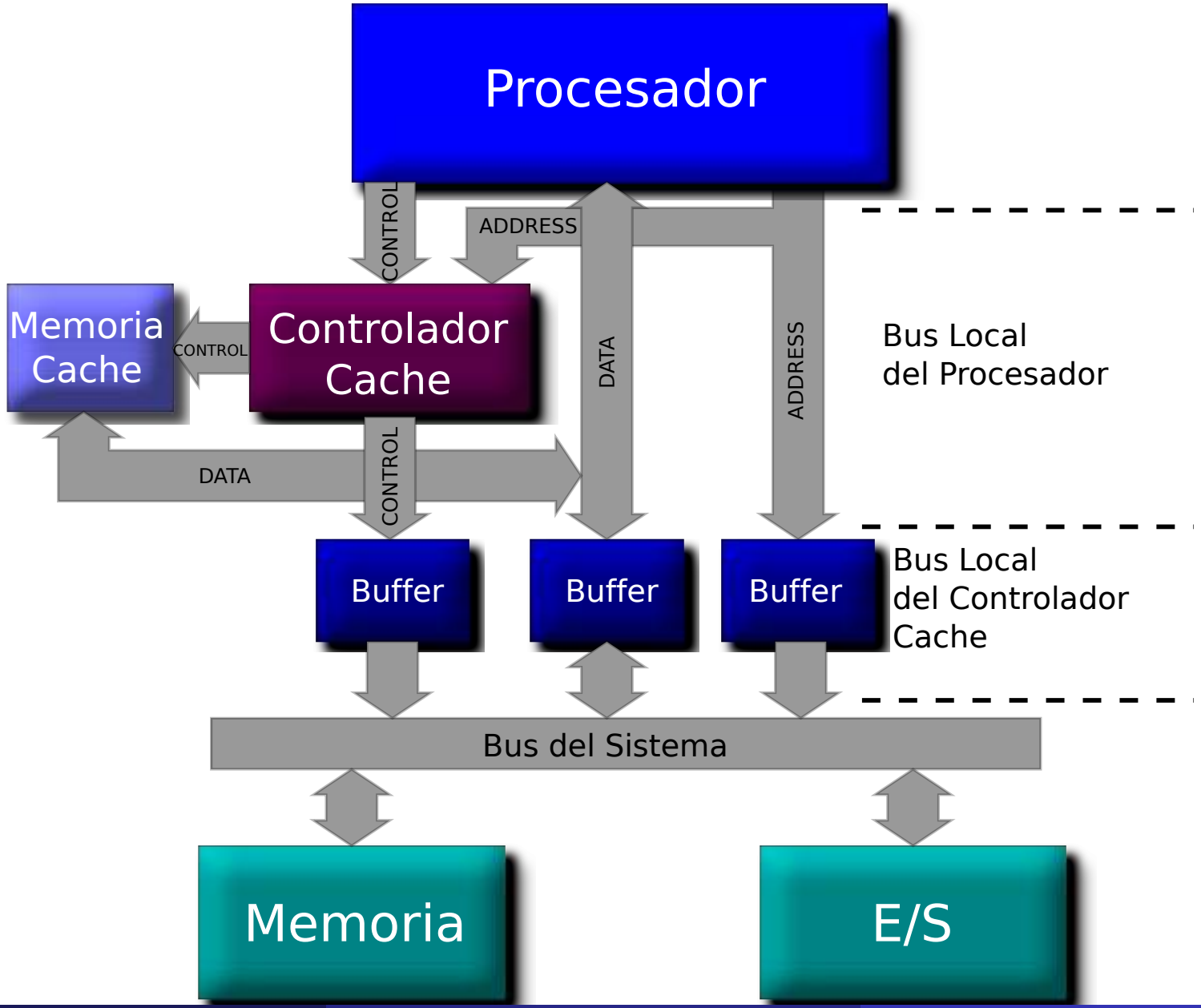
Hit cuando se accede a un ítem (dato o código) y éste *se encuentra* en la memoria cache

Miss cuando se accede a un ítem (dato o código) y éste *no se encuentra* en la memoria cache

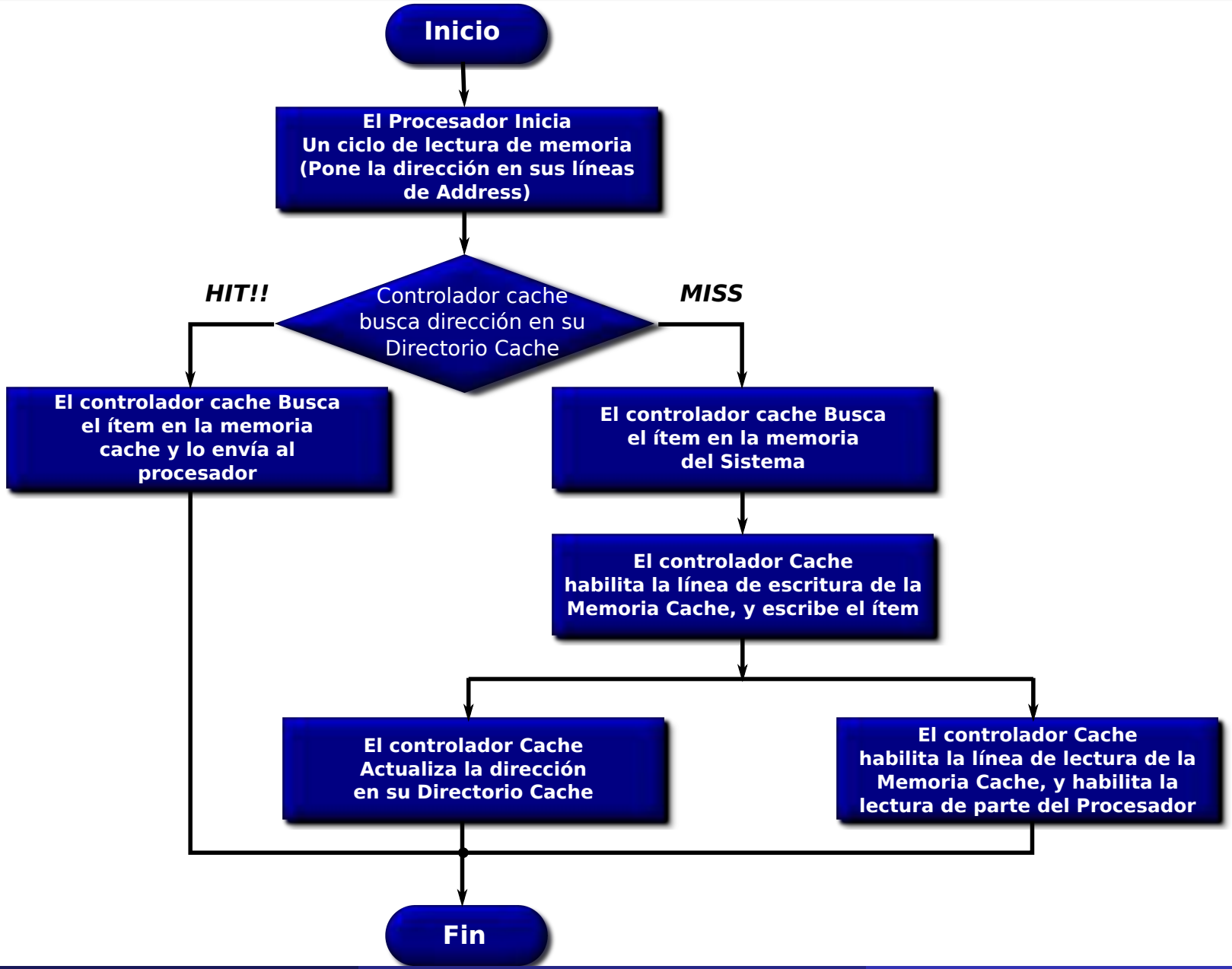
hit rate $hitrate = \frac{Cantidad\ de\ Accesos\ con\ hit}{Cantidad\ de\ Accesos\ Totales}$

Se espera un hit rate lo mas alto posible

Subsistema Cache de Hardware



Operación de acceso a memoria para lectura



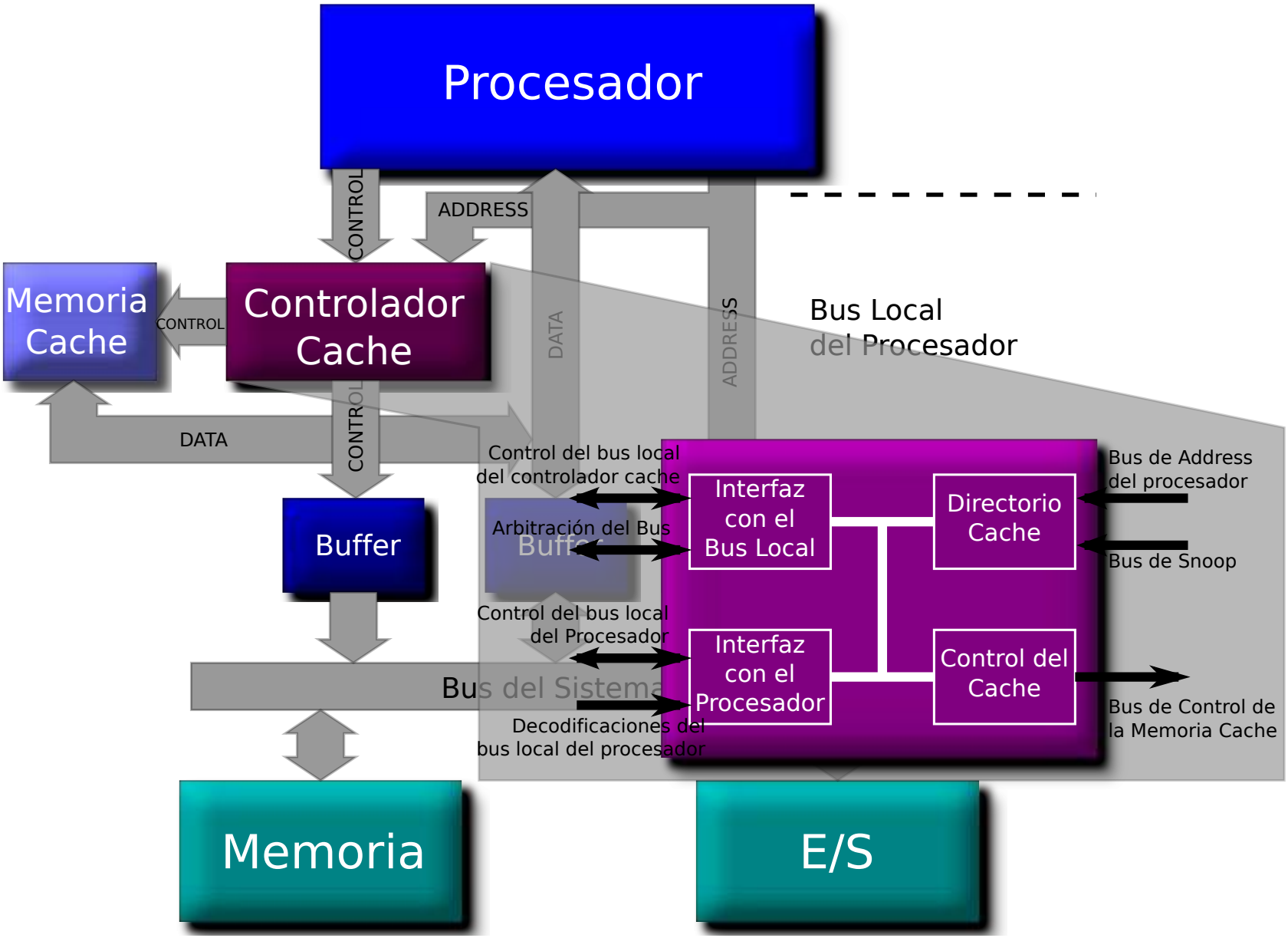
Principio de funcionamiento. . . ¿black magic?

- El controlador cache trabaja mediante dos principios que surgen de analizar el comportamiento de los algoritmos de software que se emplean habitualmente.
- Principio de vecindad temporal: Si un ítem es referenciado, la probabilidad de ser referenciado en el futuro inmediato es alta.
- Principio de vecindad espacial: Si un ítem es referenciado, es altamente probable que sean referenciados sus ítems vecinos.
- Ejemplo: Algoritmo de convolución

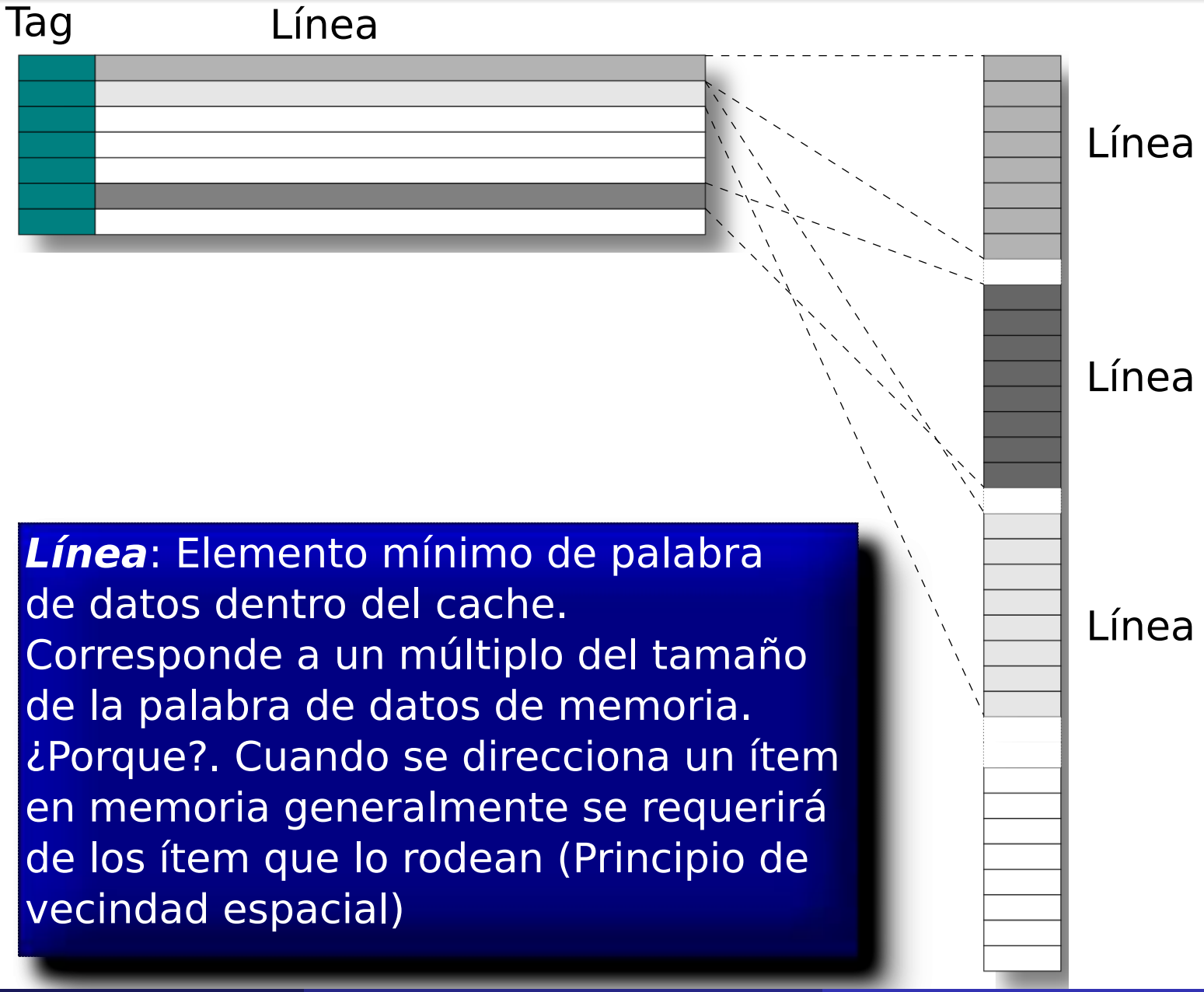
```
1     for (i = 0 ; i < 256 ; i++)
2     {
3         suma = 0.0f;
4         for (j = 0 ; (j <= i && j < 256) ; j++)
5             suma += v0[i-j] * v1[j];
6             fAux[i] = suma;
7     }
```

- i, j, suma, se utilizan a menudo. Por lo tanto si se mantienen en el cache, el tiempo de acceso a estas variables por parte del procesador es óptimo.

El Controlador Cache

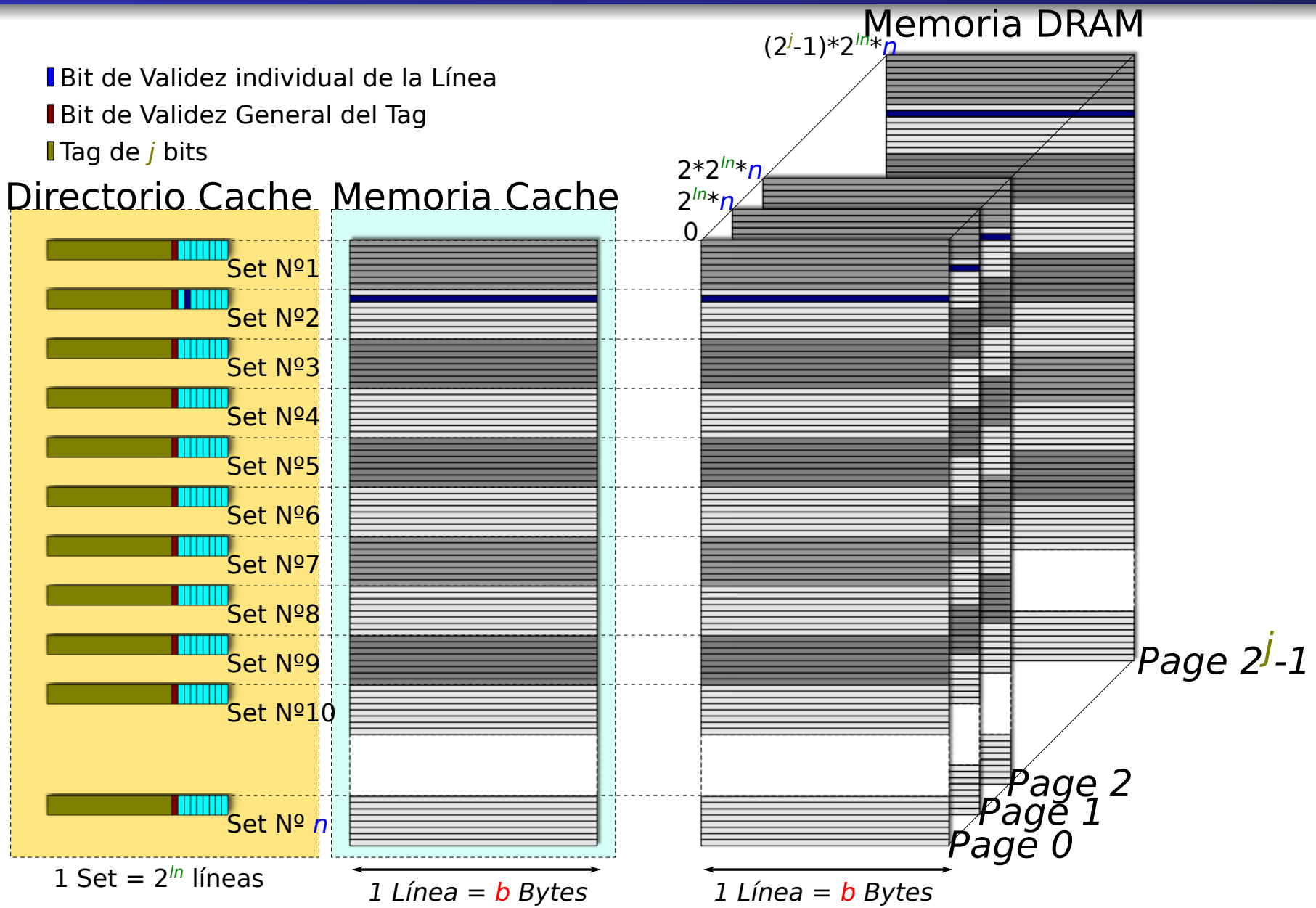


Organización del cache. Líneas

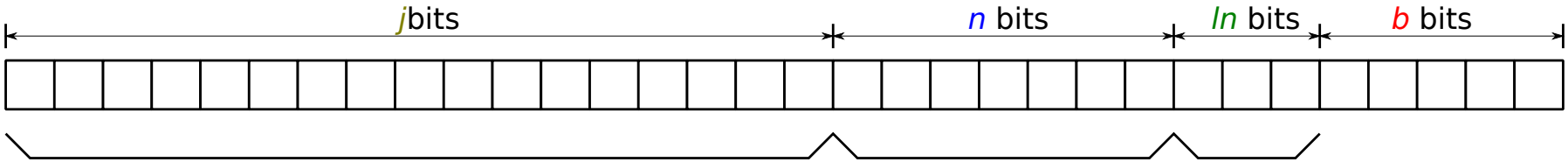


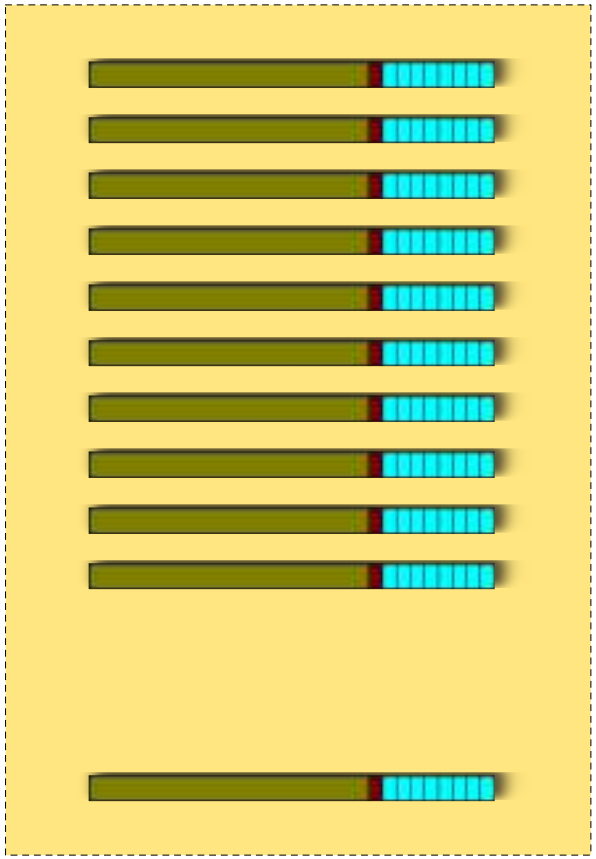
Línea: Elemento mínimo de palabra de datos dentro del cache.
Corresponde a un múltiplo del tamaño de la palabra de datos de memoria.
¿Porque?. Cuando se direcciona un ítem en memoria generalmente se requerirá de los ítem que lo rodean (Principio de vecindad espacial)

Sistema Cache de Mapeo Directo

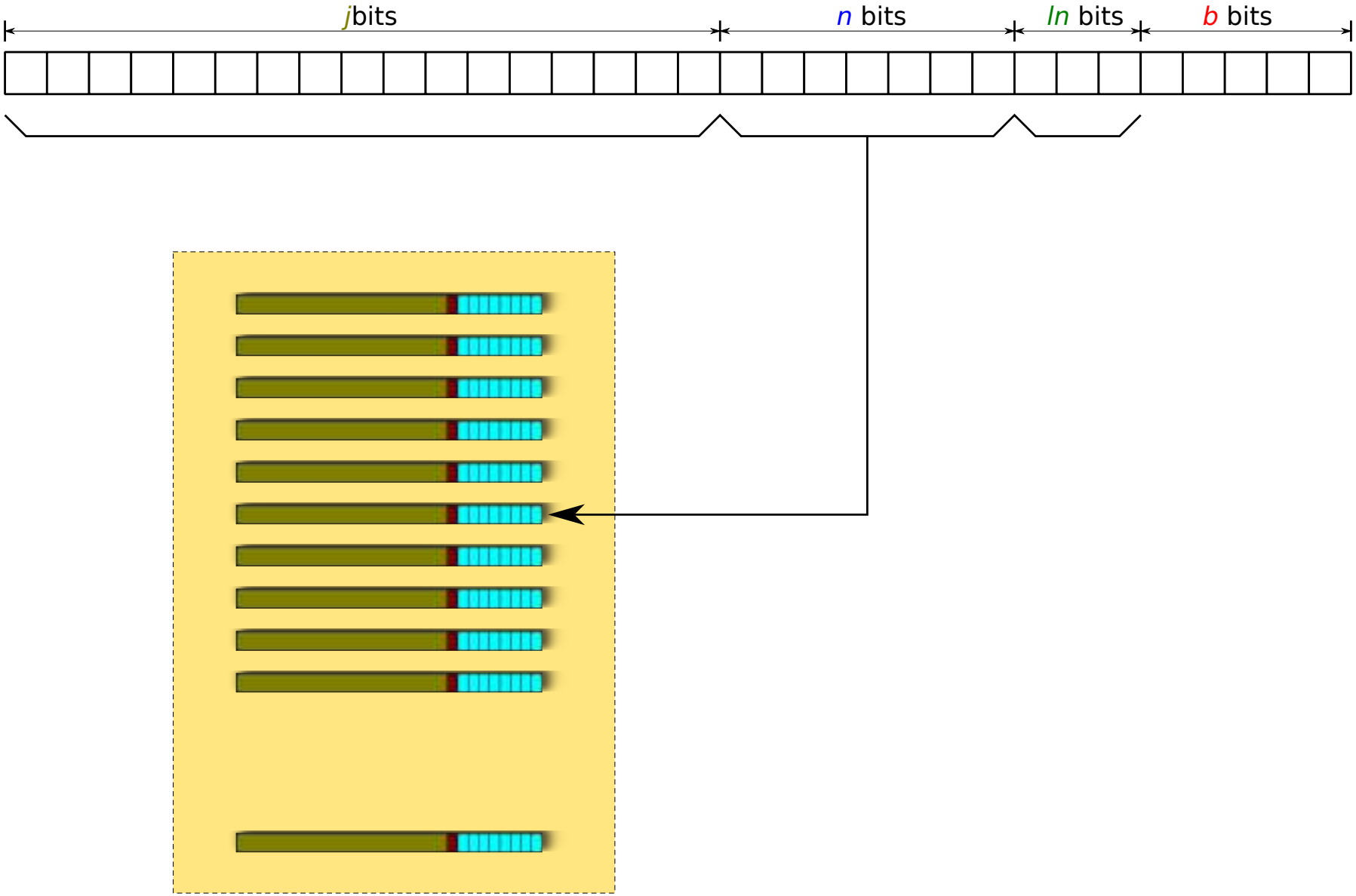


Sistema Cache de Mapeo Directo

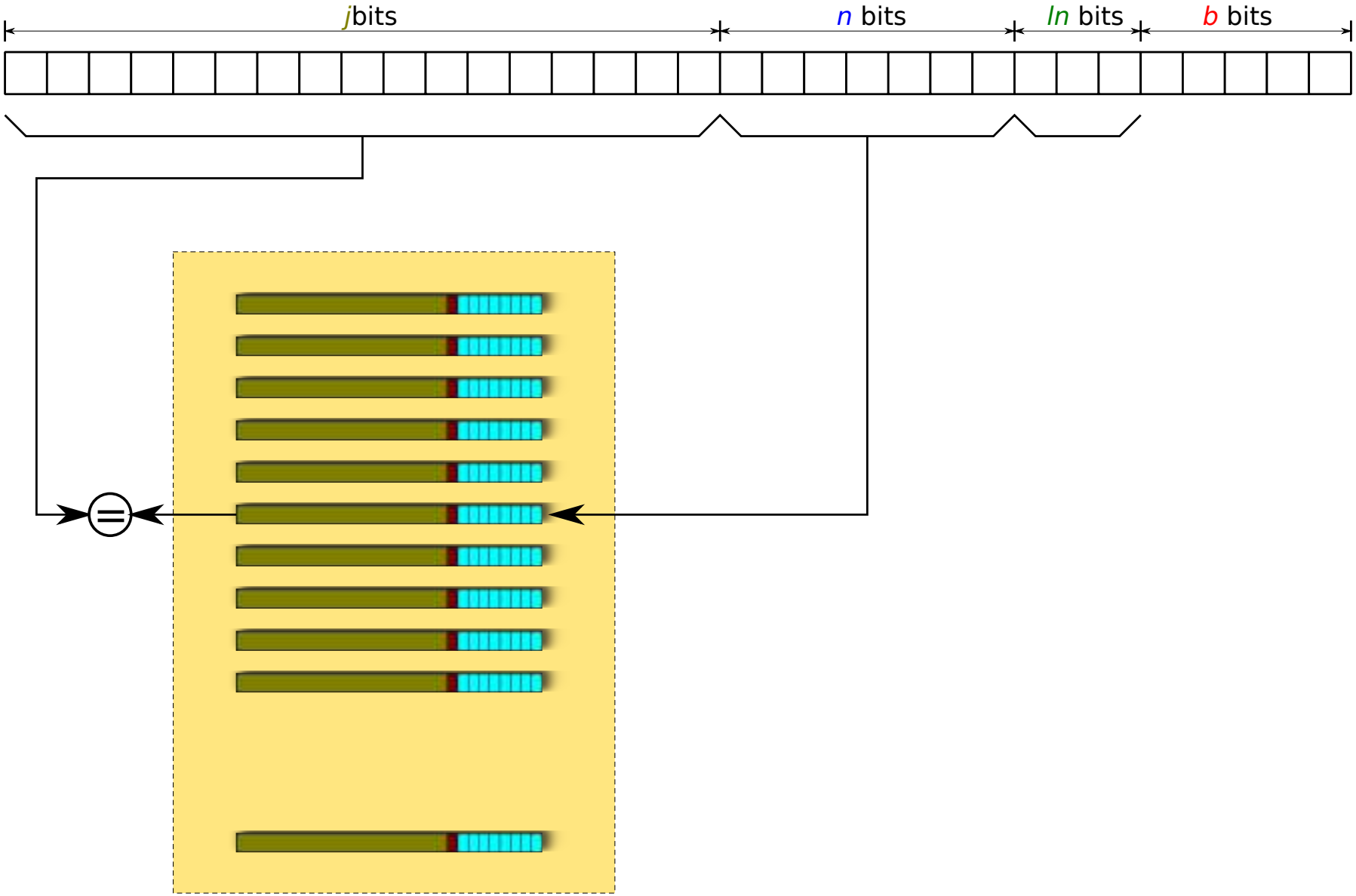




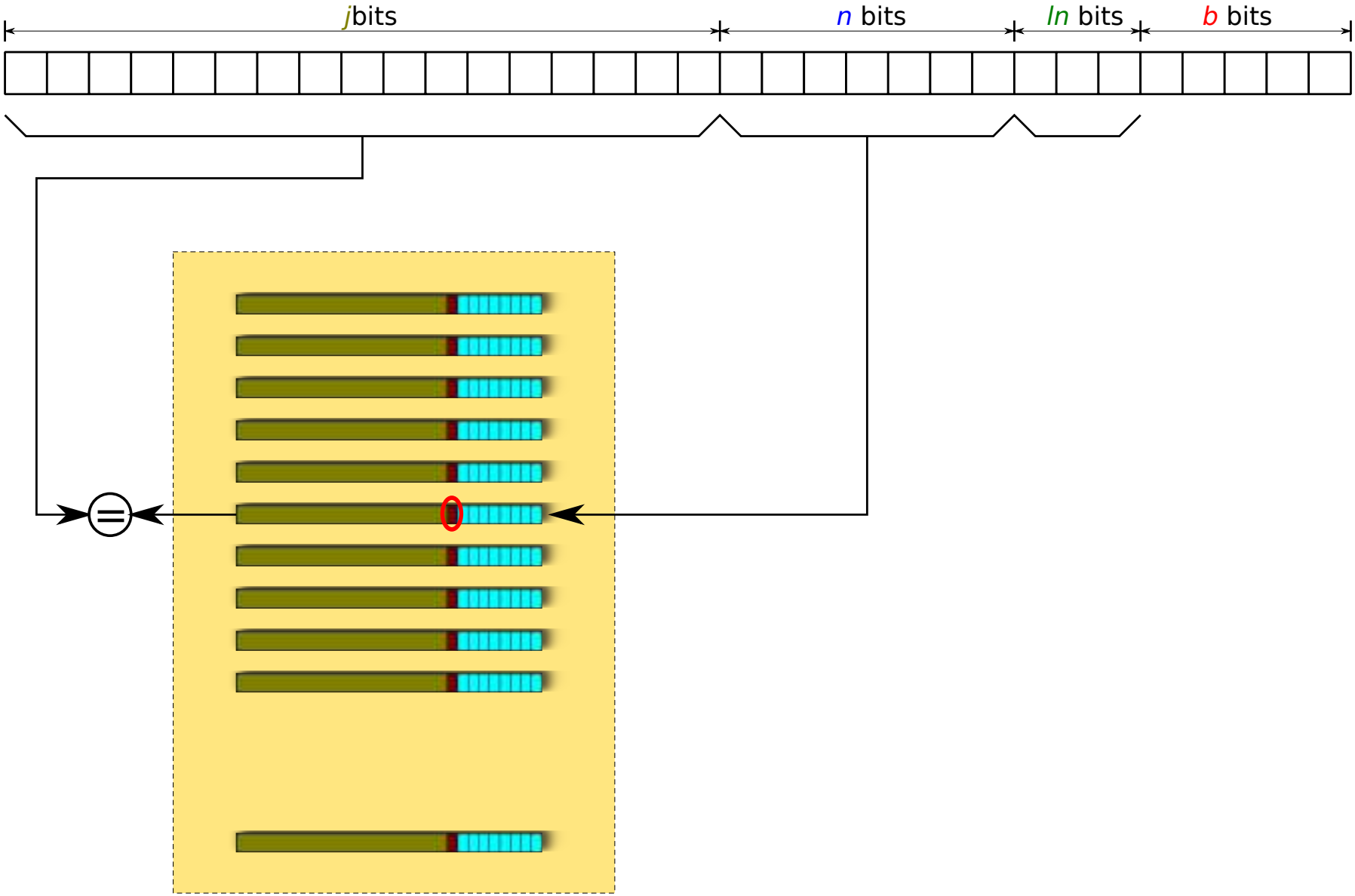
Sistema Cache de Mapeo Directo



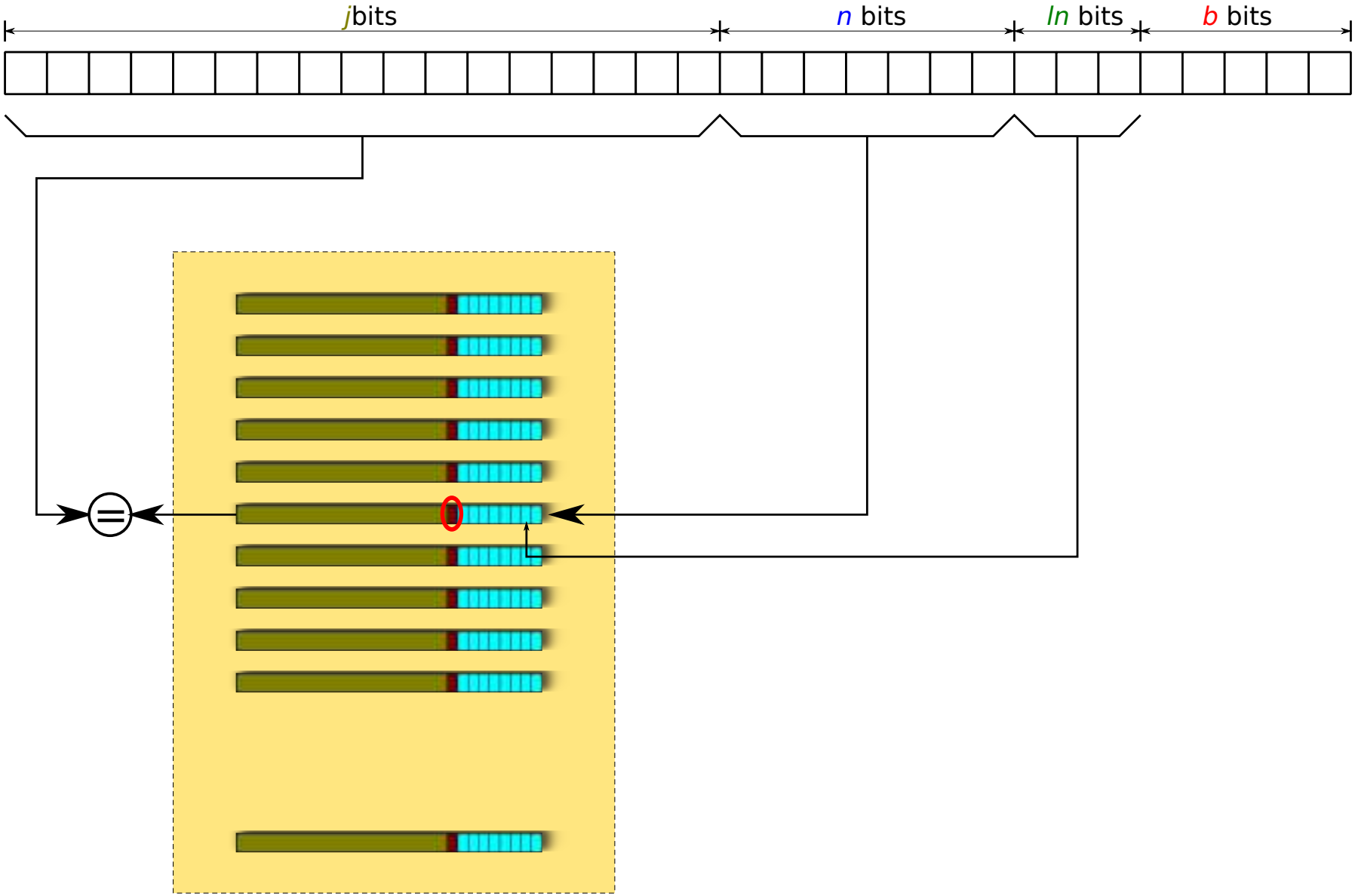
Sistema Cache de Mapeo Directo



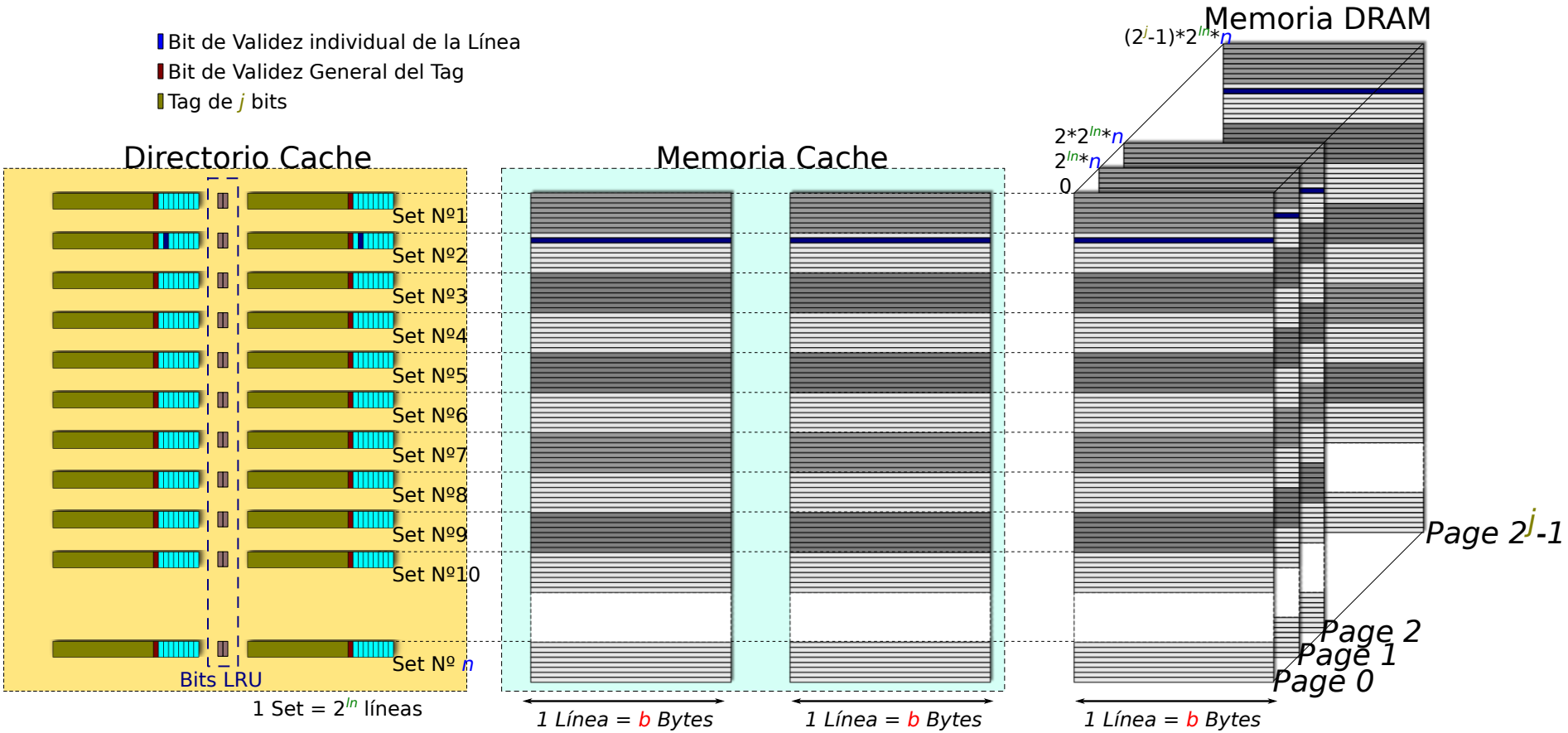
Sistema Cache de Mapeo Directo



Sistema Cache de Mapeo Directo



Sistema Cache Asociativo de 2 Vías



¿Que ocurre durante las escrituras?

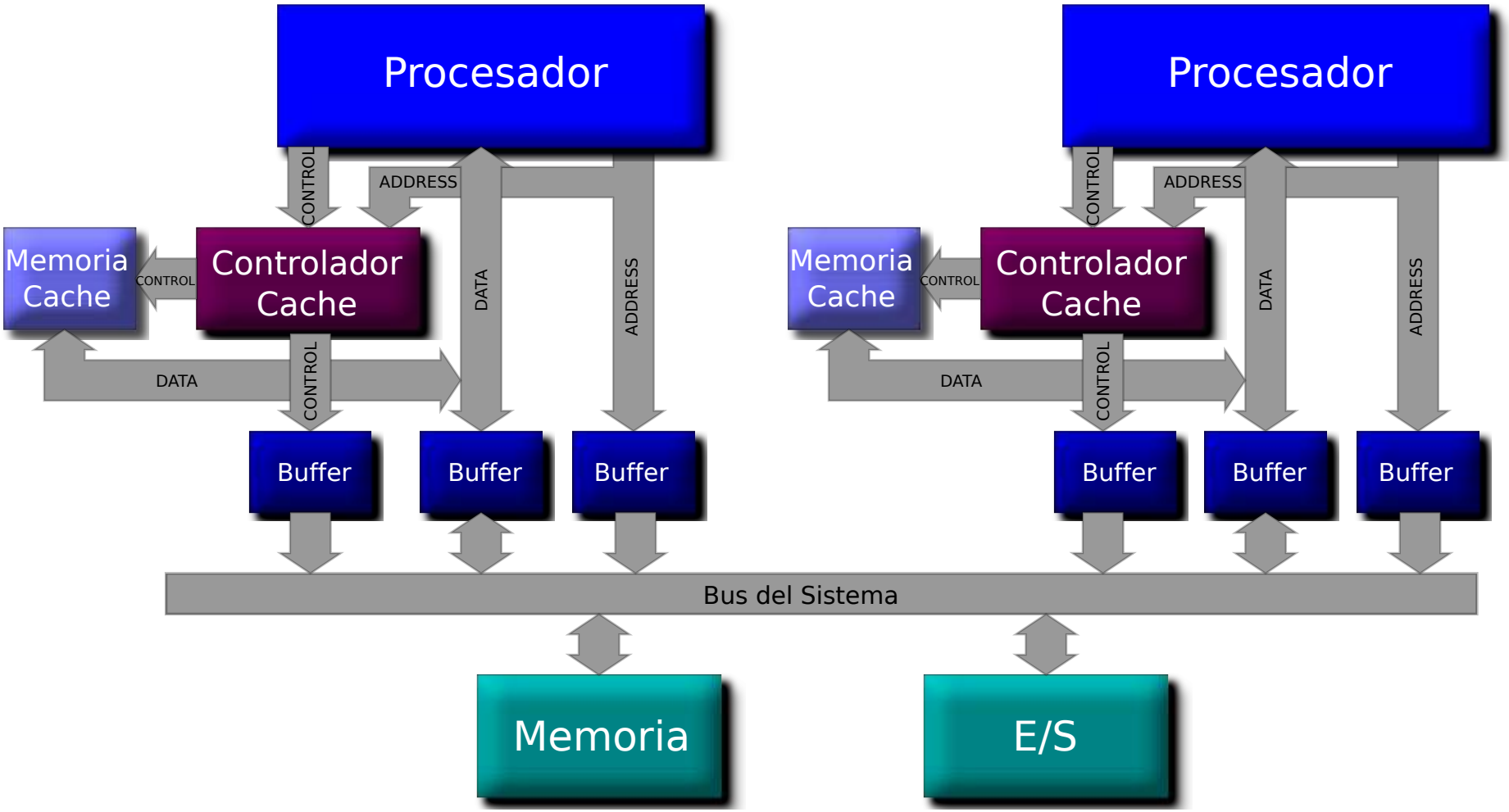
- Una variable que está en el caché también está alojada en alguna dirección de la DRAM.
- Ambos valores deben ser iguales
- Cuando el procesador la modifica hay varios modos de actuar

  Write through el procesador escribe en la DRAM y el controlador cache refresca el cache con el dato actualizado

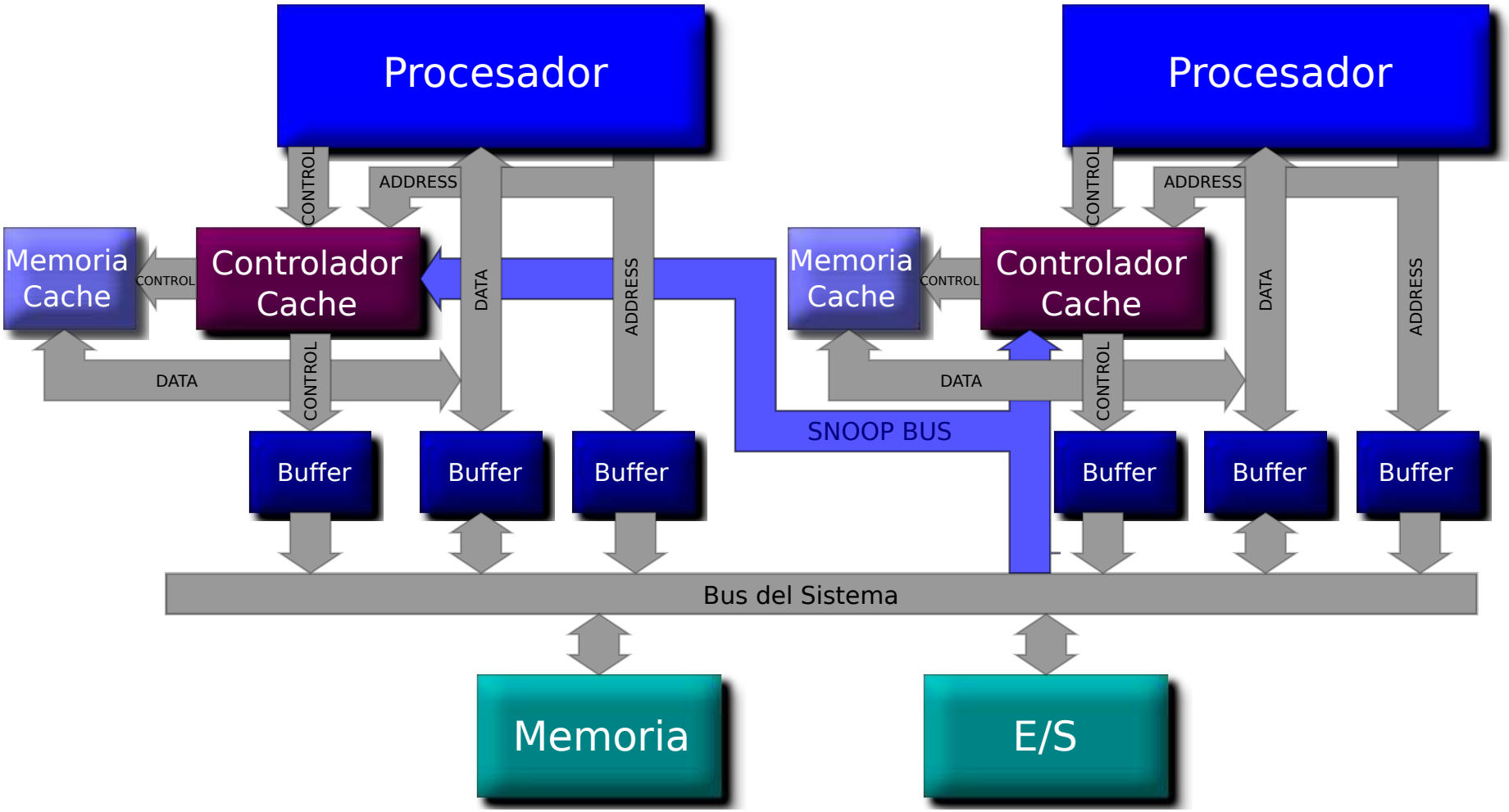
  Write through buffered el procesador actualiza la SRAM cache, y el controlador cache luego actualiza la copia en memoria DRAM mientras el procesador continúa ejecutando instrucciones y usando datos de la memoria cache

  Copy back Se marcan las líneas de la memoria cache cuando el procesador escribe en ellas. Luego en el momento de eliminar esa línea del caché el controlador cache deberá actualizar la copia de DRAM.
- Si el procesador realiza un miss mientras el controlador cache está accediendo a la DRAM para actualizar el valor, deberá esperar hasta que controlador cache termine la actualización para recibir desde este la habilitación de las líneas de control para acceder a la DRAM.

Coherencia en sistemas SMP



Coherencia en sistemas SMP



Protocolo MESI

M - Modified : Línea presente solamente en éste cache que varió respecto de su valor en memoria del sistema (dirty). Requiere write back hacia la memoria del sistema antes que otro procesador lea desde allí el dato (que ya no es válido).

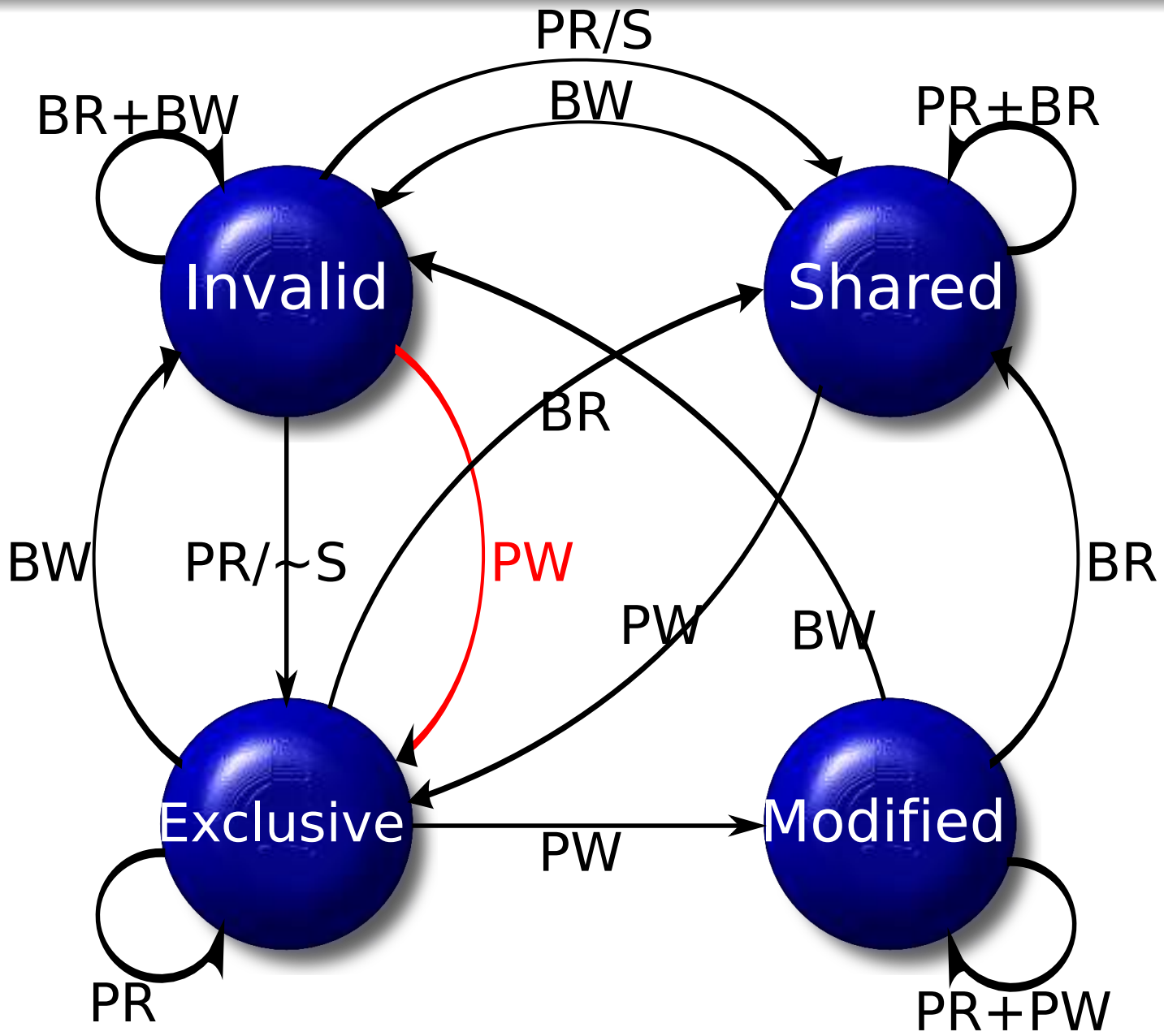
E – Exclusive Línea presente solo en esta cache, que coincide con la copia en memoria principal (clean).

S – Shared Línea del cache presente y puede estar almacenada en los caches de otros procesadores.

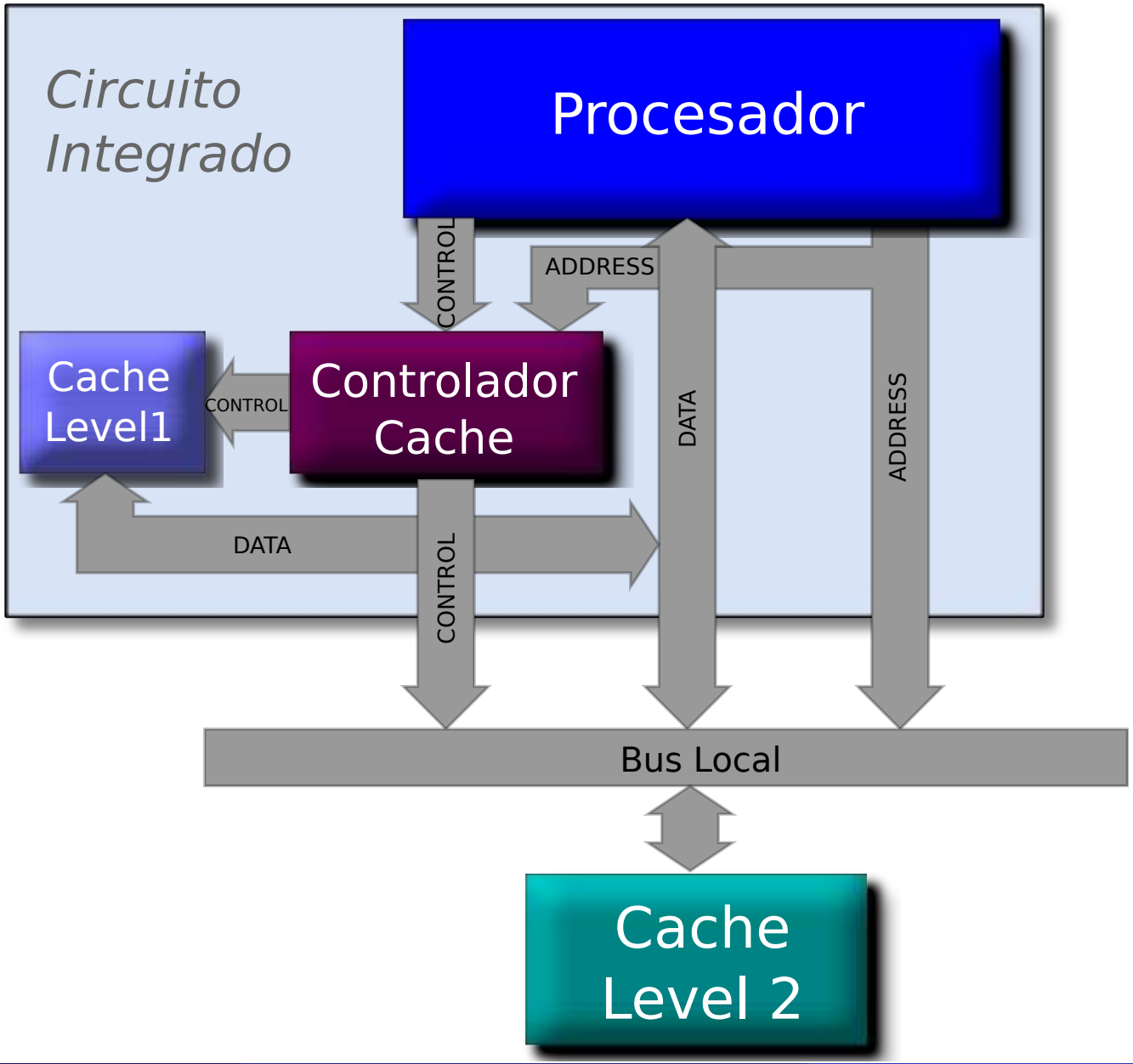
I – Invalid Línea de cache no es válida.

- Aplica a cache L1 de datos y L2/L3
- Para cache L1 de código solo Shared e Invalid

Protocolo MESI



Cache Multinivel



Máquina de estados elemental

- En la década del 40 Von Newman definió un modelo básico de CPU, que a estas alturas está mas que superado.
- Sin embargo algunos conceptos de ese modelo viven en los mas modernos procesadores.
- Uno de ellos es la máquina de ejecución

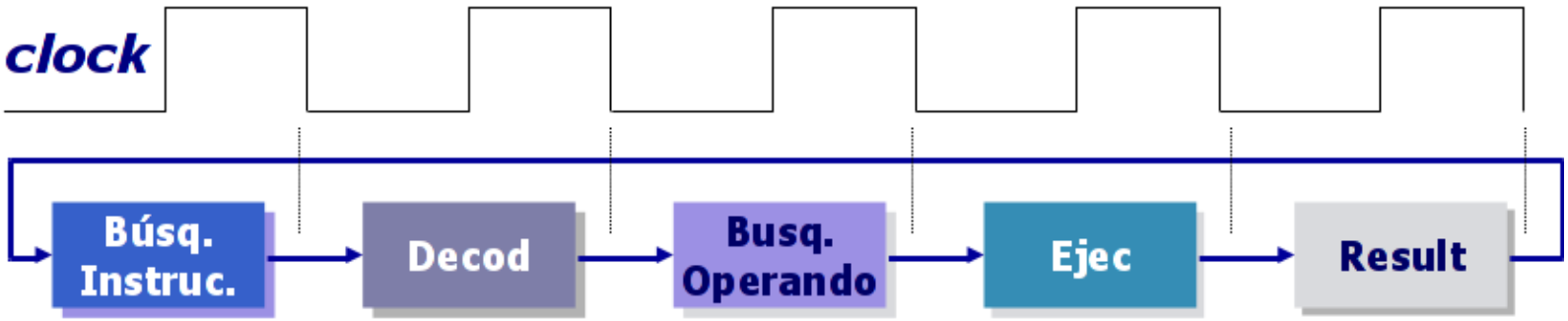


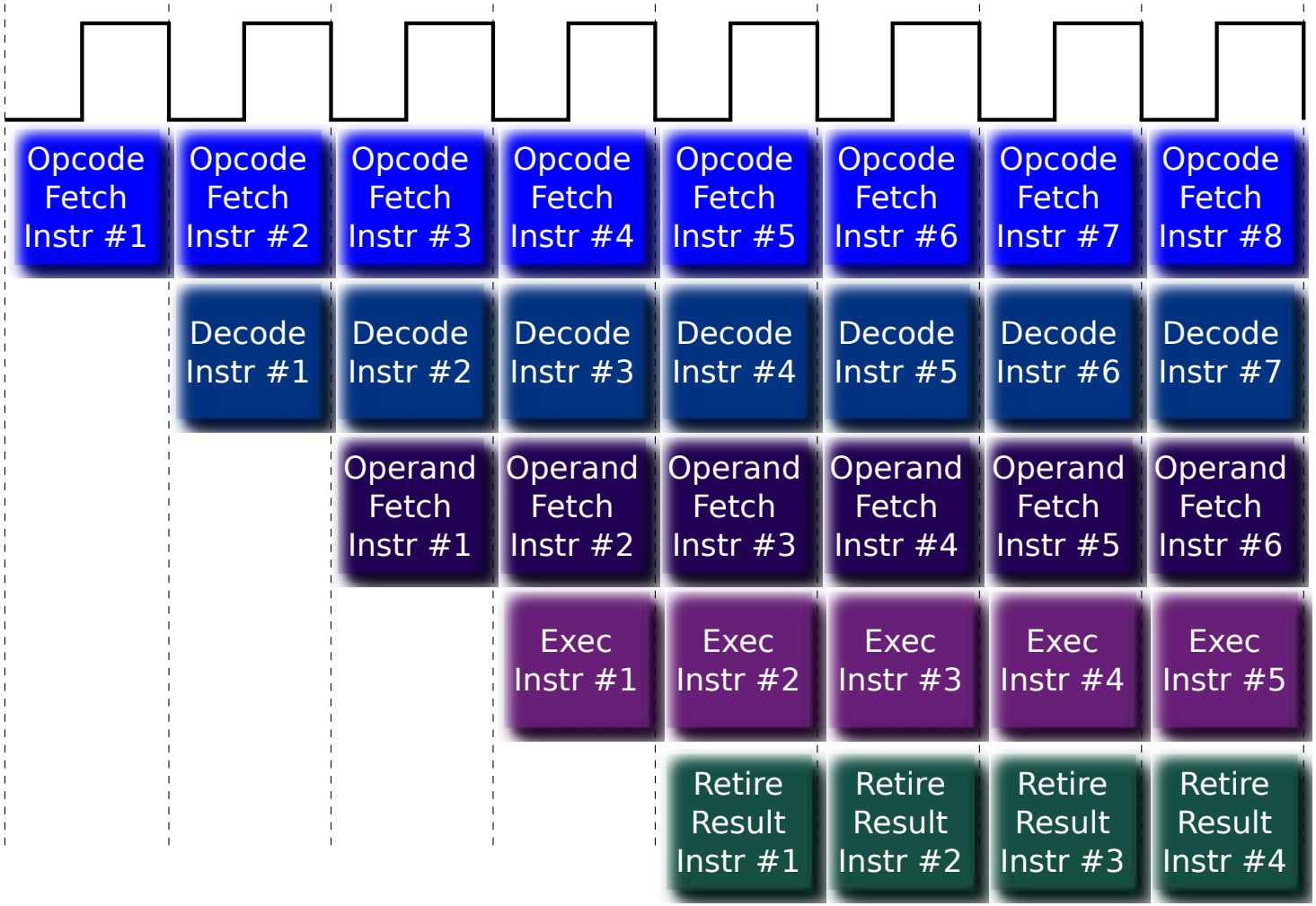
Figura: Etapas mínimas en la ejecución de una instrucción

- En las primeras generaciones de microprocesadores cada etapa de este ciclo se ejecutaba en un ciclo de clock, y la CPU entera estaba dedicada a esa tarea.
- Ejecutar una instrucción insumía de varios ciclos de clock. . .

Pipeline

- Técnica que permite crear el efecto de superponer en el tiempo, la ejecución de varias instrucciones a la vez.
- Con ésta técnica se formaliza el concepto de ***Instruction Level Paralelism (ILP)***.
- No requiere hardware adicional. Solo se necesita lograr que los diferentes bloques del procesador que resuelven las diferentes etapas que componen la ejecución de una instrucción, operen en forma simultánea.
- El truco para lograrlo es que cada bloque funcional trabaje en paralelo con el resto pero en instrucciones diferentes.
- Es algo parecido al concepto de una línea de montaje, en donde cada operación se descompone en partes, y se ejecutan en un mismo momento diferentes partes de diferentes operaciones.
- Cada parte se denomina etapa (stage)

Pipeline

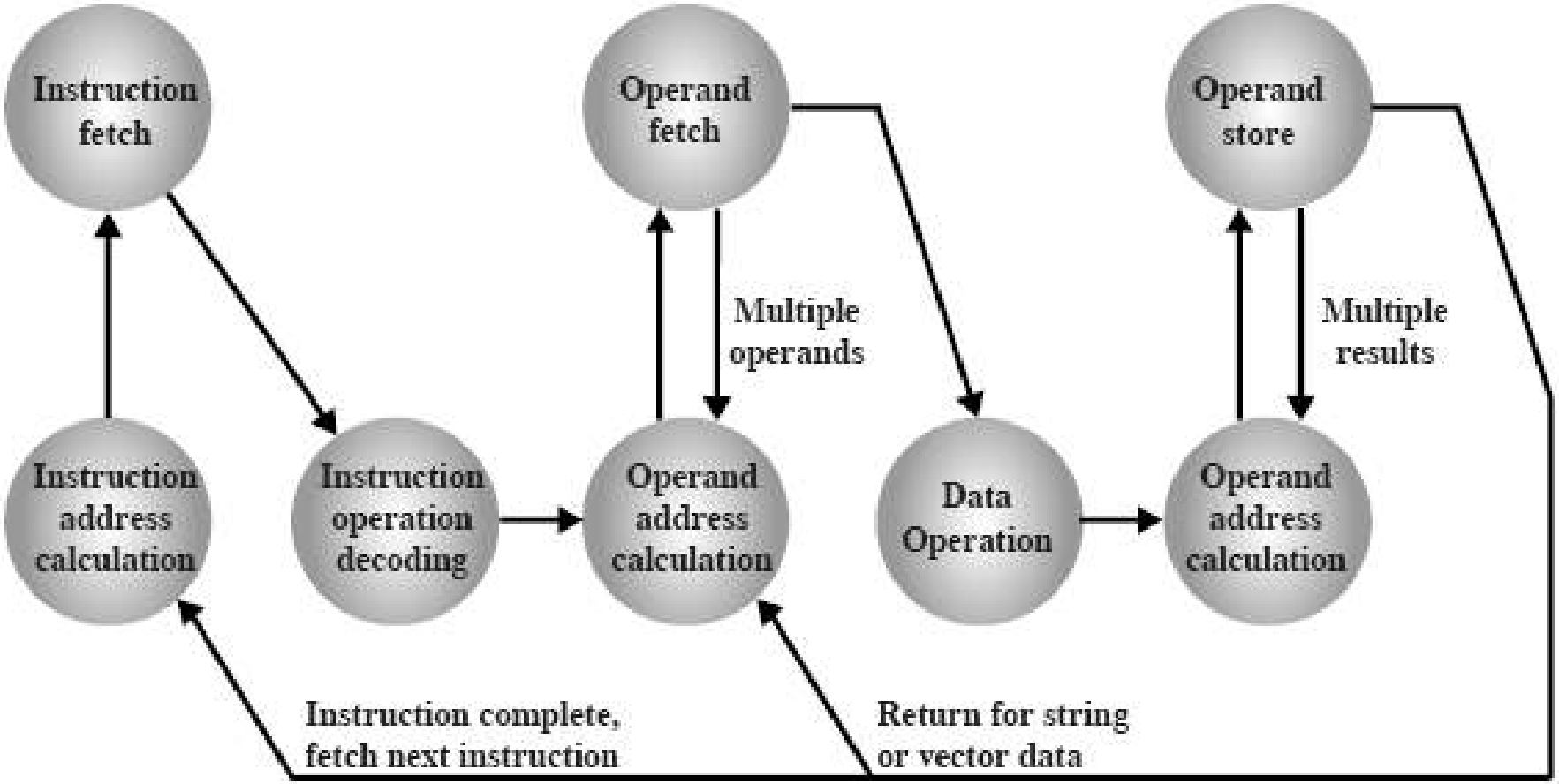

Opcode Fetch Instr #1
Opcode Fetch Instr #2
Opcode Fetch Instr #3
Opcode Fetch Instr #4
Opcode Fetch Instr #5
Opcode Fetch Instr #6
Opcode Fetch Instr #7
Opcode Fetch Instr #8
Decode Instr #1
Decode Instr #2
Decode Instr #3
Decode Instr #4
Decode Instr #5
Decode Instr #6
Decode Instr #7
Operand Fetch Instr #1
Operand Fetch Instr #2
Operand Fetch Instr #3
Operand Fetch Instr #4
Operand Fetch Instr #5
Operand Fetch Instr #6
Exec Instr #1
Exec Instr #2
Exec Instr #3
Exec Instr #4
Exec Instr #5
Retire Result Instr #1
Retire Result Instr #2
Retire Result Instr #3
Retire Result Instr #4

Pipeline

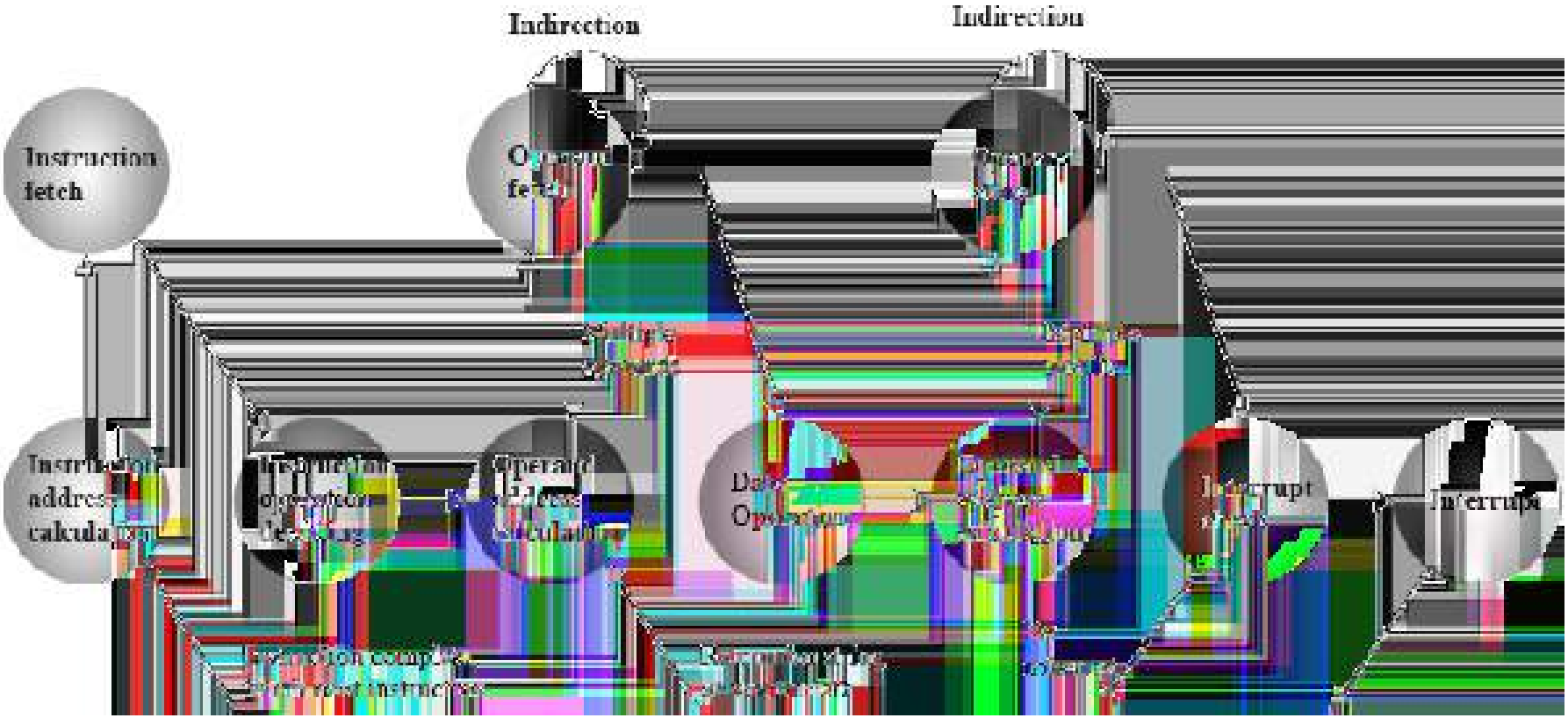
Resultado

- En un pipeline de 5 etapas, y en el caso ideal en que cada etapa consuma solamente un ciclo de clock, una arquitectura pipeline provee un resultado de instrucción por cada ciclo de clock, a partir del ciclo de clock en el cual se llega a resolver la primer instrucción.
- El pipeline tarda en llegar a esa condición de régimen tantos ciclos de clock como etapas tenga.

Deep Pipeline



Deeper



Profundidad de Pipeline. Casos prácticos.

Procesador / uArquitectura	Etapas	Procesador / uArquitectura	Etapas
ARM7TDMI(-S)	3	ARM7EJ-S	5
ARM810	5	ARM9TDMI	5
ARM1020E	6	XScale PXA210/PXA250	7
ARM1136J(F)-S	8	ARM1156T2(F)-S	9
ARM Cortex-A5	8	ARM Cortex-A8	13
AVR32 AP7	7	AVR32 UC3	3
DLX	5	Intel P5 (Pentium)	5
Intel P6 (Pentium Pro)	14	Intel P6 (Pentium III)	10
Intel NetBurst (Willamette)	20	Intel NetBurst (Northwood)	20
Intel NetBurst (Prescott)	31	Intel NetBurst (Cedar Mill)	31
Intel Core	14	Intel Atom	16
LatticeMico32	6	R4000	8
StrongARM SA-110	5	SuperH SH2	5
SuperH SH2A	5	UltraSPARC	9
UltraSPARC T1	6	UltraSPARC T2	8
WinChip	4	LC2200 32 bit	5

Eficiencia de un pipeline

- ¿Porque querríamos aumentar y aumentar el número de etapas?
- Para un tiempo de procesamiento interno de una instrucción ya establecido en una arquitectura “no pipelinezada“, intuitivamente puede comprenderse que cuantas mas etapas podamos definir para ejecutar esta operación, al ponerlas a trabajar a todas en paralelo en un pipeline, el tiempo de ejecución se reducirá proporcionalmente con la cantidad de etapas.

$$TPI = \frac{Tiempo\ por\ instrucción\ en\ la\ CPU\ ''No - Pipeline''}{Cantidad\ de\ etapas} \tag{4}$$

Donde ***TPI*** significa Time Per Instruction

Eficiencia de un pipeline

- Considerar la situación teórica e ideal dada por la ecuación (15) no es 100 % cierto, sin embargo, se aproxima bastante a la situación real.
- En la práctica existen overheads introducidos por el pipeline, que suman pequeñas demoras, pero de todos modos el tiempo se aproxima mucho al ideal.
- El resultado es que la reducción puede apreciarse como si se requiriesen finalmente menos CPI para completar una instrucción.
- En consecuencia, el pipeline no reduce el tiempo de ejecución de cada instrucción individual, sino que incrementa el número de instrucciones completadas por unidad de tiempo. De hecho el overhead del pipeline perjudica el tiempo de ejecución individual, de manera poco significativa, pero agrega tiempo a cada instrucción.
- Sin embargo el rendimiento (throughput) del procesador mejora notablemente ya que los programas ejecutan mucho mas rápido.

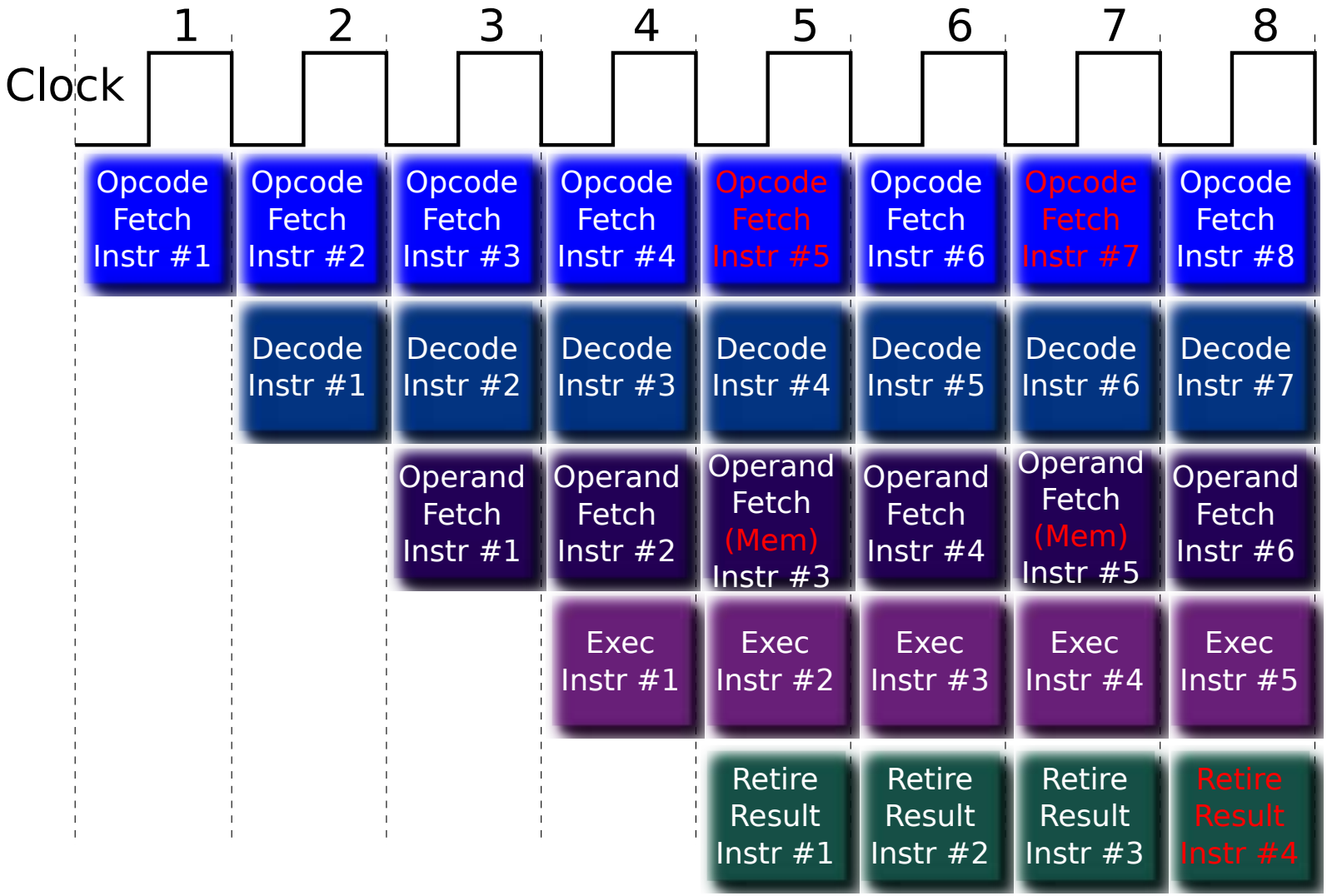
Hazards

- Hay obstáculos que conspiran contra la eficiencia de un pipeline.
- Podemos agruparlos en tres categorías:
  1. ***Obstáculos estructurales***. Existe alguna etapa en el pipeline que no está suficientemente atomizada como para cumplir con los requerimientos de algunas instrucciones, y se produce competencia de recursos en la misma etapa del pipeline.
  2. ***Obstáculos de datos***. No se puede acceder a un dato ya que otra etapa está usando los buses.
  3. ***Obstáculos de control***. Cuando se produce una discontinuidad en el flujo de instrucciones como producto de un branch.
- El efecto se denomina ***pipeline stall***.
- Su efecto consiste en degradar la performance del procesador

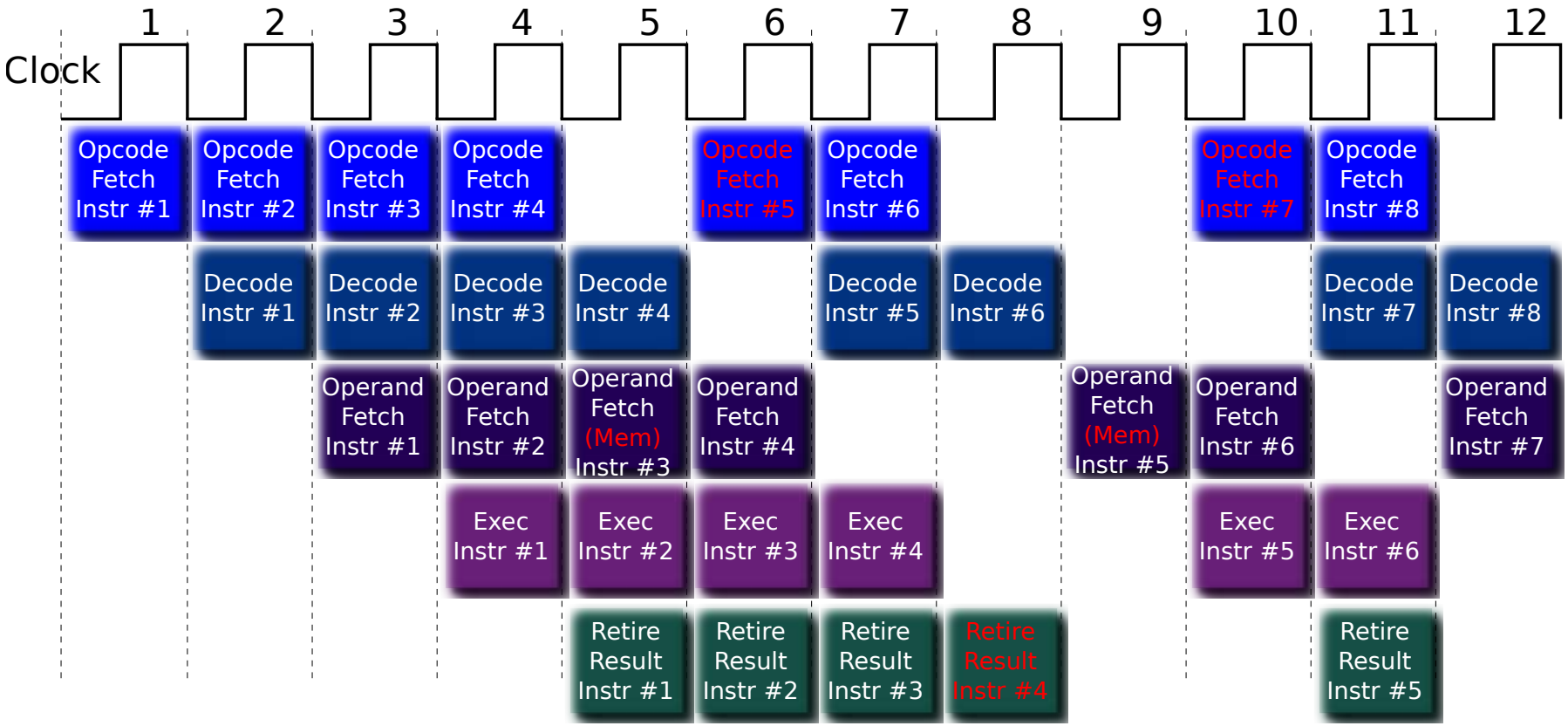
Obstáculos Estructurales

- Surgen cuando una etapa no está suficientemente atomizada, y concentra aún funciones que para algunas instrucciones pueden generar conflicto de recursos para su ejecución.
- Este grupo de instrucciones entonces, no puede pasar por esa etapa del pipeline en un ciclo de clock.
- En estas circunstancias una de las instrucciones debe detenerse. Como consecuencia CPI se incrementa en 1 respecto del valor ideal.
- Ejemplo: Hay procesadores que solo tienen una etapa para acceder a memoria y la comparten para datos e instrucciones. En el caso de que se necesite un operando de memoria, el acceso para traer este operando interferirá con la búsqueda del operando de una instrucción mas adelante del programa.

Obstáculos Estructurales - Accesos concurrentes a memoria



Obstáculos Estructurales - Efecto en CPI



- Por cada Obstáculo, pospone una operación. $CPI = CPI + 1$
- En general la cantidad de CPI que se incrementan es igual a la cantidad de concurrencias menos 1, en el lapso considerado

Obstáculos Estructurales - Posibles soluciones

- Cualesquiera sean las soluciones que deseemos implementar, es necesario agregar hardware.
- En el caso de accesos a memoria, se puede resolver mediante las siguientes opciones por separado o combinadas:
  1. Desdoblamiento de las memorias cache Level 1 en Cache de datos y cache de instrucciones.
  2. Empleo de buffers de instrucciones implementados como pequeñas colas FIFO.
  3. Ensanchamiento de los buses mas allá de los anchos de palabra del procesador.
- Otros tipo de Obstáculos estructurales pueden resolverse aumentando la profundidad del pipeline, lo cual derivará en mas etapas pero mucho mas simples y por lo tanto factibles de resolver en un clock mayor variedad de instrucciones.

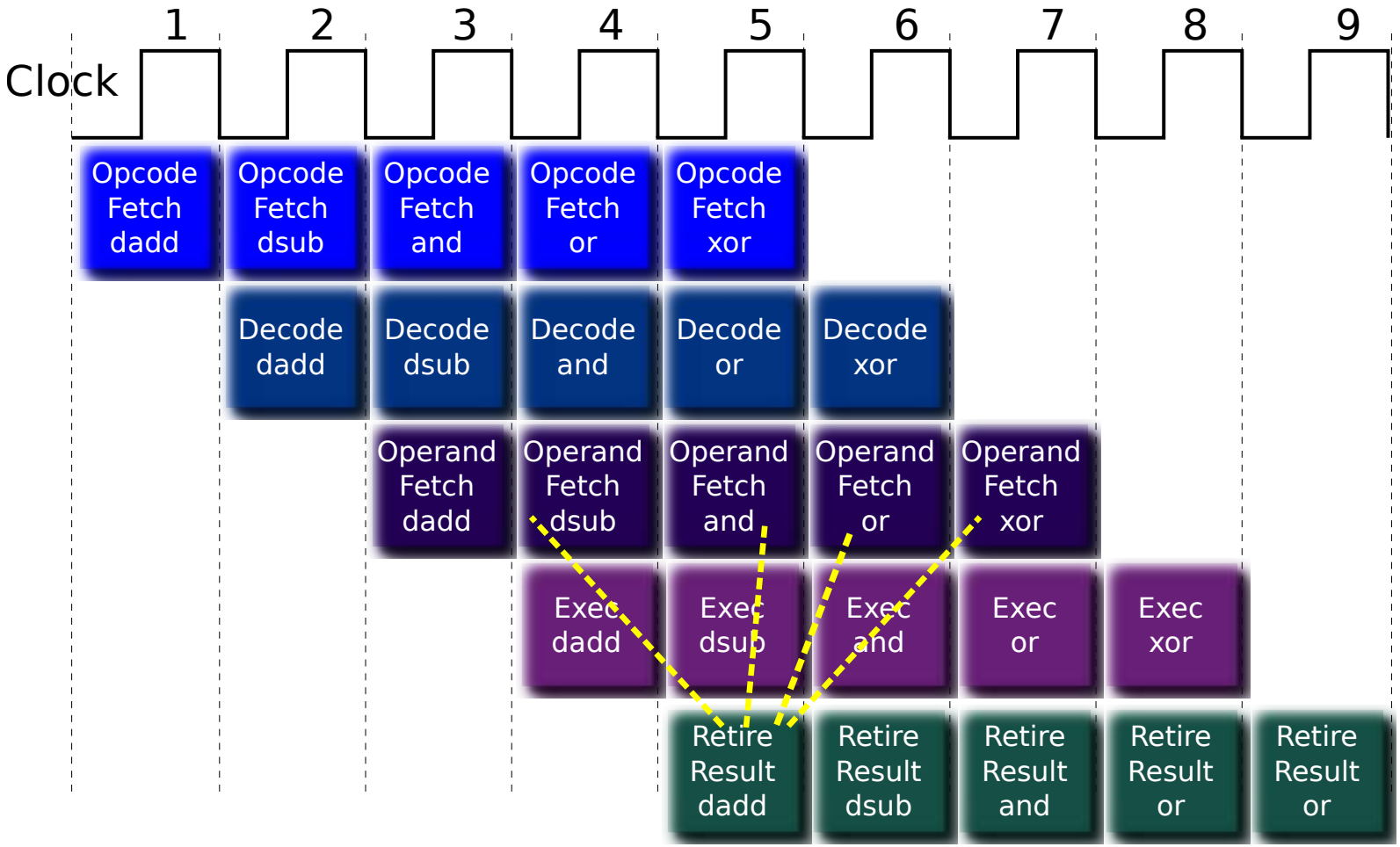
Obstáculos de datos

- Se producen cuando por efecto del pipeline, una instrucción requiere de un dato antes de que este esté disponible por efecto de la secuencia lógica prevista en el programa.
- Consideremos el siguiente código para un procesador MIPS.

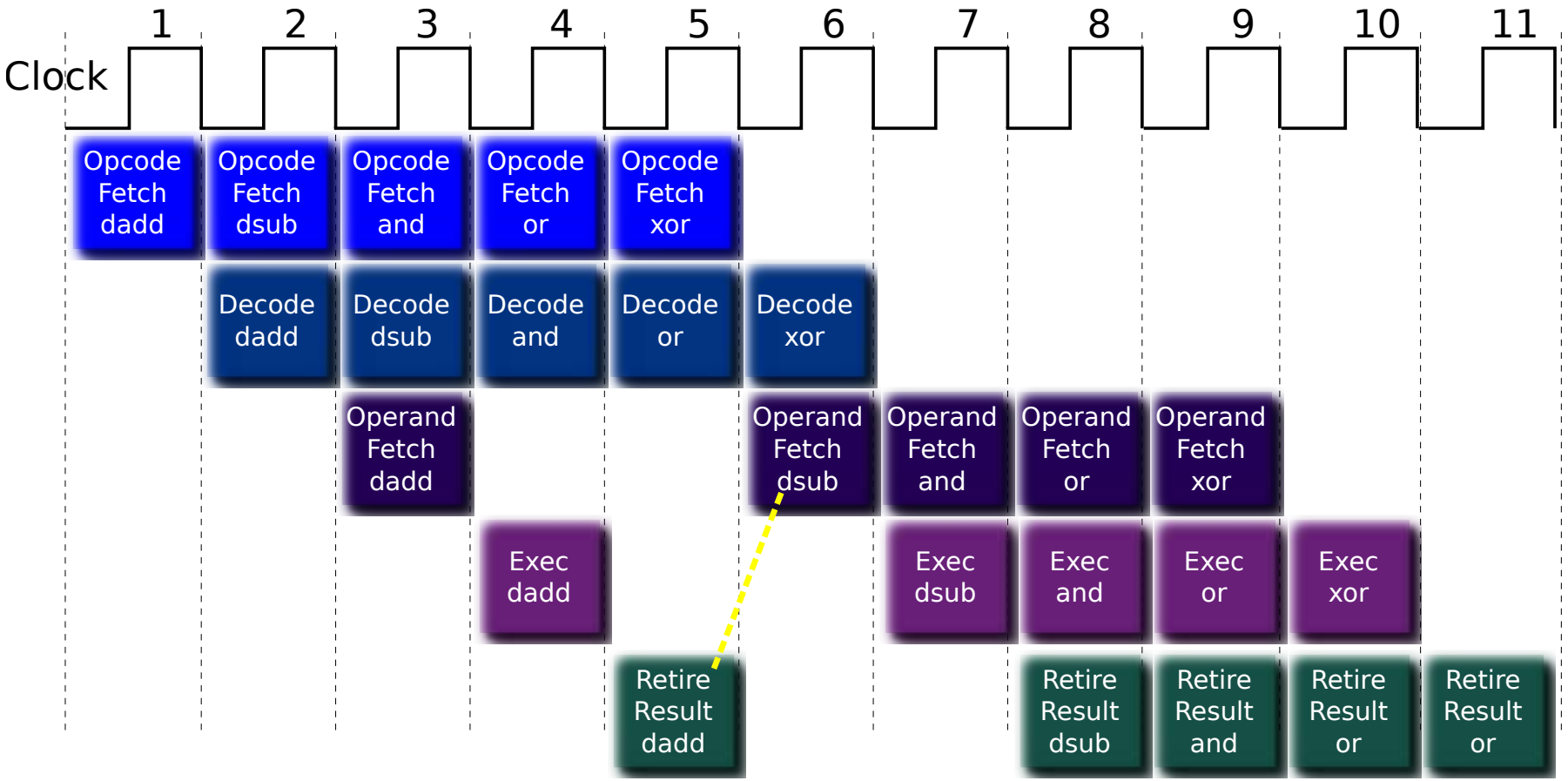
```
1    dadd   R1, R2, R3
2    dsub   R4, R1, R5
3    and    R6, R1, R7
4    or     R8, R1, R9
5    xor    R10, R1, R11
```

- Tenemos dependencias para el Registro R1. Hasta que la instrucción dadd no complete su operación, R1 no tiene un valor válido. Por lo tanto no puede continuar aplicándolo en las restantes que lo utilizan.
- La situación en el pipeline es la siguiente:

Obstáculos de Datos - Conflicto de recursos



Obstáculos de Datos - Efecto en CPI

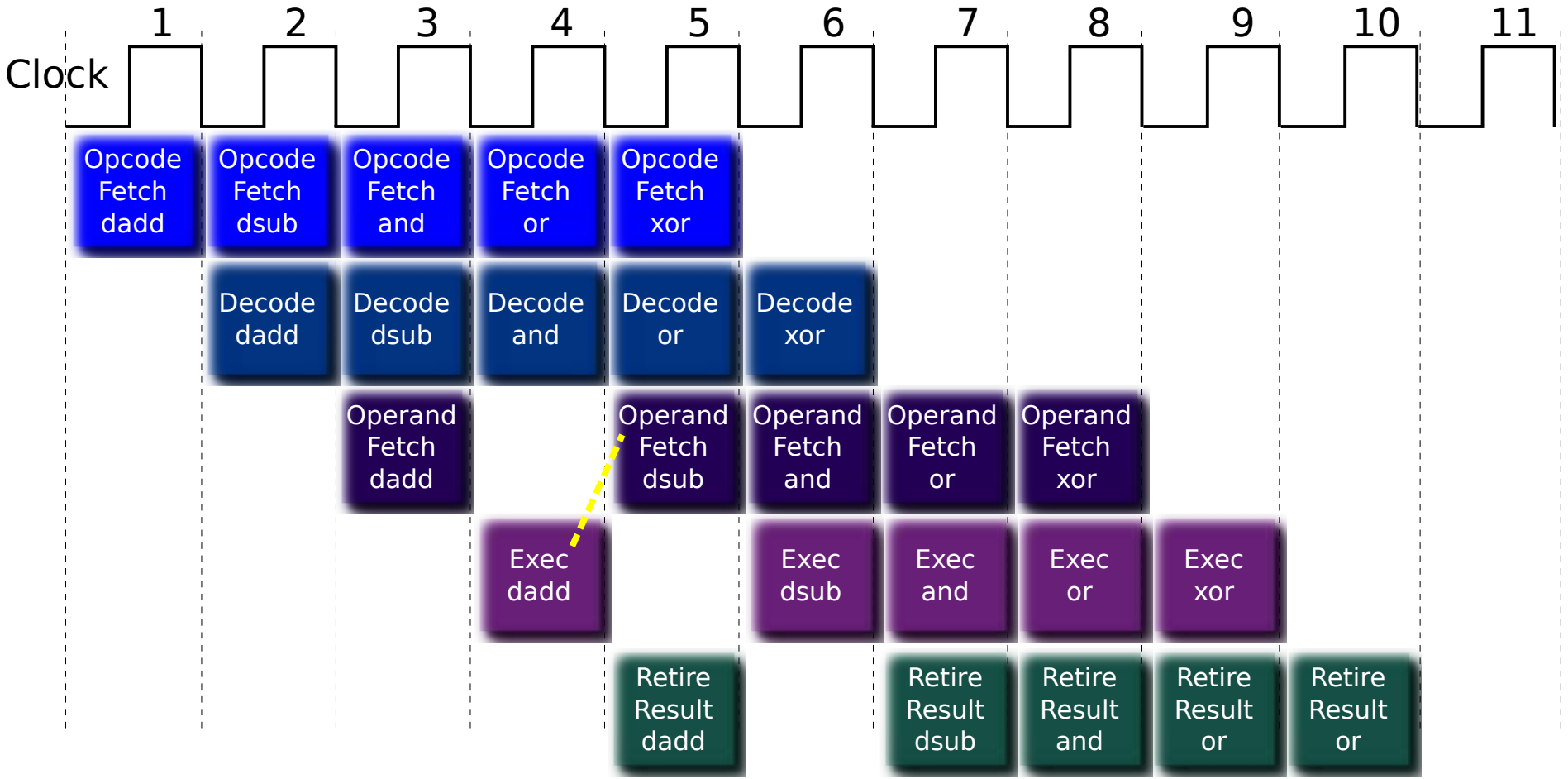


- Por cada Dependencia, pospone una operación. $CPI = CPI + n$, siendo n la distancia entre las etapas del pipeline que requieren el mismo dato.

Obstáculos de Datos - Forwarding

- Para resolver o mitigar el problema de dependencias de datos se aplica un método denominado forwarding
- Consiste en extraer el resultado directamente de la salida de la unidad de Ejecución (ALU, Floating Point, o la que corresponda a la instrucción), y enviarlo a la entrada de la etapa que lo requiere sin que ésta deba esperar que se retire el resultado (es decir que se aplique en el operando destino).
- Se aplica solamente a las etapas posteriores que quedarían en estado stall.
- A aquellas etapas que no quedarían en estado stall como consecuencia de la dependencia de datos, se les envía la salida del resultado cuando este es aplicado en el operando destino.

Obstáculos de Datos - Forwarding



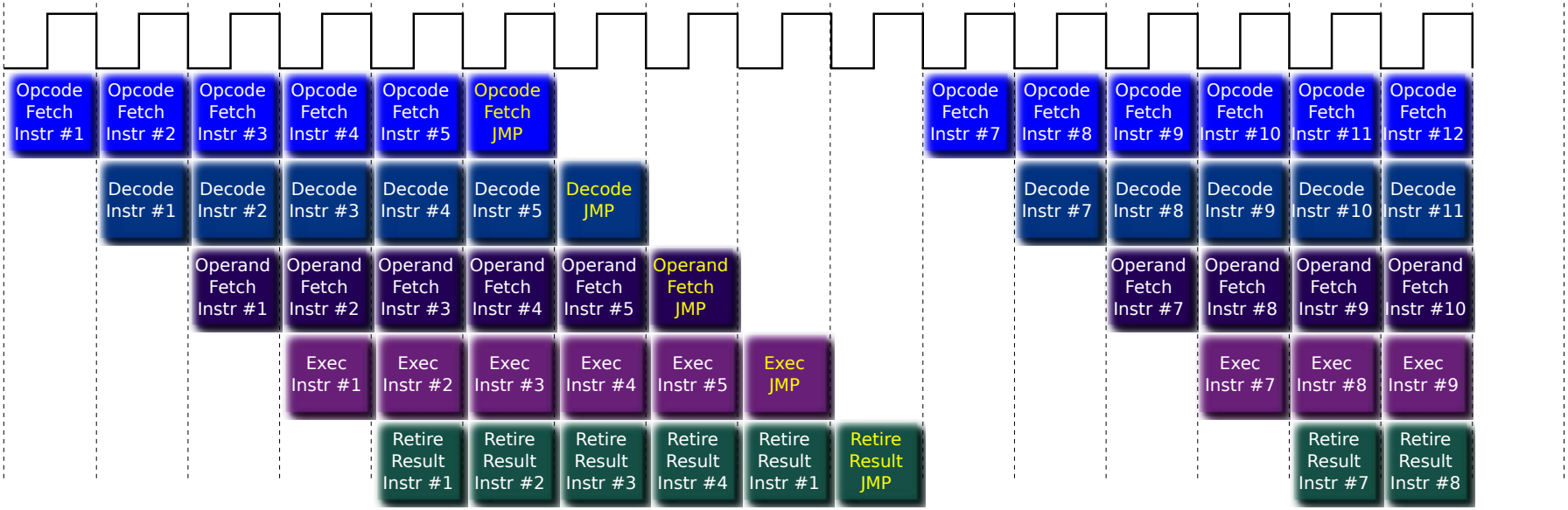
Algunas veces forwarding no es factible

- Normalmente forwarding aplica a operaciones de ALU denominadas back-to-back, ya que el bypass se efectúa desde la ALU a un registro entero del procesador.
- Consideremos ahora este otro código para un procesador MIPS.

```
1    ld      R1, 0(R2)
2    dsub    R4, R1, R5
3    and     R6, R1, R7
4    or      R8, R1, R9
5    xor     R10, R1, R11
```

- En este caso el dato no puede adelantarse hasta que no se complete el ciclo de clock en el que se impacta el resultado dentro del procesador.
- En estos casos se usa una técnica denominada **pipeline interlock**. Consiste en detectada la dependencia de un resultado de una transferencia de memoria se genera un stall en el pipeline a partir de la etapa de ejecución de la instrucción dependiente.

“La conspiración de los branches”



- Un branch es la peor situación en pérdida de performance.
- Un branch es una discontinuidad en el flujo de ejecución.
- El pipeline busca instrucciones en secuencia.
- El branch hace que todo lo que estaba pre procesado deba descartarse. Y el pipeline se vacía debiendo transcurrir ***n -1*** ciclos de clock hasta el próximo resultado. Siendo ***n*** la cantidad de etapas del pipeline. Esto se conoce como **branch penalty**.

saltos, branches, interrupciones, llamadas...

- El principal inconveniente se tiene cuando el salto es condicional, ya que es necesario determinar si la condición es true o false.
- En el primer caso se habla de ***branch taken*** (cambia el registro PC a la dirección de salto), y en el segundo de ***branch untaken*** (PC apunta a la instrucción siguiente al branch)
- En el esquema de pipeline clásico que venimos analizando, esta condición se verifica:
  1. en la fase de ejecución, si es un salto condicional.
  2. en la fase de búsqueda de operando si es un salto incondicional o llamada a subrutina, con direccionamiento indirecto (es decir la dirección de salto está en la memoria en la dirección que contiene la instrucción)
  3. o, en el mejor de los casos en la fase de decodificación si es un salto incondicional o llamada a subrutina con direccionamiento directo (es decir la dirección de salto viene a continuación del código de operación).
- En las interrupciones la situación es la del gráfico anterior.

Efectos de los branches y como neutralizarlos

- Forwarding puede ayudar a disminuir el efecto de los diferentes casos expuestos anteriormente. Sin embargo, no es óptima, ya que solo lograremos disminuir algunos ciclos de clock del **branch penalty**, pero no se puede eliminar del todo su efecto.
- Para soluciones mas eficientes es necesario recurrir a análisis mas pormenorizados, que en general tienen en cuenta el comportamiento de los algoritmos y de los saltos.
- Ingresamos al universo de las unidades de predicción de saltos. Las hay desde muy simples a muy sofisticadas, y tiene que ver no solo con las diferentes generaciones de procesadores, sino también con el tipo de procesador bajo análisis.

predicted-non-taken

- El procesador asume por default que el salto nunca se toma, es decir que continúa la búsqueda del código de operación de las instrucciones siguientes a la de salto como si el salto fuese en realidad una instrucción común y corriente.
- Funciona adecuadamente en estructuras de programa como la siguiente:
  instrucción *branch*
  instrucción sucesora secuencial.
  instrucción destino para *branch taken*.
- Es decir, cuando el salto es “hacia adelante”, o bien cuando la dirección del target es mayor que la de la dirección de memoria que contiene la instrucción de salto.

predicted-taken

- El procesador asume por default que el salto se toma siempre, es decir que continúa la búsqueda del código de operación de las instrucciones a partir de la dirección target.
- Funciona adecuadamente en estructuras de programa como la siguiente:
  instrucción destino para *branch taken*.
  grupo de instrucciones a ejecutar iterativamente.
  instrucción *branch*
- Resulta de gran utilidad en casos de estructuras de iteración.
- La gran ventaja es que siempre se acierta. Solo falla cuando expira la condición del lazo.

¿Que criterio tomar?

- Cual de los dos criterios adoptar, depende del diseño del set de instrucciones.
- El compilador, conociendo el criterio asumido por el procesador, debe organizar el código de la manera mas adecuada para aprovechar el criterio elegido por el diseñador de Hardware.

delayed branch

- En los primeros modelos de procesadores RISC se introdujo un enfoque superador denominado salto demorado (delayed brach).
- Volviendo a la estructura de programa como la siguiente:
  instrucción *branch*
  instrucción sucesora secuencial.
  instrucción destino para *branch taken*.
- La *instrucción sucesora secuencial* se envía al slot de salto demorado.
- Se ejecuta si o si independientemente del resultado de la evaluación del branch.
- Se aplica o no el resultado dependiendo del resultado de la evaluación de la condición. En este caso no genera demoras. En caso de branch-taken, se descarta la ejecución y se tiene un ciclo de clock para que salga el resultado de la instrucción destino.

Loop Unrolling en el compilador

- Los compiladores son hasta ahora los responsables de armar los loops de modo de usar las instrucciones en función del branch Prediction.
- Otra posibilidad es desenrollar los branches, técnica conocida como loop unrolling.
- Para implementarla es necesario, que los datos dentro del loop sean paralelizables. Esto es por ejemplo el siguiente código:

```
1     for (i = 0 ; i < 256 ; i++ )
2     {
3         suma = 0.0f; /*  */
4         for (j = 0 ; (j <= i && j < 256) ; j++)
5             suma += v0[i-j] * v1[j];
6         fAux[i] = suma;
7     }
```

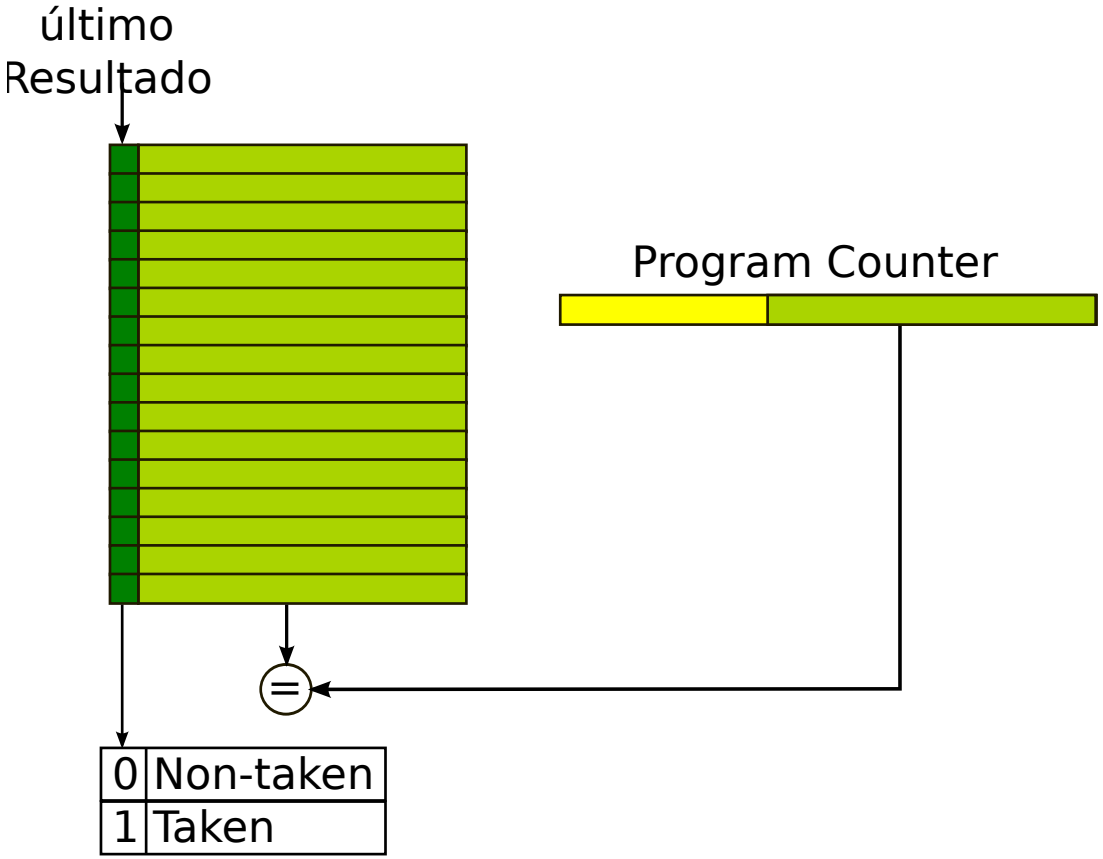
- La ventaja de deshacer el loop y hacer una instrucción detrás de la otra es que se eliminan los branches que componen cualquier loop y desaparece la penalización.

Predicción de saltos dinámica

- Todos los métodos vistos hasta aquí dependen exclusivamente del compilador. Esto significa que el hardware interno del procesador no realiza ningún análisis del código ni agrega adaptatividad.
- Cuando el procesador comienza a efectuar un análisis del flujo de instrucciones y toma decisiones en función de lo que encuentra se tiene un paso adelante en la predicción de saltos.
- Ingresamos al universo de la predicción dinámica.
- Los métodos subsiguientes son de esta categoría y corresponden a microarquitecturas mas avanzadas, con mayor paralelismo a nivel de instrucciones.

Branch Prediction Buffer

- Es una tabla simple indexada por la dirección de memoria de la instrucción de salto (o su campo de bits menos significativos), y un bit que indica simplemente el resultado reciente del salto (*taken* o *non-taken*)



Branch Prediction Buffer

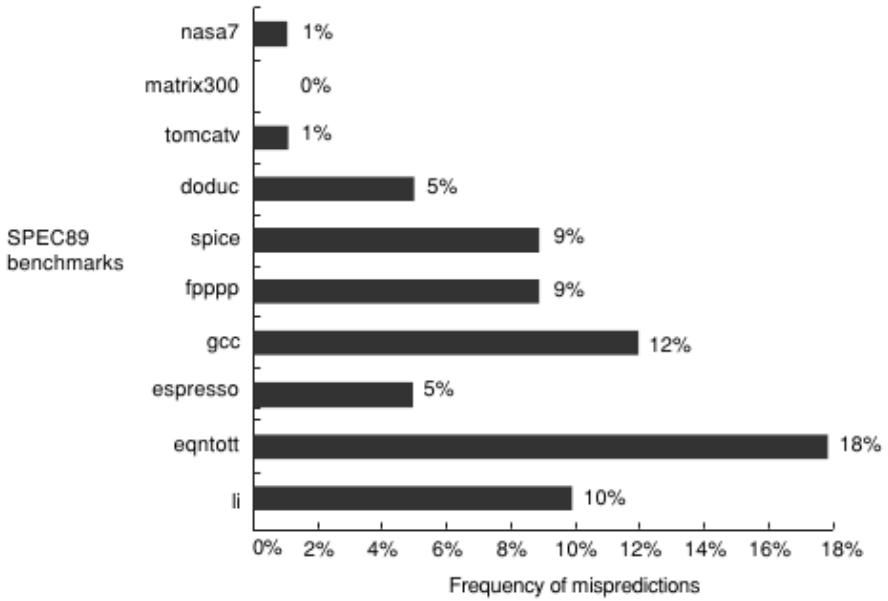
- En realidad el valor del bit es solo una pista para la Unidad de Ejecución.
- No reviste seguridad en la predicción.
- Hasta incluso puede tratarse de una etiqueta ingresada por otra instrucción de salto cuya dirección finaliza con el mismo patrón de bits que la actual.
- Si el resultado del salto coincide con el valor almacenado en el Branch Prediction Buffer, se mantiene el contenido, y si falla se complementa a 1.
- Este modelo se llama predicción simple de 1 bit. Limitación: cuando un salto siempre resulta *taken*, y falla una vez produce dos predicciones fallidas seguidas, ya que el bit se invierte.
- Para mejorar su rendimiento se implementó un modelo de 2 bits, el cual es mas eficiente en su predicción ya que cumple un diagrama de estados como el siguiente.

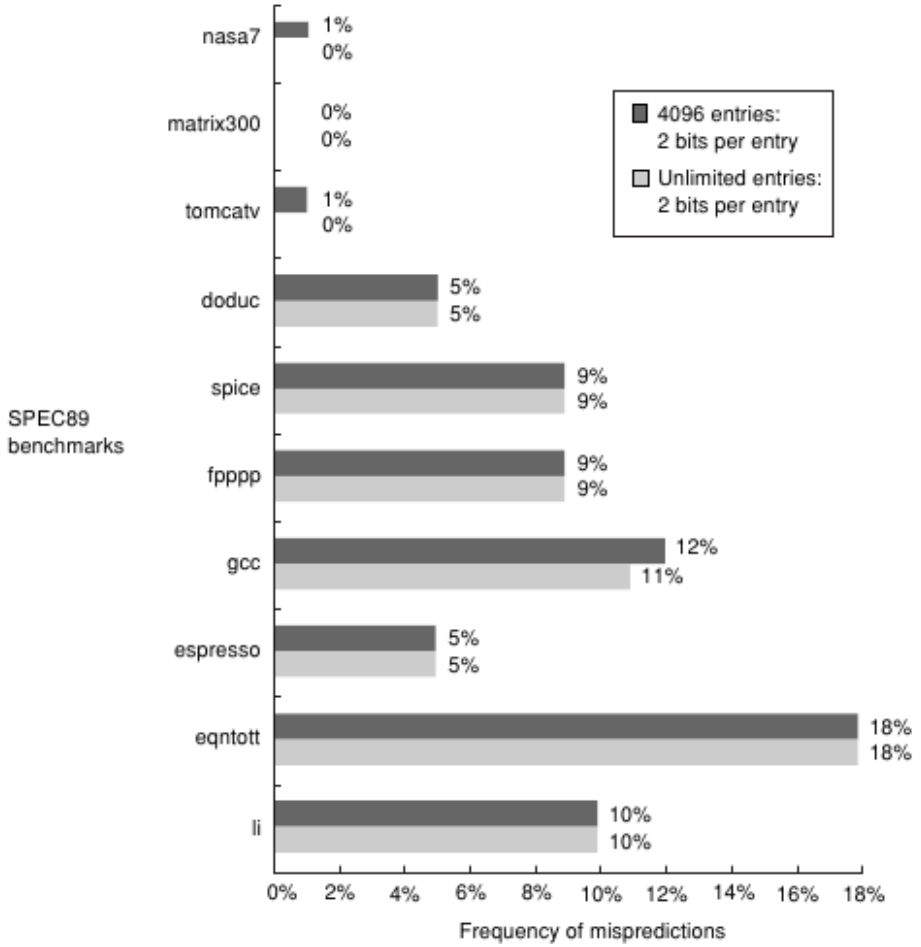
Predicción de 2 bits

Implementación práctica

- En la práctica, este tipo de Branch Predictor, se implementa en la etapa de Instruction Fetch del pipeline, como un pequeño cache de direcciones de salto accesible mediante las direcciones de las instrucciones.
- Otra forma de implementación es agregar un par de bits a cada bloque de líneas en el cache de instrucciones, que se emplean únicamente si ese bloque tiene instrucciones de salto condicional.
  - En el caso en que la predicción para la instrucción de salto de ese bloque sea *taken*, el Program Counter se setea con la dirección destino del salto y se continúa buscando instrucciones a partir de allí.
  - En otro caso se sigue buscando las instrucciones en secuencia.

Eficiencia. Benchmarks SPEC89





Performance Branch Predictor de 2 bits: Conclusiones

- El método tiene una eficiencia superior al 82 %, para cualquier tipo de programa
- La eficiencia es superior en programas de punto flotante (*missprediction rate* $< 4\,\%$) frente a los programas de cálculo entero ($4\,\% <$ *missprediction rate* $< 18\,\%$)
- El tamaño del buffer de predicción no genera efecto en la eficiencia mas allá de los 4Kbytes.
- Tampoco se obtuvieron mejoras aumentando la cantidad de bits de predicción mas allá de 2. En general un branch predictor de n bits tomaría valores entre 0 y $2^n - 1$, tomando el salto para los valores contenidos en la mitad mas ata del rango y no lo tomaría para la mitad menos significativa. La complejidad en el diseño no se ve compensada por la mejora en la funcionalidad de predicción.

Predicción por Correlación

- En el siguiente bloque de código:

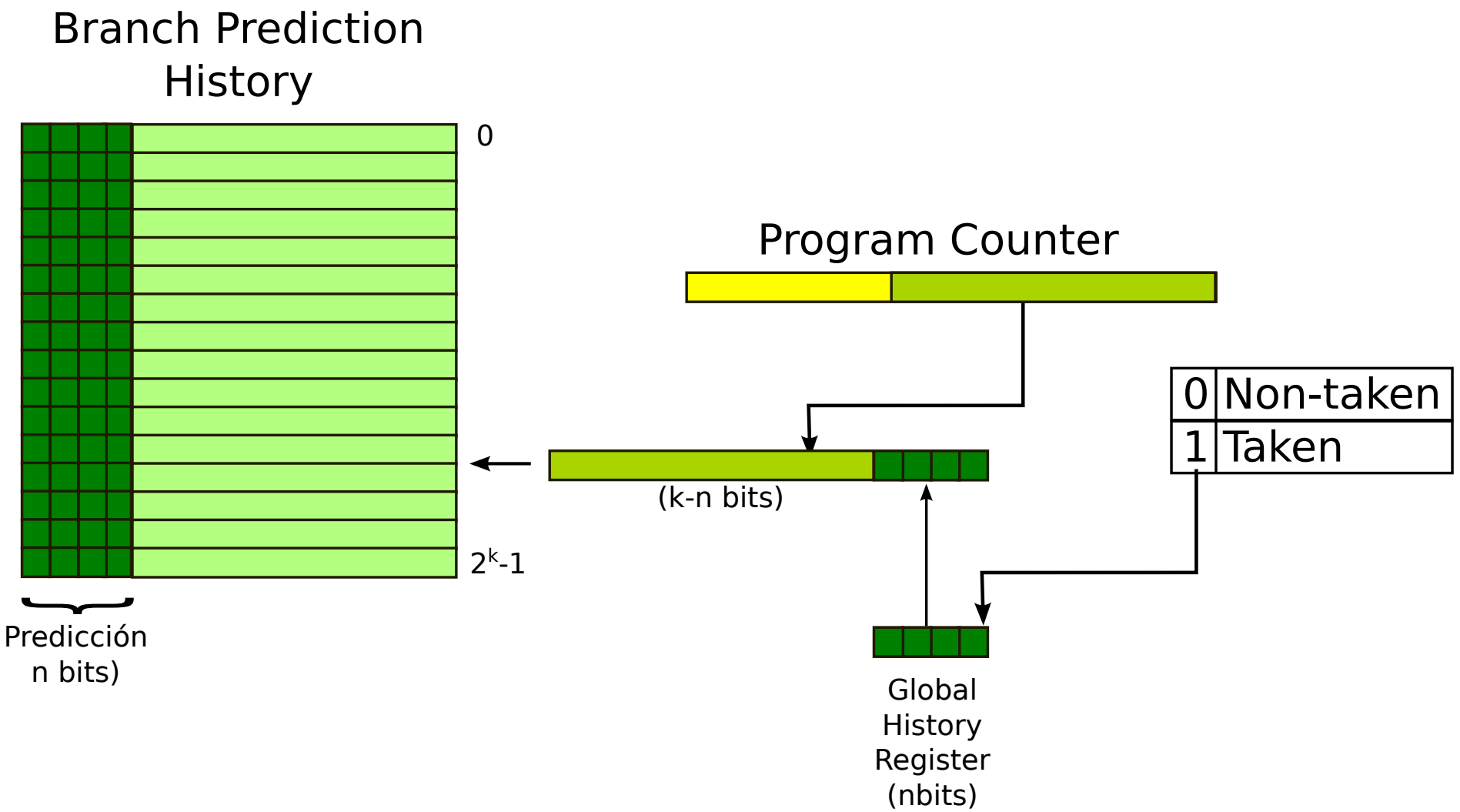
```
1    if (i == 2)
2        i = 0;
3    if (j == 2)
4        j = 0;
5    if (i == j)
```

- Puede verse claramente que el resultado de la tercer sentencia de decisión está correlacionado con los resultados de las dos sentencias de decisión previas.
- Si los branches 1 y 2 resultaron no taken (es decir las dos condiciones fueron true), el tercer branch resultará taken.
- Un predictor de saltos clásico no mira este tipo de comportamiento.
- Un predictor de saltos con correlación (o predictor en dos niveles) estudia los resultados de los saltos recientes para analizar su eventual efecto sobre el que está bajo análisis.

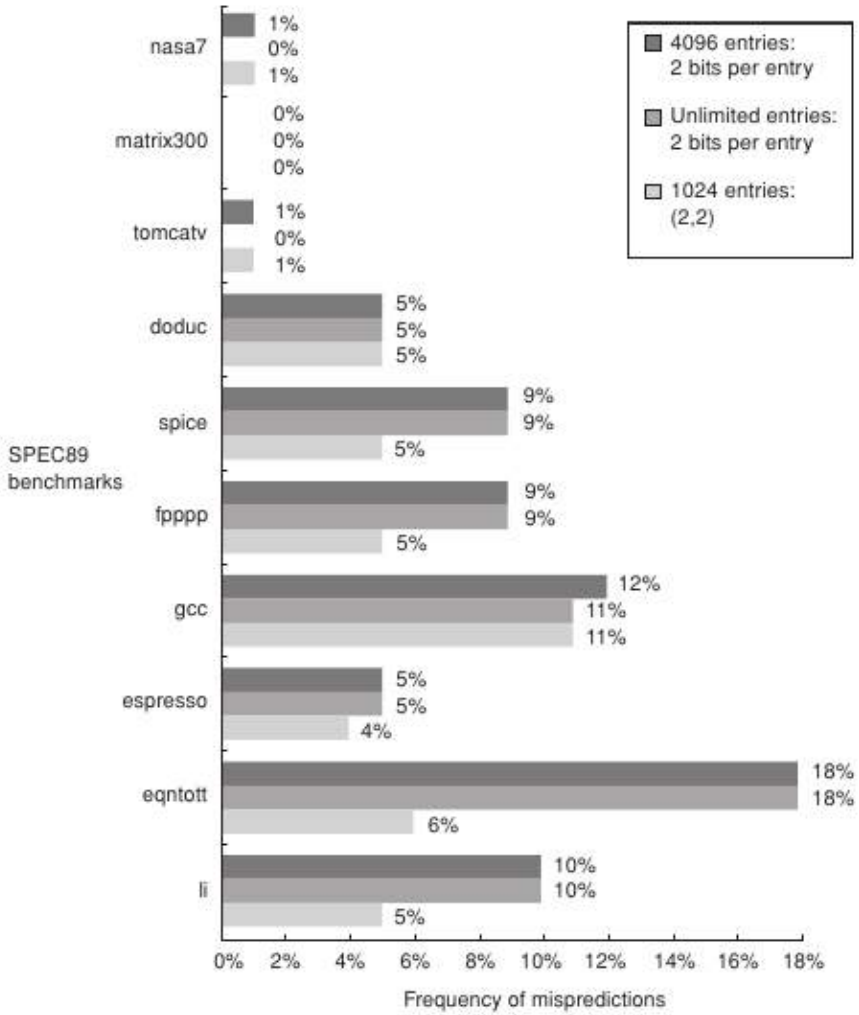
Predicción por Correlación

- Un predictor de saltos (m,n) se compone de 2^m predictores de n bits que analizan los m branches previos y en función los resultados toman la predicción.
- Los resultados son mejores que el predictor de 2-bits y el hardware adicional es insignificante.
- En un predictor (m,n), tomamos un shift register de m bits y se almacena la historia global de los últimos m branches.
- Cada bit indica si el resultado fue *taken* o *Non-taken*.
- Entonces para un buffer de 2^k entradas, se concatenan los k-m bits menos significativos de la dirección que contiene la instrucción de salto, con los m bits del shift register, para seleccionar la posición que contiene la predicción del salto.

Predicción por Correlación

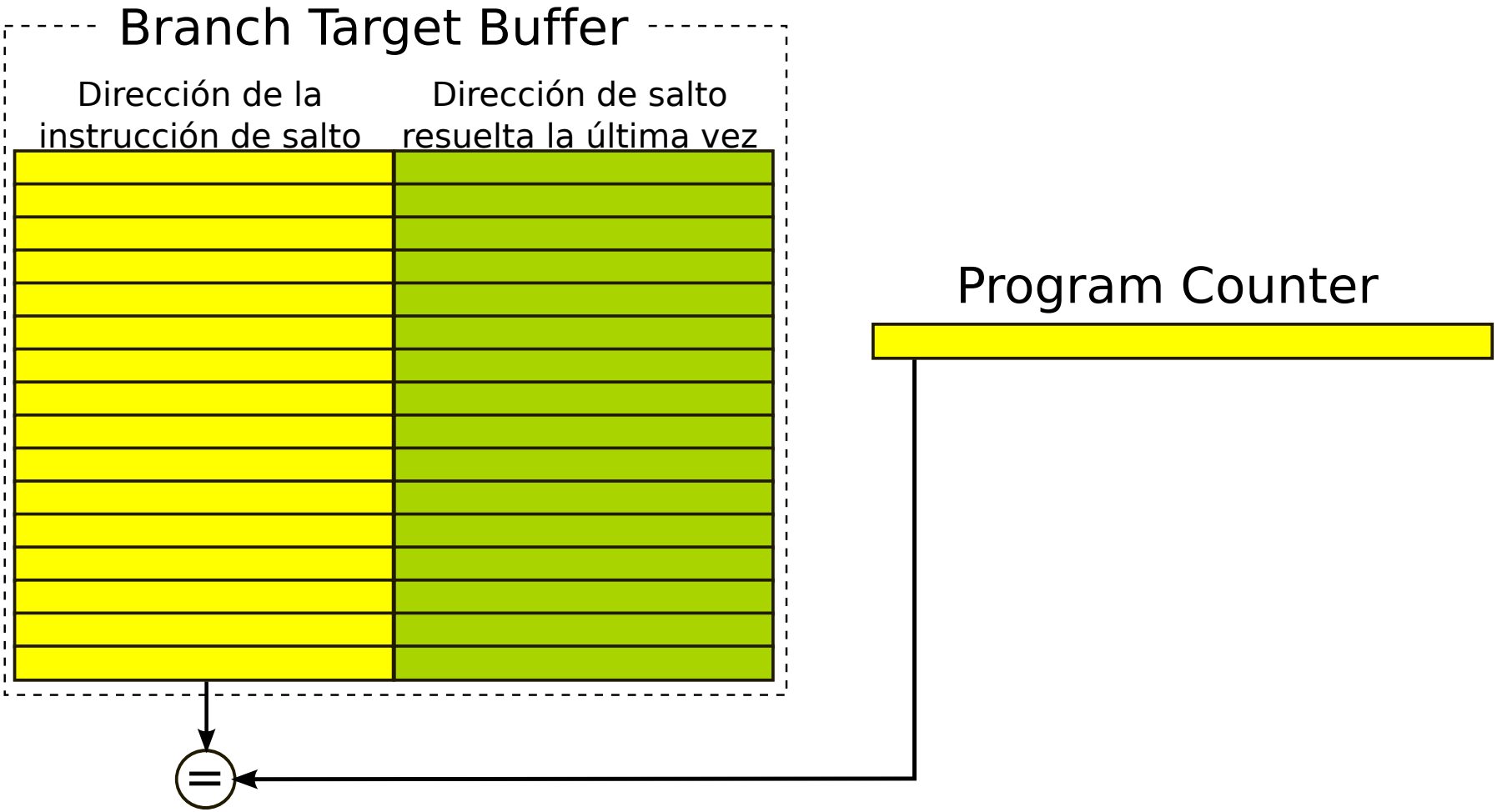


Eficiencia. Benchmarks SPEC89



Branch Target Buffer

- Es un cache de instrucciones de salto que contiene para cada entrada el par dirección de la instrucción de salto, y dirección del target resuelta (No los Bits *taken* o *Non-taken*).



Branch Target Buffer

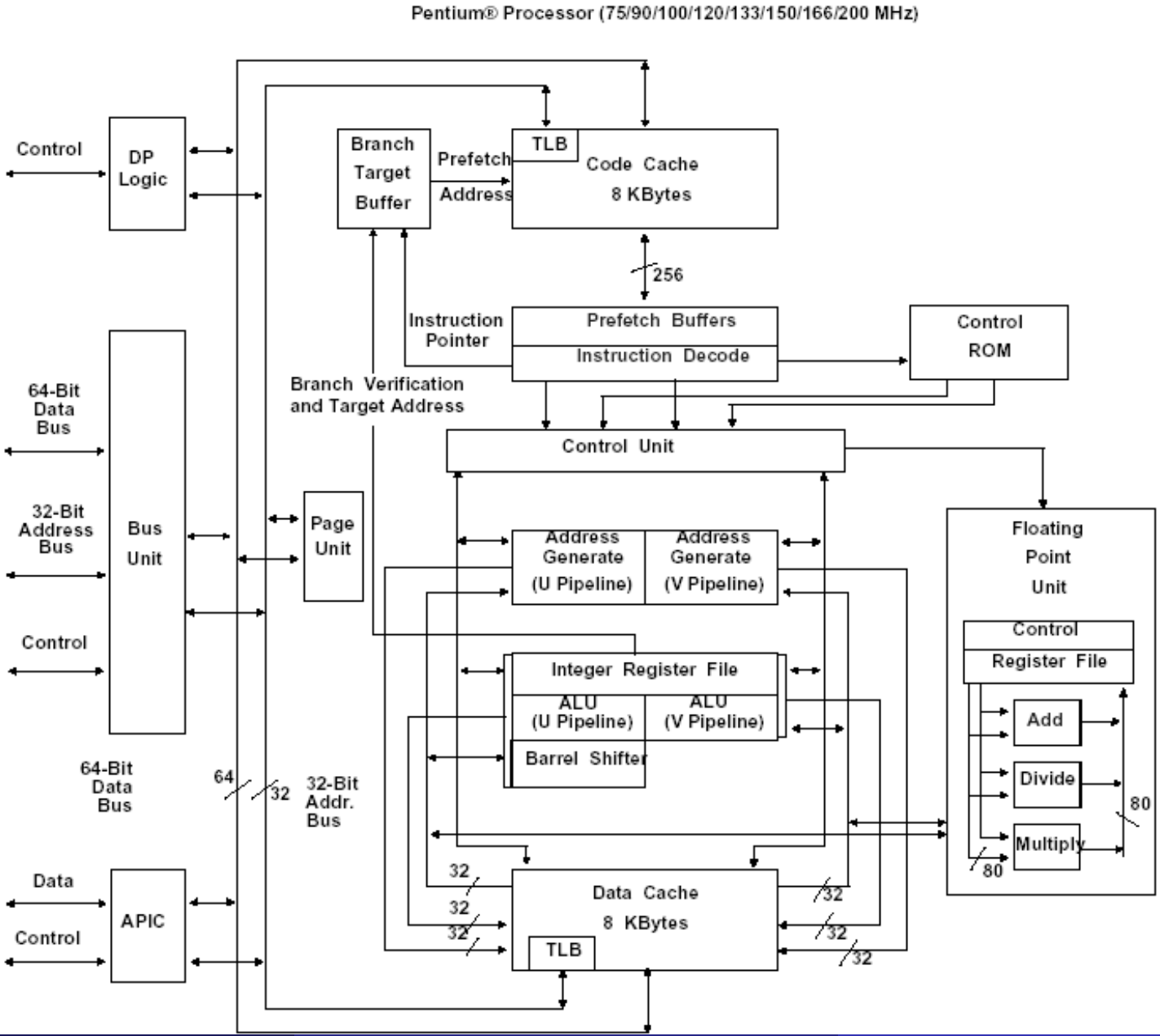
- Se trabaja como una memoria de acceso por contenido.
- Se accede mediante el valor completo del Program Counter.
- Si el valor no se encuentra se asume *taken*.
  - Si el resultado es *Non-taken* se acepta el delay en el pipeline, y no se almacena nada en el BTB
  - Si el resultado es *taken*, Se ingresa el valor al BTB.
- Si el valor se encuentra en el BTB, se aplica el campo de dirección target almacenado
  - Si el resultado es *taken*, no hay penalidad, y no se guarda en el BTB ningún nuevo valor ya que el que está almacenado es el que nos sirve.
  - Si el resultado es *Non-taken*, guarda el nuevo valor en el BTB ya que el que está no sirvió, luego de la penalidad correspondiente en el pipeline.

Mas allá de 1 Instrucción por ciclo de clock...

- Podemos trabajar a nivel de Microarquitectura y Hardware, para tratar de aumentar la paralelización jugar con el clock y violar algunas leyes de la física, para llegar a que el pipeline pueda obtener una eficiencia de liberar un resultado por ciclo de clock.
- ¿Como seguimos?
- Simple... tratando de ejecutar mas de una instrucción por ciclo de clock.
- El problema es ¿como?.
- La respuesta es... ILP (Instruction Level Paralelism).
- ¿Como? ¿De que venimos hablando entonces hasta ahora?

Superscalar de dos vías

Un Superscalar de dos vías: El pentium



Mas ILP, aumenta los obstáculos

- Los obstáculos estructurales quedan mas expuestos. Recordando el hazard de acceso simultáneo a memoria, ahora cada etapa debe lidiar con las otras etapas de su propio pipeline que pueden acceder en simultáneo para acceder a instrucciones o datos, y además con las mismas etapas del otro pipeline. La probabilidad de accesos concurrentes aumenta con la cantidad de vías del Superscalar.
- Ahora se pueden ejecutar en dos ALUs dos instrucciones. Si una depende del resultado de la otra, esto no es posible.
- Una falla en el branch prediction... es letal. Limpia ambos pipelines!

Obstáculos de Datos. Aumento conforme se avanza

- Hasta aquí estudiamos pipelines con scheduling de instrucciones estático.
- Estos pipelines simplemente buscan una instrucción y la envían a la unidad de ejecución.
- En caso de tener dependencias con otra instrucción en el pipeline, el envío es demorado.
- En caso de superescalares el estudio de dependencias se debe extender a instrucciones en los diferentes pipelines.
- Estos y otros casos que veremos, vuelven insuficiente el método de Forwarding aplicado en los primeros pipelines.
- Hay que ir por mas.

Idea Fuerza

- Si una instrucción *j*, depende una instrucción *i*,que ,para completarse requiere de varios ciclos de clocks adicionales debido a obstáculos de tipo estructural, o de datos, todas las instrucciones que siguen a a instrucción *j* no pueden ser ejecutadas.
- Esta situación genera una obstrucción del pipeline, dejando en estado idle a todas las unidades funcionales que compongan la Unidad de Ejecución del pipeline.
- Para ilustrarlo consideremos el siguiente código para un procesador MIPS.

```
1    div.d     F0,F2,F4
2    add.d     F10,F0,F8
3    sub.d     F12,F8,F14
```

- La instrucción ***div.d***, lleva varios ciclos de clock para completarse, lo cual obstruye la ejecución de la instrucción ***add.d*** ya que utiliza el registro F0, en donde se almacenará el resultado de la instrucción previa.
- La instrucción ***sub.d***, no guarda dependencias de datos con las previas, y queda esperando que se complete la instrucción responsable del atasco.

Ejecución Fuera de Orden

- Esta idea fuerza consiste en tratar de enviar las instrucciones a ejecución independientemente del orden en el que están en el código. Esta decisión se puede tomar una vez decodificada la instrucción, ya que allí se sabe si hay un atasco de datos o estructural que haga que la instrucción que se comience a ejecutar pueda impida el envío de sus dependientes próximas.
- Cada vez que lo intentemos pueden aparecer riesgos los cuales deben ser evaluados para no incurrir en errores.
- Para ilustrarlo consideremos el siguiente código para un procesador MIPS.

```
1    div.d    F0,F2,F4
2    add.d    F6,F0,F8
3    sub.d    F8,F10,F14
4    mul.d    F6,F10,F8
```

- La instrucción ***div.d*** demora varios ciclos y atasca el envío de la instrucción ***add.d***.
- Es posible enviar las instrucciones ***sub.d*** y ***mul.d***, aunque deben tomarse en cuenta que hay riesgos. ¿Cuales son?

Riesgos WAR y WAW

- En el bloque de código anterior, la instrucción 2 se encuentra demorada de envío por dependencia de la instrucción 1.
- Así que se envían las instrucciones 3 y 4, (considerando ya un procesador superescalar). Esta situación conlleva dos riesgos potenciales:
  1. La Instrucción 3 escribe su resultado en un operando de la Instrucción 2
  2. La Instrucción 4 escribe su resultado en el destino de la Instrucción 2
- La situación planteada en 1, se conoce como riesgo **WAR**, por **W**rite **A**fter **R**ead, y representa el riesgo que como consecuencia de la ejecución de la instrucción 3, ésta escriba (**Write**) el registro F8, y después (**After**) de ello la instrucción 2 lo leerá (**Read**), obteniendo un dato incorrecto y alterando el resultado de la secuencia de código generada originalmente por el programador.
- La situación planteada en 2, se conoce como riesgo **WAW**, por **W**rite **A**fter **W**rite, y representa el riesgo que como consecuencia de la ejecución de la instrucción 4, ésta escriba (**Write**) el registro F6, y después (**After**) de ello la instrucción 2 lo escribirá (**Write**), eliminando el resultado de la instrucción 4 y nuevamente alterando el resultado de lo que el programador determinó.
- Resta el que se conoce como **RAW**, **R**ead **A**fter **W**rite, que si bien no está presente en el ejemplo anterior, es el mas común. Se presenta cada vez que se lee un operando que después es escrito por una instrucción dependiente.

Manejo de excepciones en Ejecución fuera de orden

- No solo se necesita preservar el comportamiento de las instrucciones de modo que los resultados aparezcan en el mismo orden en el que ocupan las instrucciones en el programa, también en manejo de las excepciones debe preservar el comportamiento de modo que resulte idéntico al que se tendría como resultado de procesar en el orden que establece el programa.
- Al ejecutar fuera de orden, el procesador puede generar lo que se denomina *excepciones imprecisas*, que son aquellas que al producirse, el estado del procesador no es el mismo que debería ser si las instrucciones se hubiesen ejecutado en orden.
- Hay dos posibles motivos:
  1. El pipeline ha completado la ejecución de una o mas instrucciones posteriores a la que produce la excepción.
  2. El pipeline no ha completado aún al menos una instrucción previa a la que genera la excepción.
- En resumidas cuentas, el procesador debe asegurar que no se levante una excepción hasta no tener completado todo el programa incluida la instrucción que es responsable de la excepción.

Ejecución fuera de Orden

Implementación

Para implementar un procesador con Ejecución Fuera de Orden necesitamos dividir la etapa de Decodificación de instrucciones en dos sub-etapas

- La etapa de **Envío**, trabaja *en orden*, decodificando las instrucciones y enviándolas a la unidad de ejecución. Si encuentra un obstáculo estructural se detiene hasta que se resuelva el obstáculo. La unidad de Prebúsqueda continúa igualmente, hasta que se llene el buffer de prebúsqueda. Si esto ocurre se detiene (stall).
- La etapa de **Lectura de Operandos**, inicia la operación *fuera de orden*, ya que es capaz de saltear en el buffer aquellas instrucciones que tienen bloqueos de Datos. Envía a ejecutar aquellas cuyos operandos pueden ser accedidos. Y permanece revisando cuando se resuelven los obstáculos de datos para enviar esa instrucción a ejecución-

Ejecución fuera de Orden

Unidad de Prebúsqueda

Se encarga de buscar instrucciones ya sea desde un buffer de instrucciones o desde el cache L1 interno en los procesadores mas actuales. Los coloca en una cola FIFO, mediante un índice para determinar cuando la cola está completa.

La clave de Ejecución Fuera de Orden

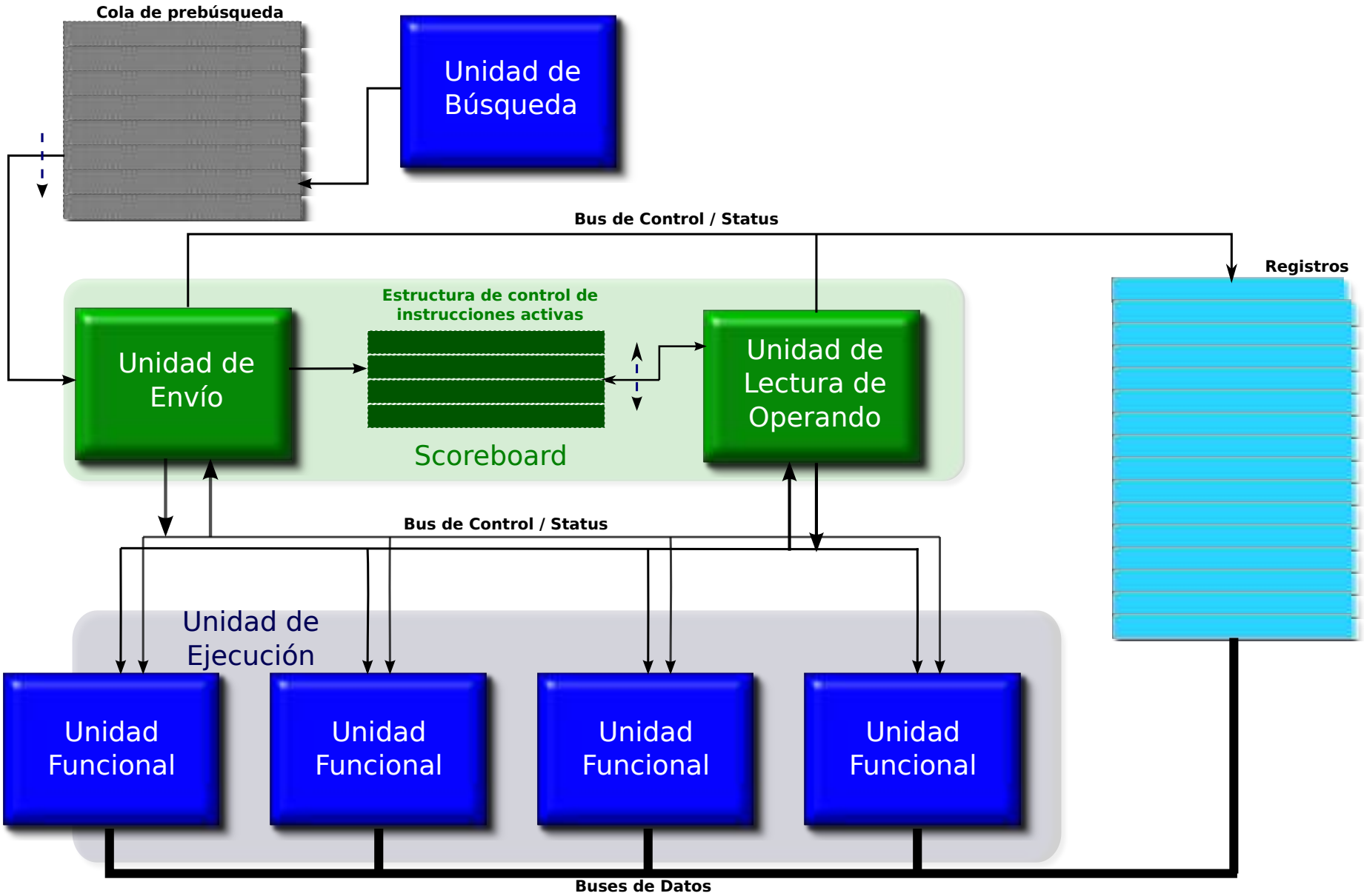
- Para poder implementar este modelo es necesario poder ejecutar varias instrucciones al mismo tiempo (mas allá de un modelo superescalar).
- Si podemos identificar el momento de inicio y finalización de una instrucción podemos definir que la instrucción entre esos momentos se encuentra *en ejecución*.

Scoreboarding

- Es el método mas sencillo (y menos sofisticado) para implementar Ejecución Fuera de Orden evitando los riesgos asociados (**WAR** y **WAW**)
- Su primer implementación fue en la CDC 6600, instalada entre otros laboratorios en la Universidad de Berkeley en 1964. Control Data Corporation, construyó este computador basado en transistores de germanio, con algunas innovaciones interesantes en su diseño:
  - Un juego de instrucciones muy sencillo en contraposición a los sets complejos de instrucciones que poseían hasta ese momento las restantes CPUs, estableciendo los cimientos de los actuales procesadores RISC.
  - Capacidad de ejecutar fuera de orden.
  - Amplia paralelización en la Unidad de Ejecución: 4 unidades de Punto Flotante, 5 Unidades de Referencias a Memoria, y 7 ALUs para enteros.

La CDC 6600. . . aunque Ud. no lo crea.(©Wikipedia)

Scoreboarding: Diagrama General



Scoreboarding - Una vuelta mas...

Unidad de Envío

- Si hay libre una Unidad Funcional que pueda ejecutar la instrucción que se termina de decodificar, y si además no hay ninguna otra instrucción activa (es decir en proceso de decodificación, espera de operando, ejecución o lo que sea), que necesite el mismo operando destino, entonces envía a la unidad de ejecución la instrucción en cuestión.
- A posteriori actualiza la estructura de datos de control interna.
- De este modo se asegura que no existen riesgos **WAW**.
- Si aparece algún Bloqueo Estructural o se detecta un riesgo **WAW**, entonces la Unidad de Envío se Bloquea (Stall), y no envía ninguna otra instrucción hasta que la actual se limpien los riesgos y bloqueos pendientes.
- Con la Unidad de Envío Stalled, la Unidad de Prebúsqueda sigue buscando instrucciones hasta llenar la cola de prebúsqueda.

Scoreboarding - Una vuelta mas...

Unidad de Lectura de Operando

- El scoreboard monitorea para cada instrucción la disponibilidad de cada operando.
- Un operando está disponible cuando ninguna de las instrucciones activas va a escribirlo.
- Cuando verifica que una instrucción tiene todos sus operandos libres, el scoreboard envía una señal a la Unidad Funcional que tiene la instrucción, y comienza su ejecución.
- De este modo asegura que no existan riesgos **WAR**.
- Por supuesto que la ejecución se puede realizar antes de otras instrucciones activas pero que ocupen lugares previos en la secuencia de programa. Esto es fuera de orden.
- Junto con la Unidad de Envío, reemplazan a la Unidad de Decodificación de un pipeline clásico.

Scoreboarding - Una vuelta mas. . .

Unidad de Ejecución

- Completa la ejecución de la instrucción ante el aviso de la Unidad de Lectura de Operando y avisa al scoreboarding para que la instrucción se ha completado.
- El resultado está en un registro temporario a la salida de la Unidad Funcional.
- Equivale a la Unidad de Ejecución de un pipeline clásico.

Unidad Escritura de Resultado

- Una vez que la Unidad Funcional envió la señal al scoreboard el aviso de ejecución de la instrucción completado, el scoreboard se asegura de que no exista un riesgo **WAR** antes de escribir el resultado en el operando destino.
- Si detecta riesgos bloquea (stall) a la unidad de Escritura de Resultados.

Limitaciones del Scoreboard

- Aparecen nuevos Obstáculos estructurales debido a que la cantidad de buses es limitada para paralelizar las transferencias entre el scoreboard y los registros.
- Los operandos se leen directamente en los registros, y por lo tanto no se aprovecha la técnica de Forwarding (adelantar el operando desde el registro de salida de la unidad funcional que ejecutó la instrucción.

Scoreboarding - Ejemplo

- Consideremos el siguiente código para un procesador MIPS.

```
1    l.d      F6,34(R2)
2    l.d      F2,45(R3)
3    mul.d    F0,F2,F4
4    sub.d    F8,F6,F2
5    div.d    F10,F0,F6
6    add.d    F6,F8,F2
```

- Vamos a determinar el comportamiento de un scoreboard en las tres tablas que se muestran en los slides siguientes.
  1. **Estado de Instrucciones**: Contiene todas las instrucciones enviadas o pendientes de envío, e indica en que etapa está cada una.
  2. **Estado de Unidades Funcionales**: Una vez enviada la instrucción se mantiene el estado de la unidad funcional, utilizando 9 campos:
     - **Busy**. indica si la Unidad Funcional está ocupada o libre.
     - **OP**. Operación que se va a realizar en la Unidad funcional
     - **Fi**. Registro Destino
     - **Fj, Fk**. Registros con Operandos Fuente
     - **Qj,Qk**. Unidades Funcionales que producen los valores de Fj y Fk
     - **Rj,Rk**. Flags que indican si Fj y Fk están listos o no para se leídos. Luego de leídos se ponen en NO.
  3. **Estado del Registro de Resultados**: Indica que unidad escribirá en que registro.

Scoreboarding - Ejemplo

Algunas consideraciones complementarias.

- Procesador MIPS (Para tomar un caso sencillo)
- Se dispone de una unidad de ejecución paralelizada en las siguientes Unidades Funcionales:

  2 Unidades de Multiplicación de Punto Flotante 10 ciclos de clock de latencia para ejecutar.

  1 Unidad de División de Punto Flotante 40 ciclos de clock de latencia para ejecutar.

  1 Unidad de Suma de Punto Flotante 2 ciclos de clock de latencia para ejecutar.

  1 Unidad de enteros

Estado de Instrucciones al inicio

		Instruction status			
Instruction		Issue	Read operands	Execution complete	Write result
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				

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Integer			Add	Divide			

Riesgos

Retomemos el ejemplo de código:

```
1    l.d      F6,34(R2)
2    l.d      F2,45(R3)
3    mul.d    F0,F2,F4
4    sub.d    F8,F6,F2
5    div.d    F10,F0,F6
6    add.d    F6,F8,F2
```

Los riesgos que identificamos son:

RAW Desde el segundo L.D hacia MUL.D, SUB.D, ADD.D; desde MUL.D hacia DIV.D; y desde SUB.D hacia ADD.D.

WAR Desde ADD.D hacia SUB.D y DIV.D.

Estructural En la unidad de suma para ADD.D y SUB.D.

Entonces, cuando se ejecuta MUL.D, las tablas quedan como en el siguiente slide, y cuando se ejecuta DIV.D las tablas quedan como en el segundo siguiente slide.

Justo antes que MUL.D escriba su resultado

	Instruction status			
Instruction	Issue	Read operands	Execution complete	Write result
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	√	√	√	

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult 1			Add		Divide			

Justo antes que DIV.D escriba su resultado

	Instruction status			
Instruction	Issue	Read operands	Execution complete	Write result
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	√	√	√	√

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Divide	Yes	Div	F10	F0	F6			No	No

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU						Divide			

Approach de Tomasulo

- Proviene de un trabajo de implementación de scheduling dinámico en la unidad de Punto Flotante de la IBM 360/91.
- Su interés es minimizar los riesgos **RAW**, e implementar Register Renaming en los **WAR** y **WAW**.
- Para comprenderlo consideremos el siguiente código para un procesador MIPS.

```
1   div.d     F0,F2,F4
2   add.d     F6,F0,F8    # WAR hazard por F8 con sub.d
3   s.d       F6,0(R1)
4   sub.d     F8,F10,F14
5   mul.d     F6,F10,F8   # WAW hazard por F6 con add.d
```

- Un escenario para la solución sería disponer de dos registros, que denominaremos S y T que permitan tomar datos y utilizarse para su acceso a partir de ese momento.

Approach de Tomasulo

- El código anterior con la inclusión de estos dos registros auxiliares queda como se indica a continuación, libre de dependencias:

```
1    div.d     F0,F2,F4
2    add.d     S,F0,F8    # WAR hazard por F8 con sub.d
3    s.d       S,0(R1)
4    sub.d     T,F10,F14
5    mul.d     F6,F10,T   # WAW hazard por F6 con add.d
```

- Además cualquier uso posterior de F8 es reemplazado por el registro temporario.
- El análisis del código de manera de “mirar mas adelante” de cada instrucción en busca de estos riesgos, demanda gran sofisticación (y consecuentemente una mayor complejidad) en el compilador. Eventualmente también algún soporte de hardware.
- Además el compilador puede recurrir a saltos para resolver estas interdependencias. De modo que puede resultar mas perjudicial la solución que el problema.

La solución: Renaming de registros y la Reservation Station

- Se utilizan bancos o archivo de registros internos agrupados en uno o mas bloques que se denominan Reservation Station. Cada registro posee un tag para identificarlo.
- La Reservation Station que se encarga de buscar los operandos ni bien estén disponibles almacenándolos en estos registros internos.
- A medida que se emiten instrucciones, por cada operando pendiente se renombra el registro que lo contiene a un registro de la Reservation Station.
- De este modo no hay posibilidad de que una instrucción cuya ejecución se adelanta respecto de otra previa en la secuencia del programa, pueda modificar o utilizar un registro de la instrucción previa y que ésta luego use una copia incorrecta del mismo.
- Una vez disponible el operando, la Reservation Station se encarga de su búsqueda y aplicación en cuanto registro destino lo necesite.
- Si un operando destino recibe múltiples escrituras, a Reservation Station solo aplicará la última.
- Si se incluye en la Reservation Station registros en cantidad suficientemente superior a los registros de la arquitectura se puede potencialmente eliminar los riesgos estudiados

Implementación

Implementación

Para implementar el modelo de Tomasulo se requiere de dos bloques en el pipeline que trabajen de la siguiente manera:

- La etapa de **Envío**, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Si los operandos no están disponibles en los registros, la etapa de **Envío** rastrea cuales son las unidades de Ejecución que los producirán y actualiza sus estructuras internas con esta información.
- Esto pone fin a potenciales riesgos WAR y WAW, ya que los registros se renombran.
- Hay quienes llaman a esta etapa ***Dispatch***.

Implementación

Implementación

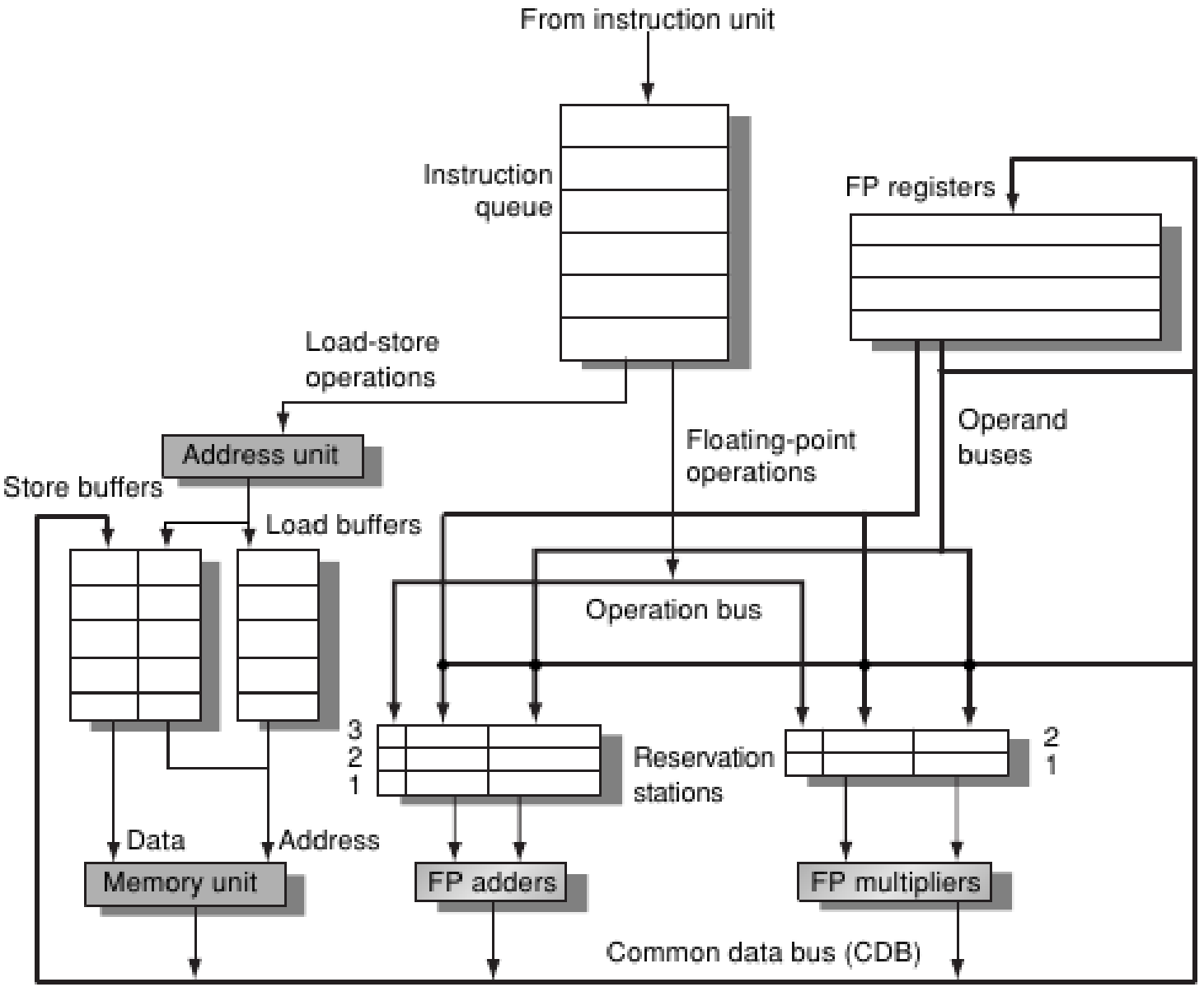
- La etapa de **Ejecución**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Puede ocurrir que en un ciclo de clock determinado se tornen disponibles varios operandos que destraban sendas instrucciones.
- Si bien hay múltiples unidades de ejecución puede ocurrir que la cantidad de instrucciones de un tipo determinado superen la cantidad de Unidades de Ejecución disponibles.
- En general el orden en que se ejecutan es arbitrario, ya que los registros están renombrados. Esto es estrictamente cierto para todo tipo de instrucciones excepto por las de carga y almacenamiento en memoria.
- Las instrucciones de carga y almacenamiento se ejecutan siempre en dos pasos: Calculo de la dirección efectiva de memoria y acceso al operando. Este último paso puede demorar varios ciclos de clock de acuerdo a la tecnología de la memoria que se accede.
- Ejecutar en orden estas operaciones permite evitar riegos de memoria.

Implementación

Implementación

- No se permite ejecutar instrucciones hasta que hayan sido resueltos todos los branches previos.
- La etapa de **Escritura de Resultado**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Si se registra una excepción como resultado de la ejecución de una instrucción, se guarda este registro pero no se genera la excepción sino hasta que se entre en la etapa de escritura del resultado de la instrucción.

Detalle de lo visto



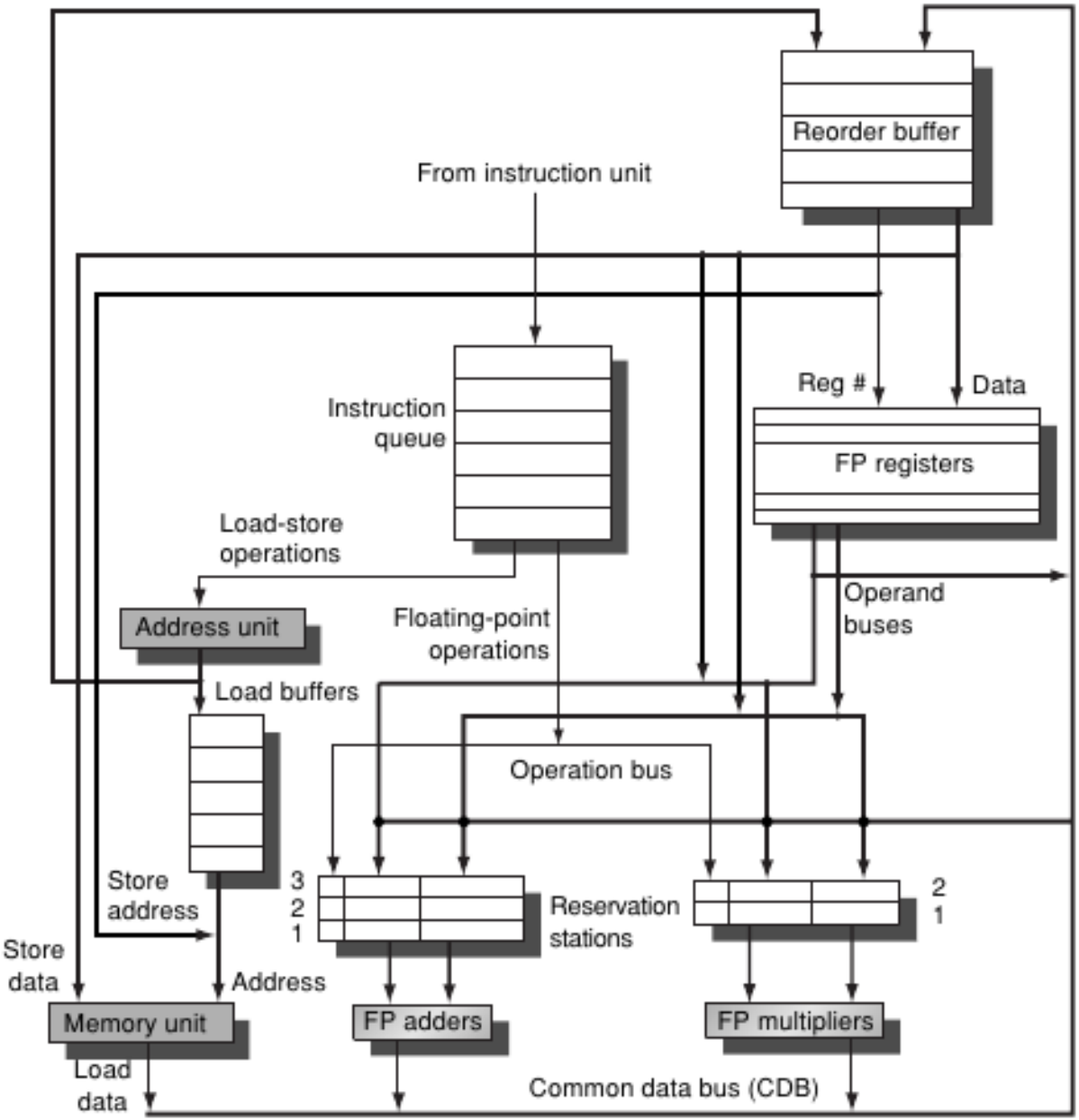
La conspiración de los branches (el regreso)

- A medida que avanzamos en incrementar el paralelismo a Nivel de Instrucciones, el control de las dependencias del programa se complica proporcionalmente.
- Los Branch Target Buffers fueron útiles para evitar stalls en las instrucciones de salto, pero a medida que la ejecución fuera de orden se fue sofisticando considerablemente, es necesario repensar su funcionamiento.
- En general ejecución especulativa es la capacidad de una arquitectura para ejecutar instrucciones sin tener aún los resultados de sus dependencias, sino simplemente asumiendo que el resultado será uno determinado, y lo mas importante, tener la capacidad de deshacer la operación si la especulación no fue correcta.
- Una vez que se tiene el resultado la instrucción deja de ser especulativa y con esta certeza está en condiciones de escribir en el registro destino.

Reordenando los resultados

- Este tipo de ejecución especulativa hace que se tengan pre almacenados resultados de instrucciones posteriores que luego deben impactarse en sus operandos destino
- El commit de los resultados debe hacerse en orden.
- Esta es la función del ReOrder Buffer (ROB).
- Igual que la Reservation Station de Tomasulo, el ReOrder Buffer agrega registros en los cuales se van almacenando los resultados de las instrucciones ejecutadas en base a especulación por haardware.
- El resultado permanecerá en el ROB desde que se obtenga el resultado hasta que se copie (commit) en el operando destino.
- La diferencia con el algoritmo de Tomasulo, es que éste ponía el resultado en registro directamente, y a partir de entonces el resto de las instrucciones lo tenía disponible. El ROB no lo escribe, sino hasta el commit. Y durante ese lapso que al especular se puede extender, el registro de la arquitectura no tiene el valor.

ReOrder Buffer (ROB)



Estructuras asociadas al ROB

Cada entrada del ROB contiene la siguiente información.

Tipo de Instrucción Indica si la instrucción es un branch (y no tiene resultado destino), un memory store (en cuyo caso se requiere calcular la dirección del operando), o una instrucción cualquiera de la ALU en cuyo caso el resultado irá a parar a un registro de la arquitectura.

Destino Contiene el número de registro de la arquitectura en donde se guardará el resultado (En el caso de memory loads u otra operación de ALU cualquiera sea), o la dirección de memoria en caso de ser un memory store

Valor Almacena el resultado de la operación hasta el commit.

Ready Indica que la instrucción se ha completado y su resultado está disponible.

Implementación

Para implementar especulación por hardware se requieren de los siguientes bloques en el pipeline:

Envío

- La etapa de **Envío**, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía, y un slot en el ROB. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, o no hay un slot disponible en el ROB, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Se actualizan las estructuras internas con la información pertinente.

Implementación

Ejecución

- Si faltan operandos monitorea por el Bus de datos interno del procesador hasta que éstos estén disponibles.
- Chequea riesgos RAW.
- Ejecuta las operaciones en las que tiene los operandos en la RS.
- Hay instrucciones que pueden demorar varios ciclos de clock

Write Result

- Escribe el resultado en el slot correspondiente del ROB, una vez disponible.
- Si alguna RS espera el resultado, también lo escribe en el registro correspondiente.
- En el caso de un memory store escribe el resultado en el campo valor del registro correspondiente del ROB

Implementación

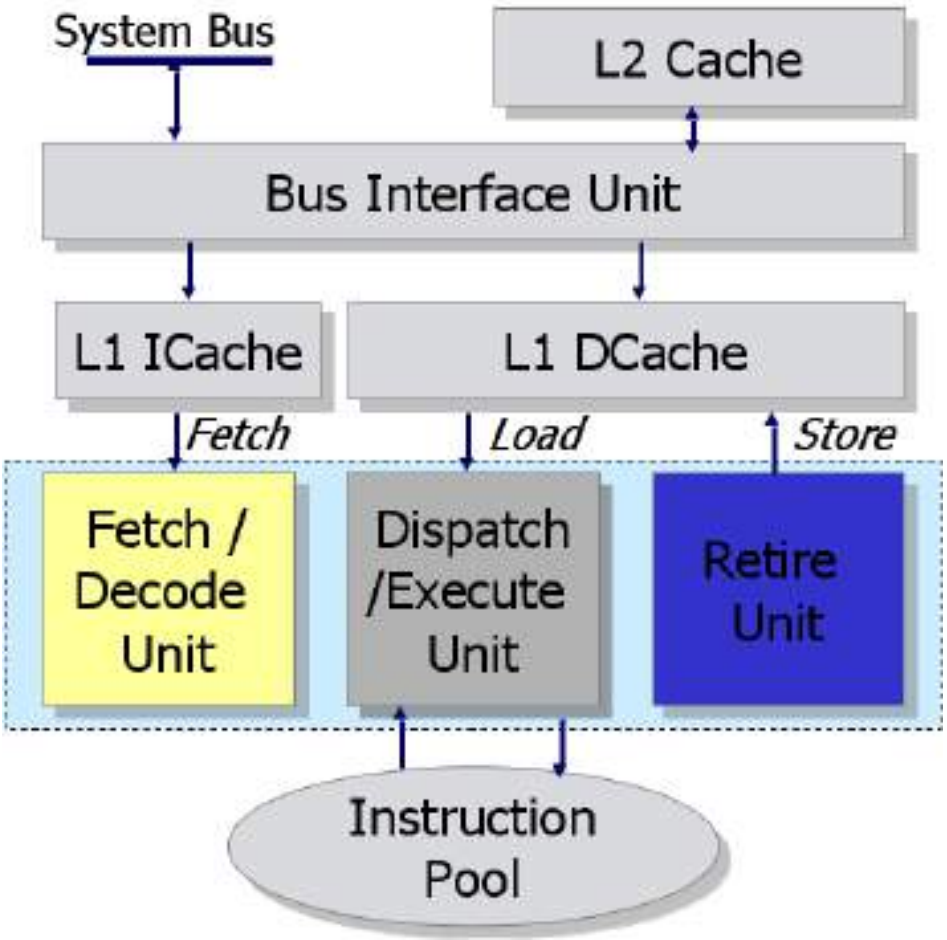
Commit

- Es la fase final de completamiento de la instrucción. A partir de aqui el resultado solo estará en el operando destino.
- Si se trata de un branch con predicción, incorrecta, flushea la entrada del ROB y recomienza a partir de la dirección sucesora correcta del branch.
- Para cualquier instrucción que finaliza correctamente (excepto memory stores) y para los branches con predicción correcta, se copia el valor del ROB en el operando destino.
- Si la operación es un memory store es similar solo que se escribe en una dirección de memoria.

Intel - Microarquitectura P6

- En 1993 Intel presenta el procesador Pentium Pro que incluye una serie de innovaciones.
- Con este procesador se inaugura la Microarquitectura que se llamó P6
- Los sucesivos procesadores que mejoran este modelo dentro de P6 son:
  - Pentium II, Pentium II Xeon,
  - Celeron
  - Pentium III, Pentium III Xeon

Intel - Three Cores Engine



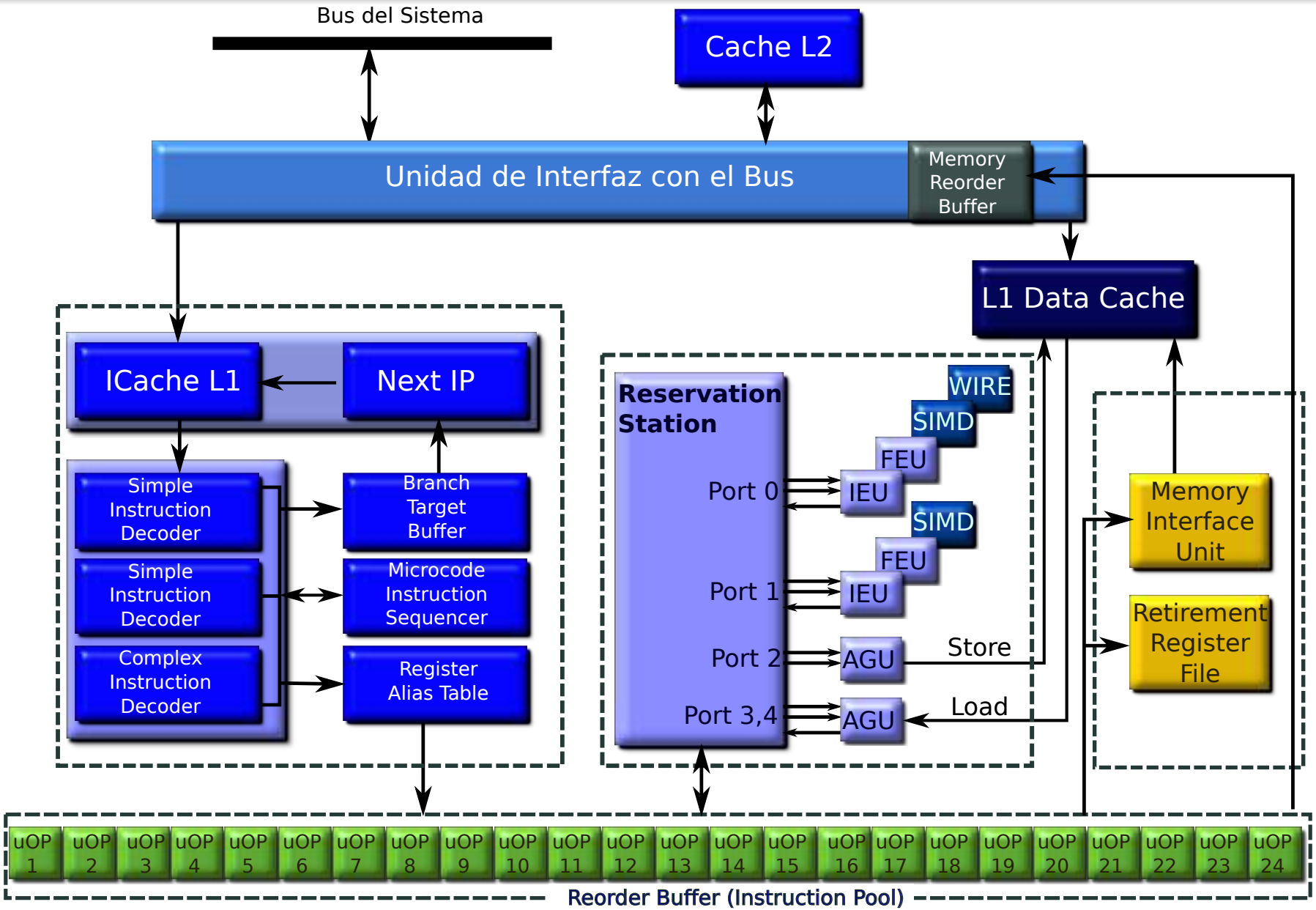
- Emplea Dynamic Instruction Scheduling
- Basado en una ventana de instrucciones y no en un pipeline superescalar.
- Las instrucciones se traducen en micro operaciones básicas (uops)
- Las uops ingresan a un pool (ventana) en donde se mantienen para su ejecución
- Los tres cores tienen plena visibilidad de esa ventana de ejecución
- Implementa ejecución fuera de orden y ejecución especulativa.
- La unidad de despacho y ejecución mantiene el modelo superescalar y lo combina con un súper pipeline de 20 etapas

Ejecución fuera de orden

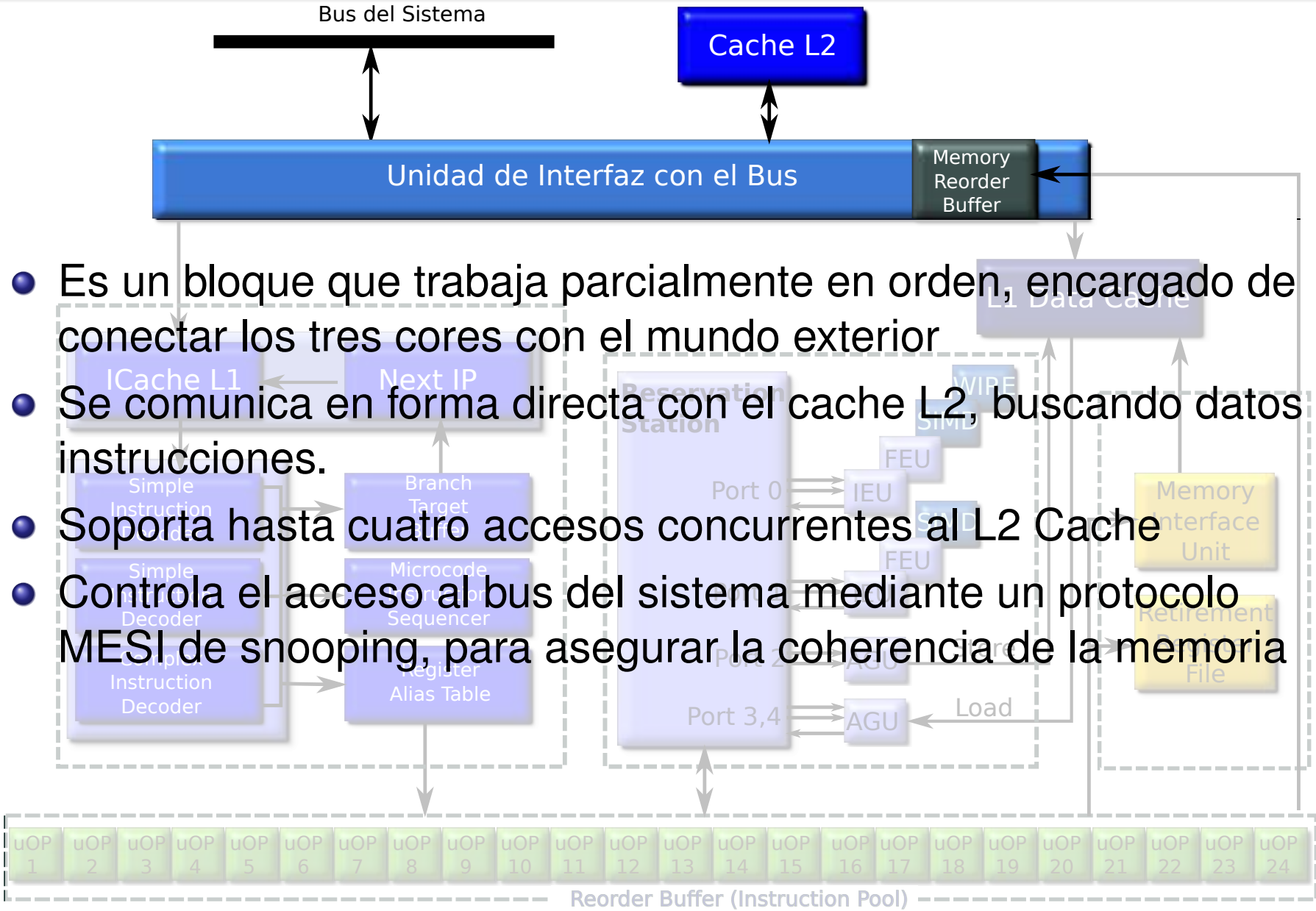
Se trabaja sobre una ventana de instrucciones

Si se observan entre 20 y 30 instrucciones los compiladores generan en promedio 5 saltos dentro de esa ventana. Estos deben ser correctamente predictos.
Como se ejecutan fuera de orden, se mantienen los resultados en el campo adecuado de las

Three Cores Engine en detalle

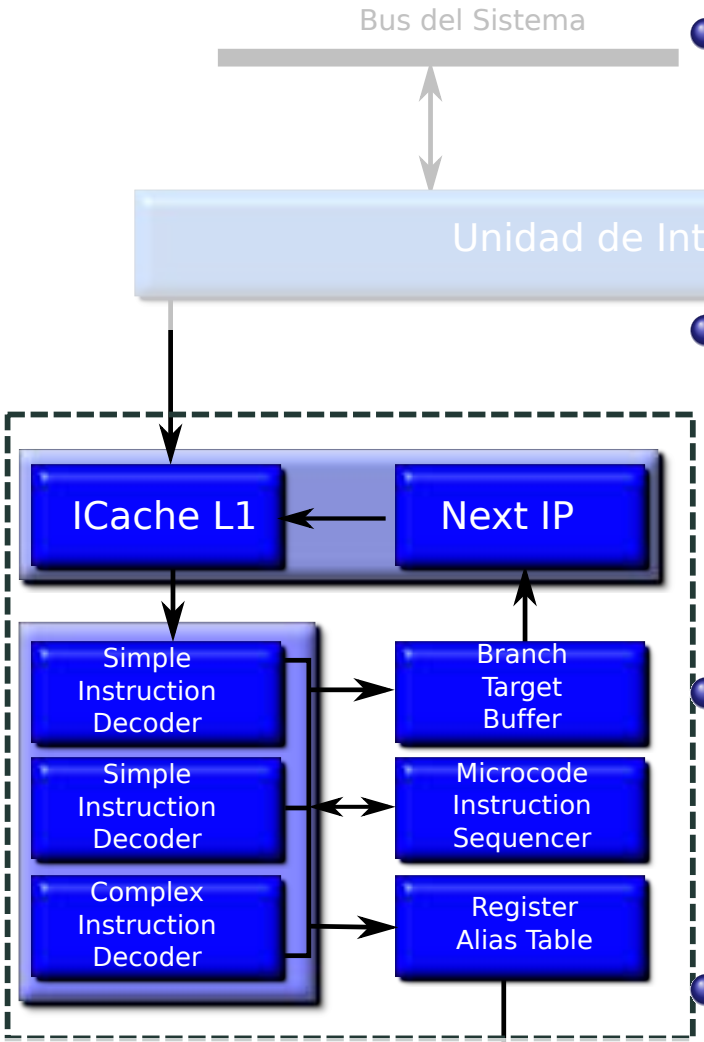


Unidad de Interfaz con el Bus



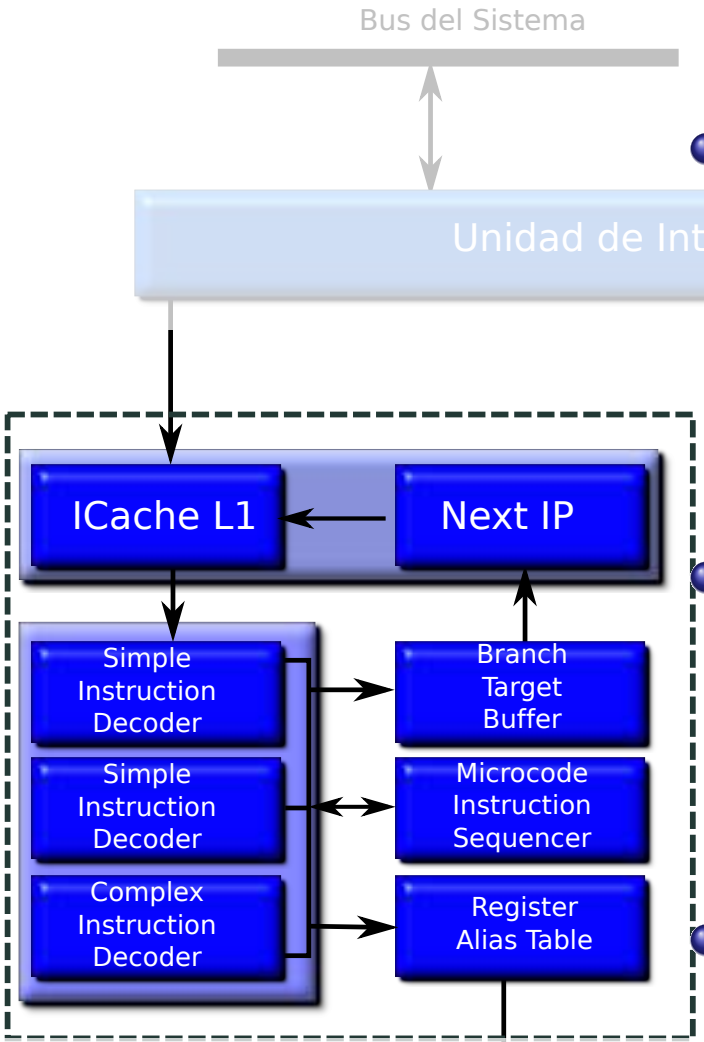
- Es un bloque que trabaja parcialmente en orden, encargado de conectar los tres cores con el mundo exterior
- Se comunica en forma directa con el cache L2, buscando datos e instrucciones.
- Soporta hasta cuatro accesos concurrentes al L2 Cache
- Controla el acceso al bus del sistema mediante un protocolo MESI de snooping, para asegurar la coherencia de la memoria

Three Cores Engine en detalle



- El bloque ***ICache***, es el cache de instrucciones.Su tamaño de línea es 16 bytes. De este modo entrega 16 byte alineados a la Unidad de Decodificación.
- El puntero ***Next_IP***, apunta a la línea de la siguiente instrucción o a la de la instrucción target de un salto de acuerdo con la predicción generada en el Branch Target Buffer.
- Hay tres decodificadores que toman en paralelo esta línea de 16 bytes proveniente del ***ICache*** y buscan dentro las instrucciones que hay que decodificar.
- Cada instrucción de la Arquitectura Intel (***IA*** de aquí en mas) se traduce en una o mas uops las cuales tienen dos operandos fuente y un operando destino.

Three Cores Engine en detalle



- La mayor parte de las instrucciones ***IA*** se traducen en una uOP. El resto se convierten 4 uOPs, y las mas complejas entran al Bloque ***MicroCode Instruction Sequencer*** (de aquí en mas MIS) que genera la cantidad de uOPs necesarias.
- Las uOPS se encolan y se envían al ***Register Alias Table***, que se encarga de traducir los registros de la ***IA*** en registros de un amplio register file interno del procesador. Esta operación se llama Register Renaming.
- Además las uOPs se ingresan al Allocator que les agrega información de estado y las ingresa al pool de instrucciones.

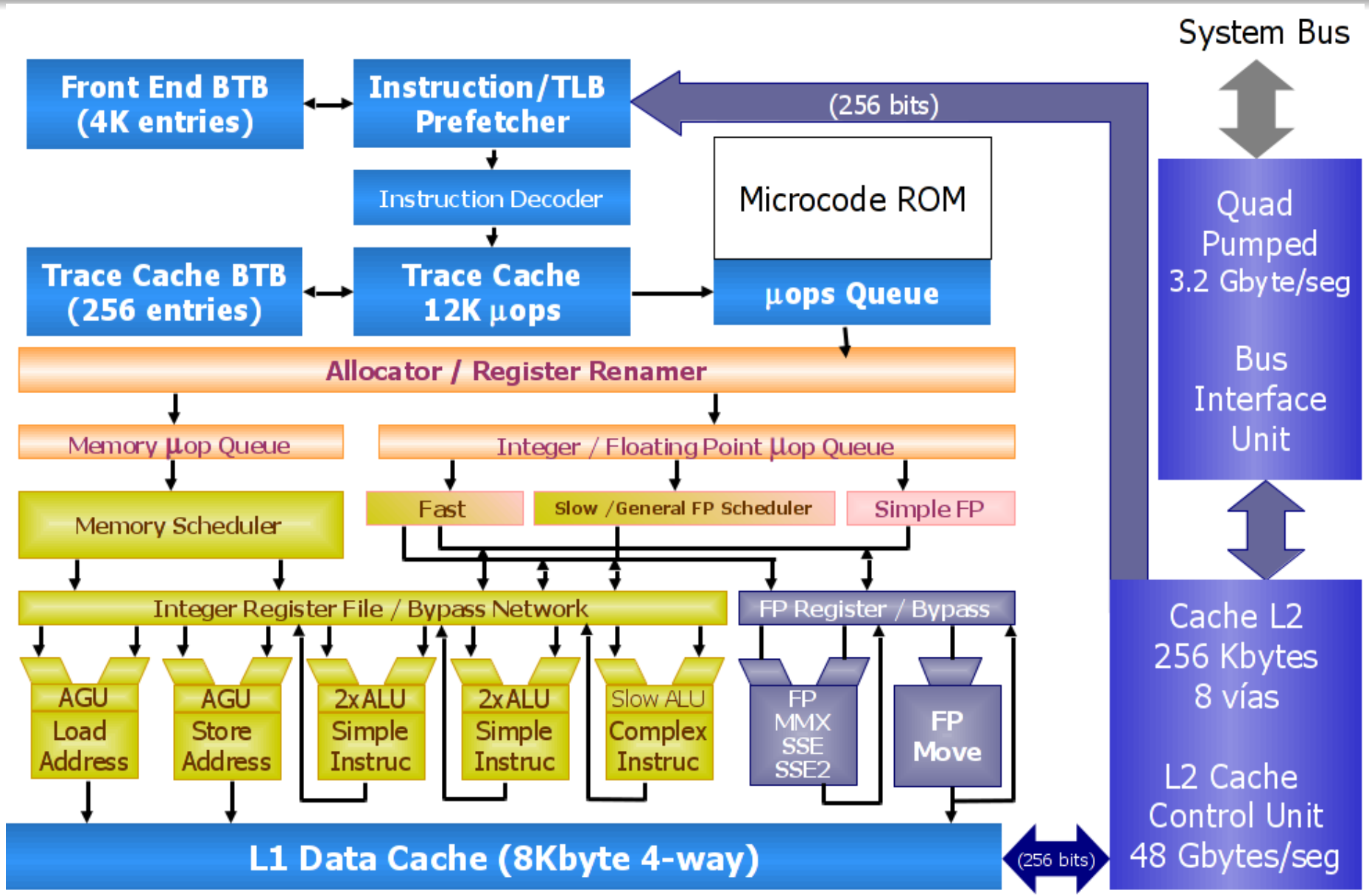
Three Cores Engine en detalle

- La Unidad de Despacho selecciona uOPs desde el pool de instrucciones dependiendo de su estado y no de su orden dentro del mismo.
- Si el estado de la uOP indica que sus operandos están disponibles, la ***Reservation Station*** verifica que esté libre algún recurso para su ejecución, y eventualmente le envía la uOP.
- El resultado será escrito en el pool de uOPs por la Unidad de Ejecución correspondiente una vez finalizada.
- Se dispone de 5 puertos de ejecución. De este modo se pueden schedular a lo sumo 5 uOPs por ciclo de clock. EN la práctica se tiene un promedio de 3.
- Respecto de los saltos, las uOPs se marcan como tales en el ***Re Order Buffer*** (de ahora en mas ***ROB***) y se les pone la dirección target predicta por el ***BTB***. Si el salto falla, se limpian del ***ROB*** las instruciones subsiguientes así estén finalizadas

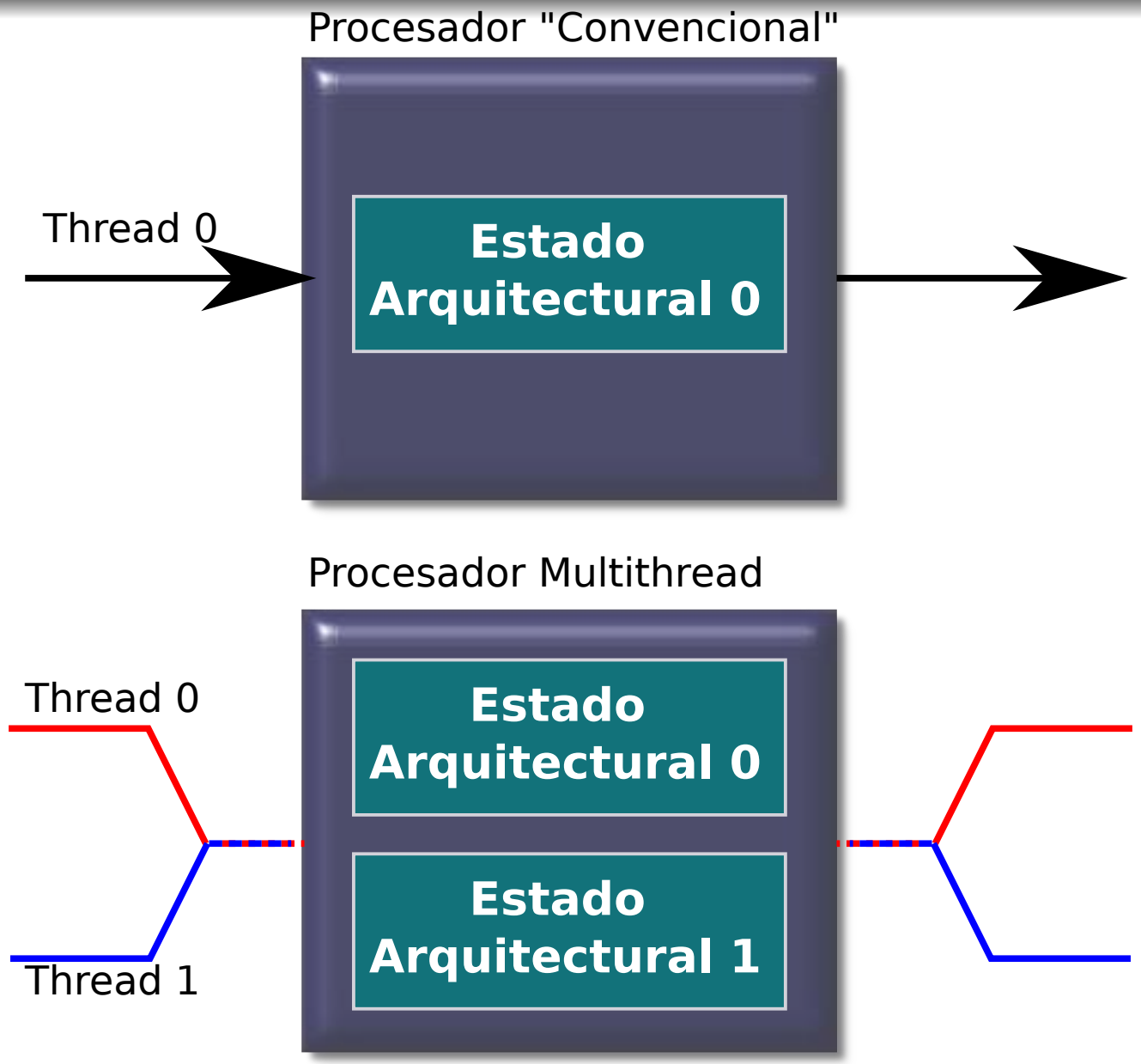
Three Cores Engine en detalle

- La Unidad de Retiro, se encarga de monitorear el estado de cada instrucción del ***ROB***.
- No solo debe chequear su estado, sino que las debe reinsertar en el orden en que ocupan en el programa.
- Esto incluye el procesamiento de Interrupciones, excepciones, trap, breackpoints, y predicciones fallidas.
- El ***Retirement Register File*** (de aquí en mas ***RRF***), se encarga de escribir los operandos resultantes en los registros de la ***IA*** y la Unidad de Interfaz de memoria, aplica en el cache L1 de Datos el resultado si este corresponde a una dirección de memoria.
- Puede retirar 3 uOPs por ciclo de clock.

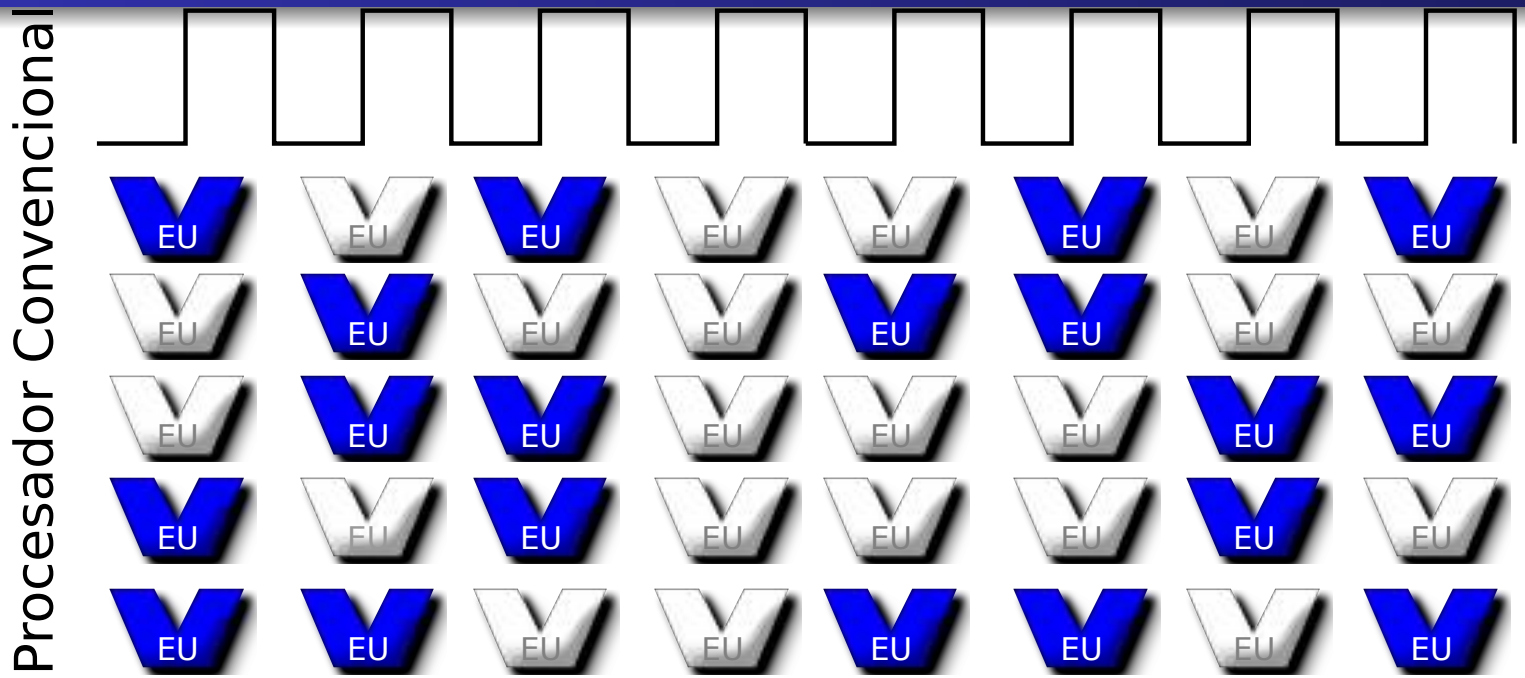
Intel Netbusrt



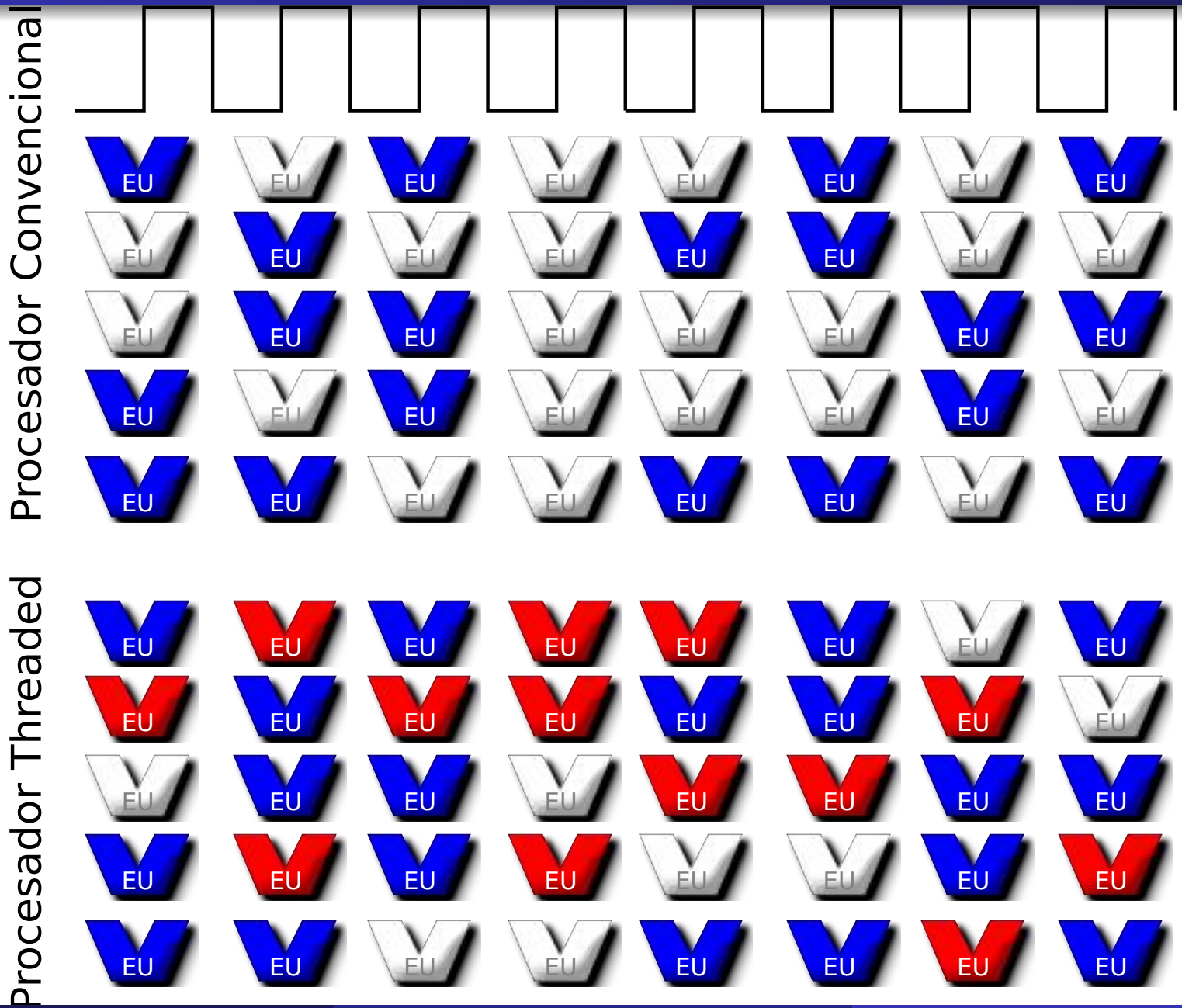
Procesadores Multithred



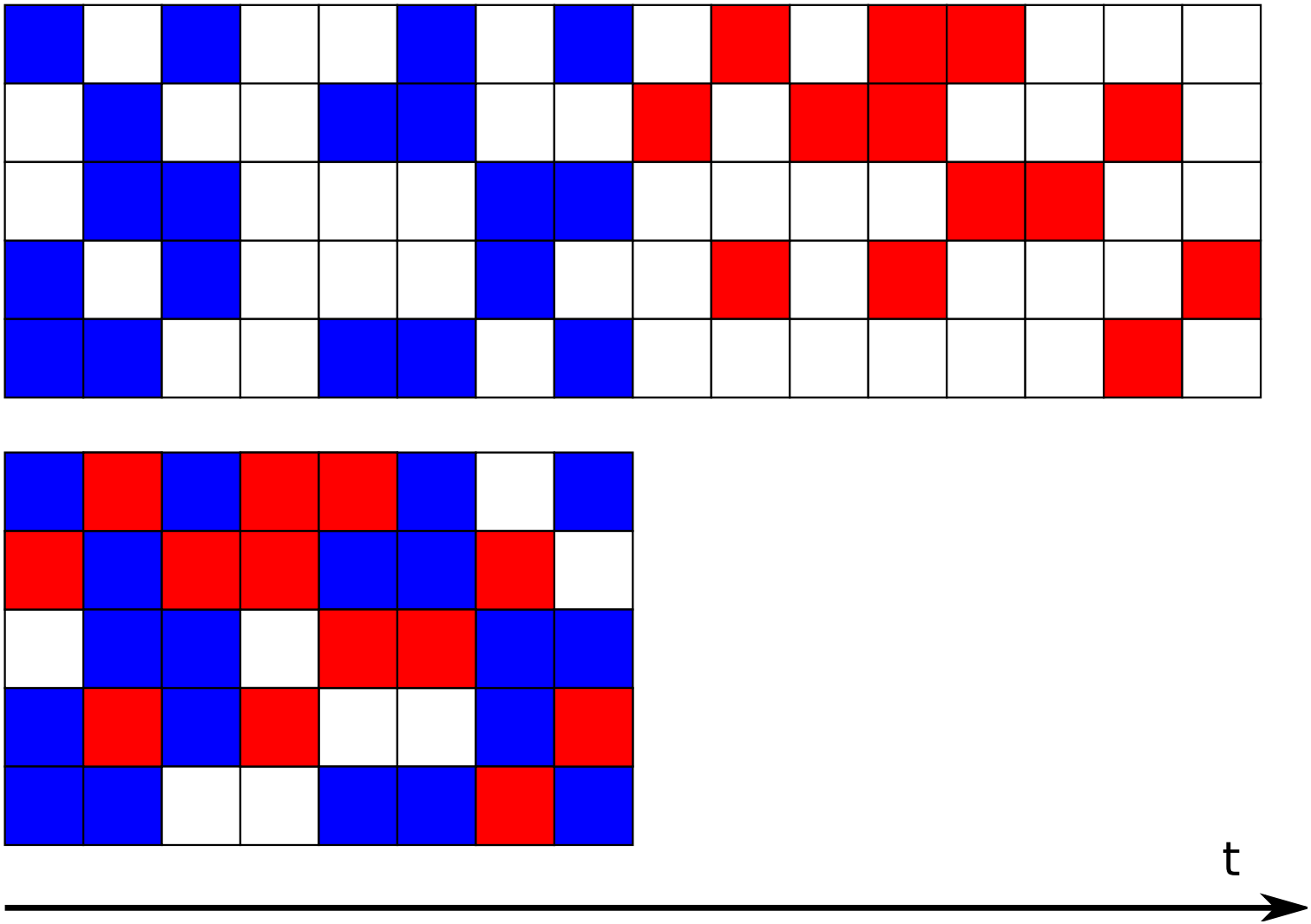
Ejecución en los Procesadores Multithred



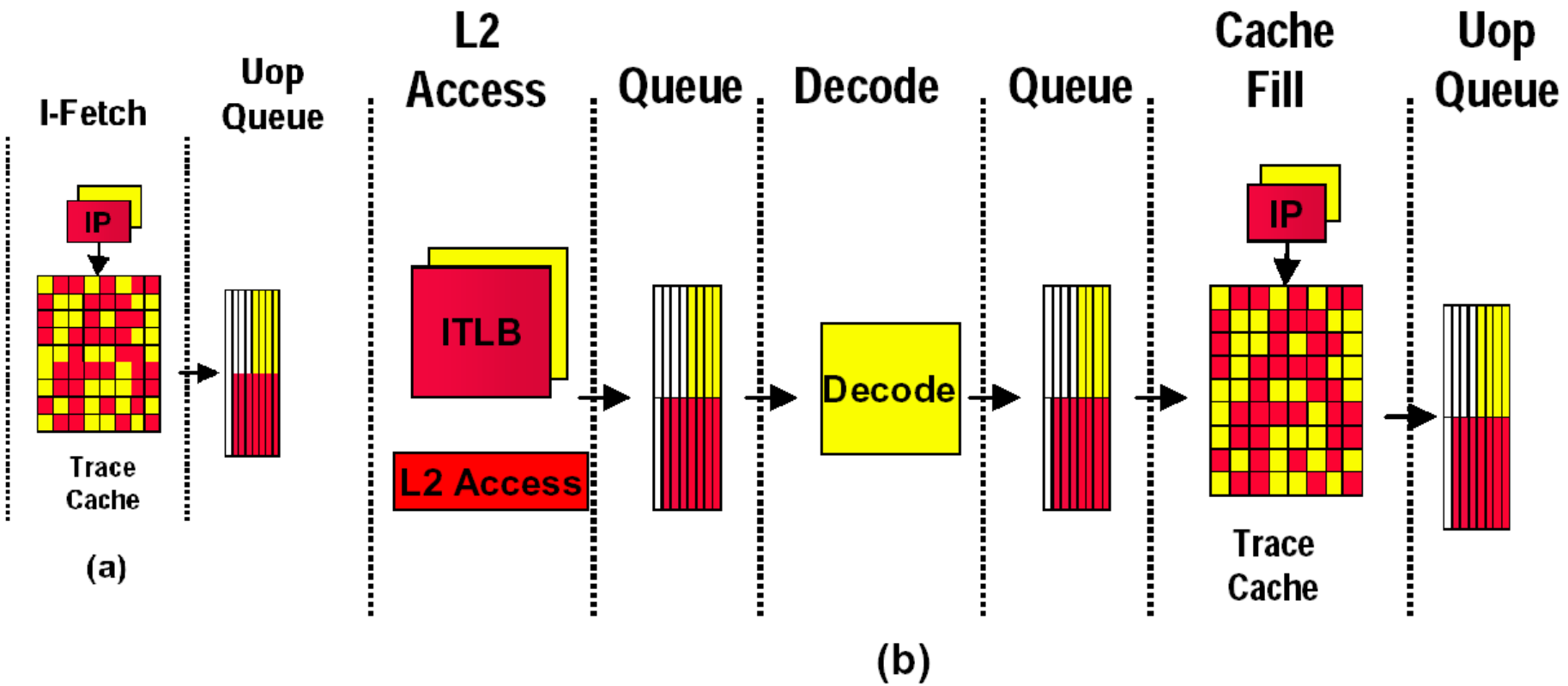
Ejecución en los Procesadores Multithred



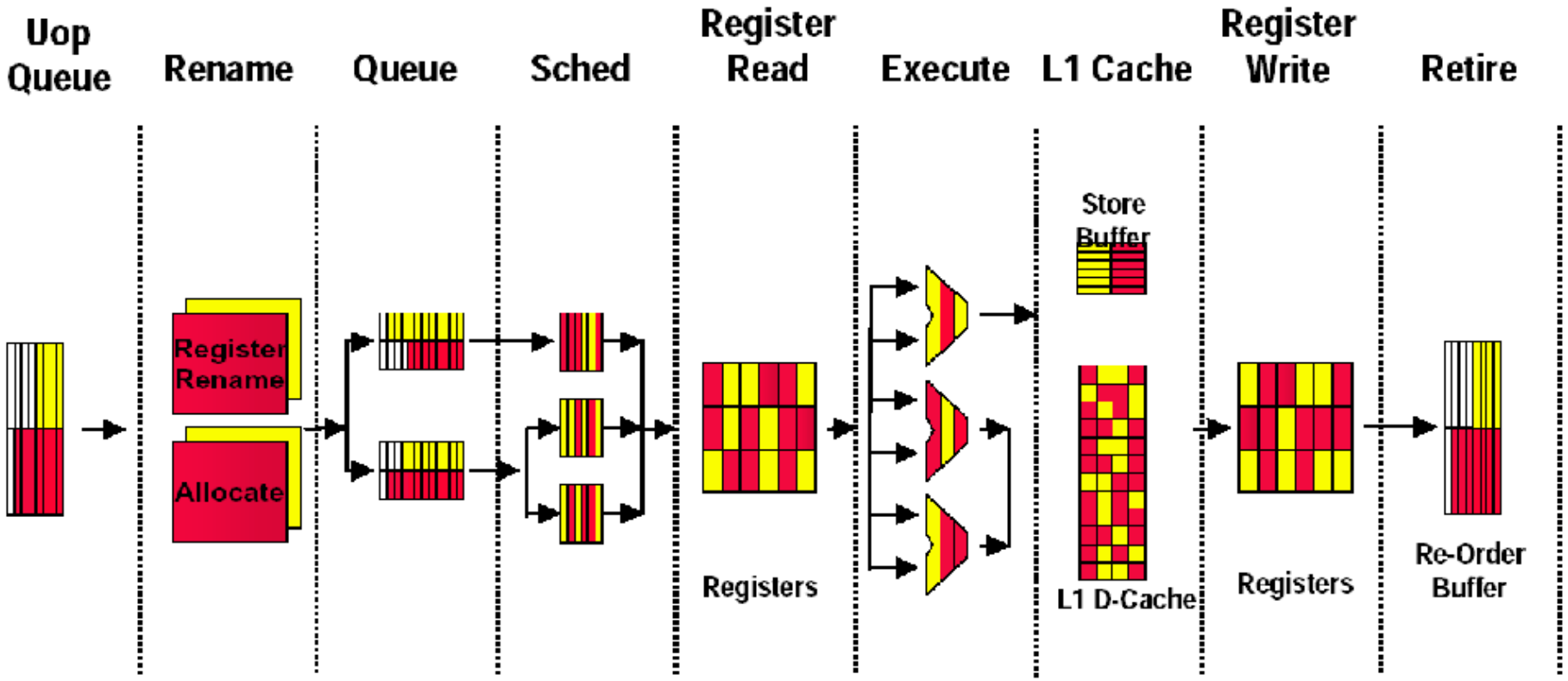
Ejecución en los Procesadores Multithred



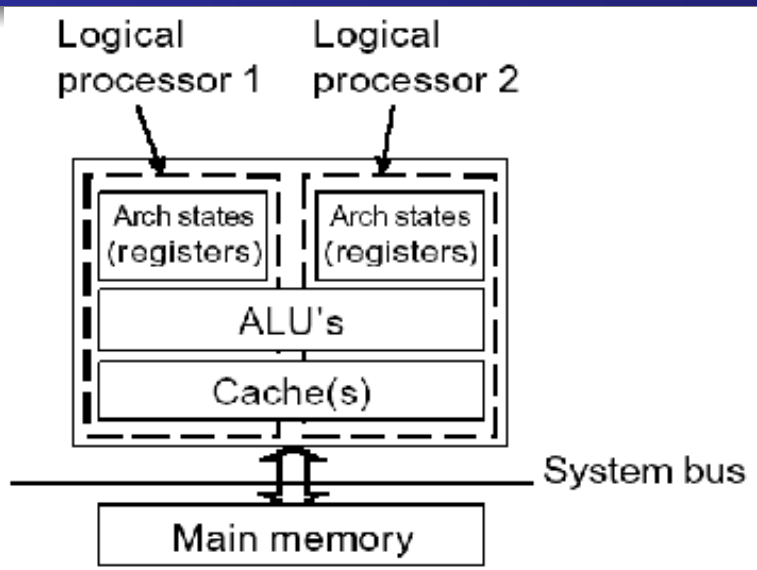
Intel Hyperthreading



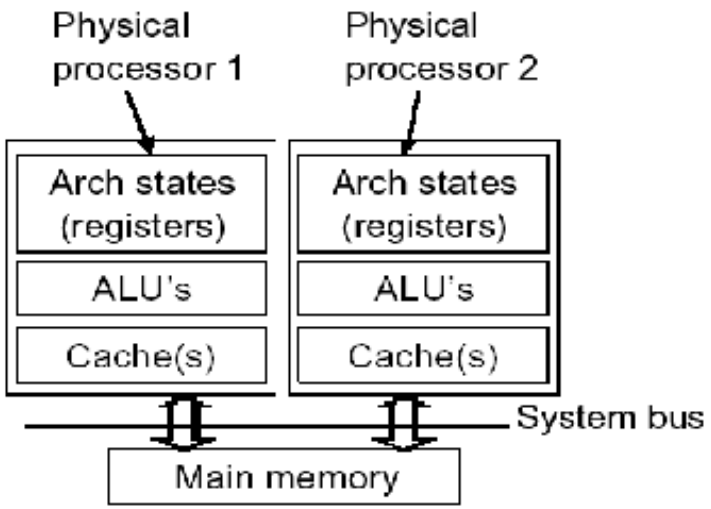
Intel Hyperthreading



Hyperthreading versus Multicore



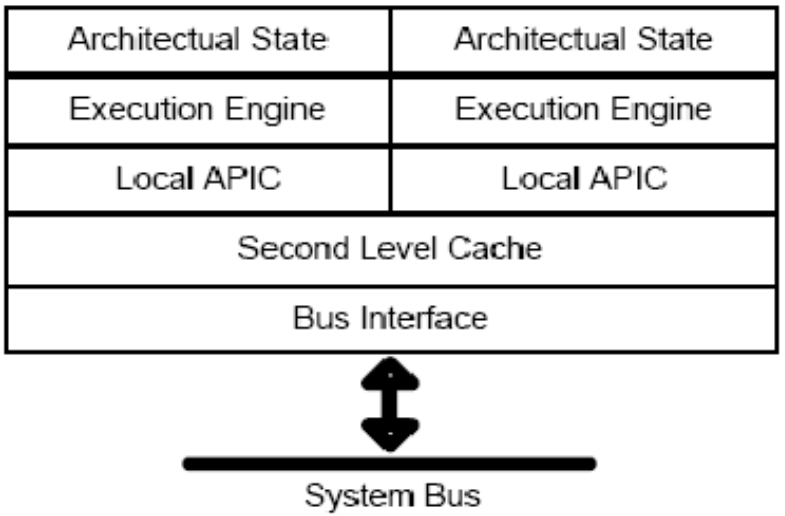
(a)



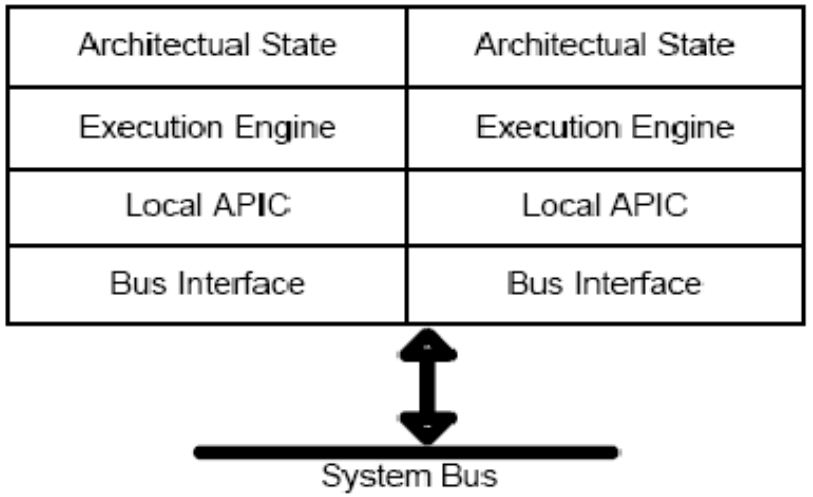
(b)

Multicore

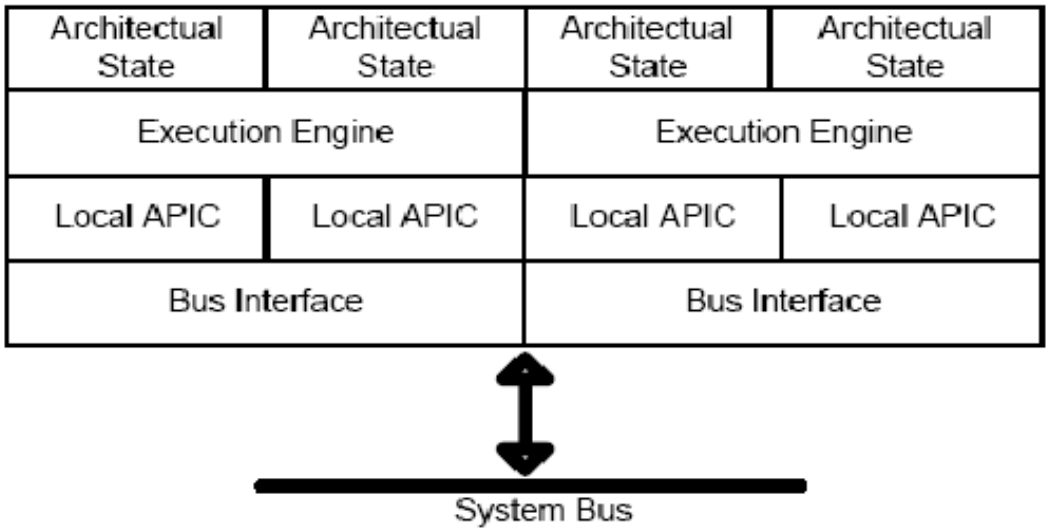
Intel Core Duo Processor
Intel Core 2 Duo Processor
Intel Pentium dual-core Processor



Pentium D Processor



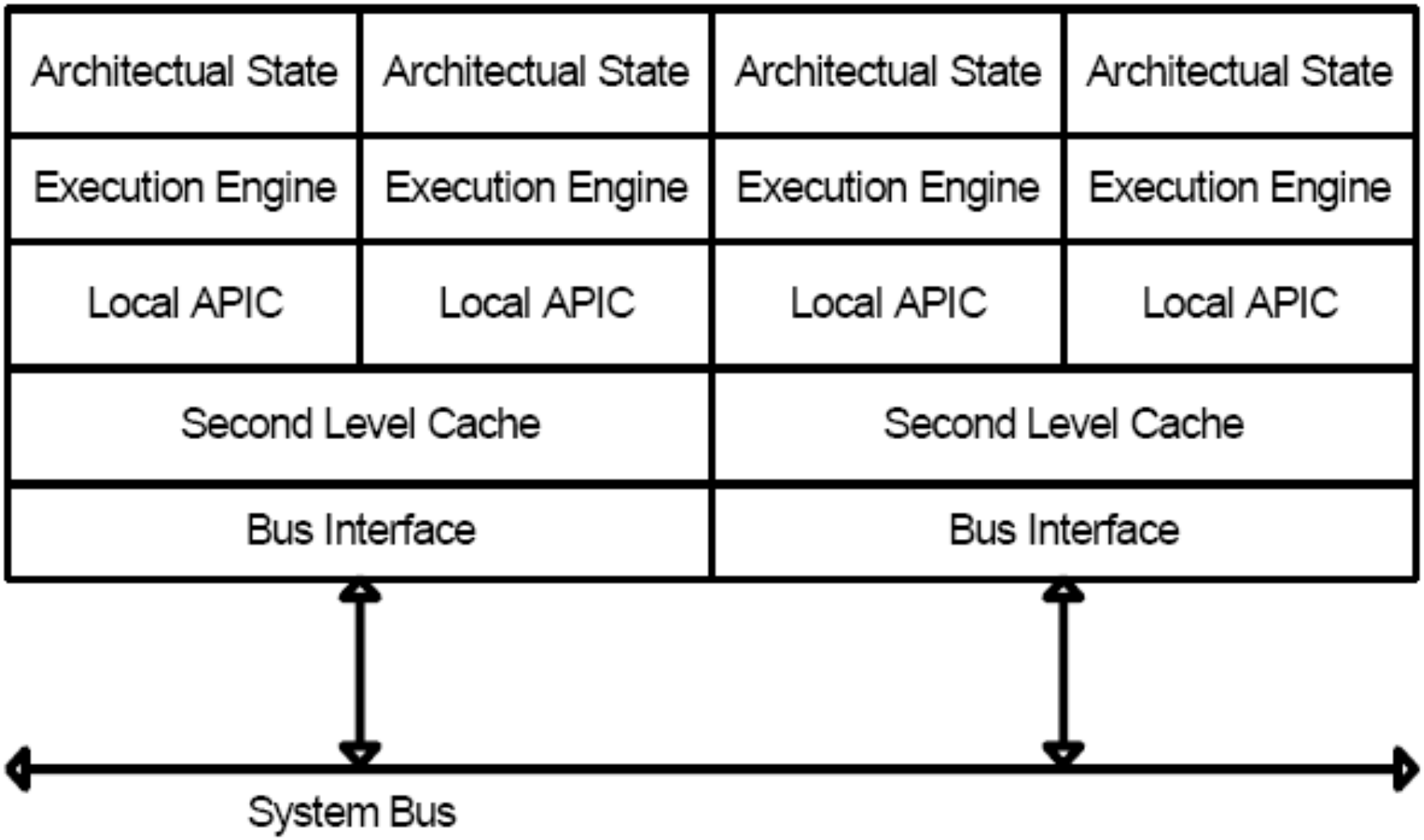
Pentium Processor Extreme Edition



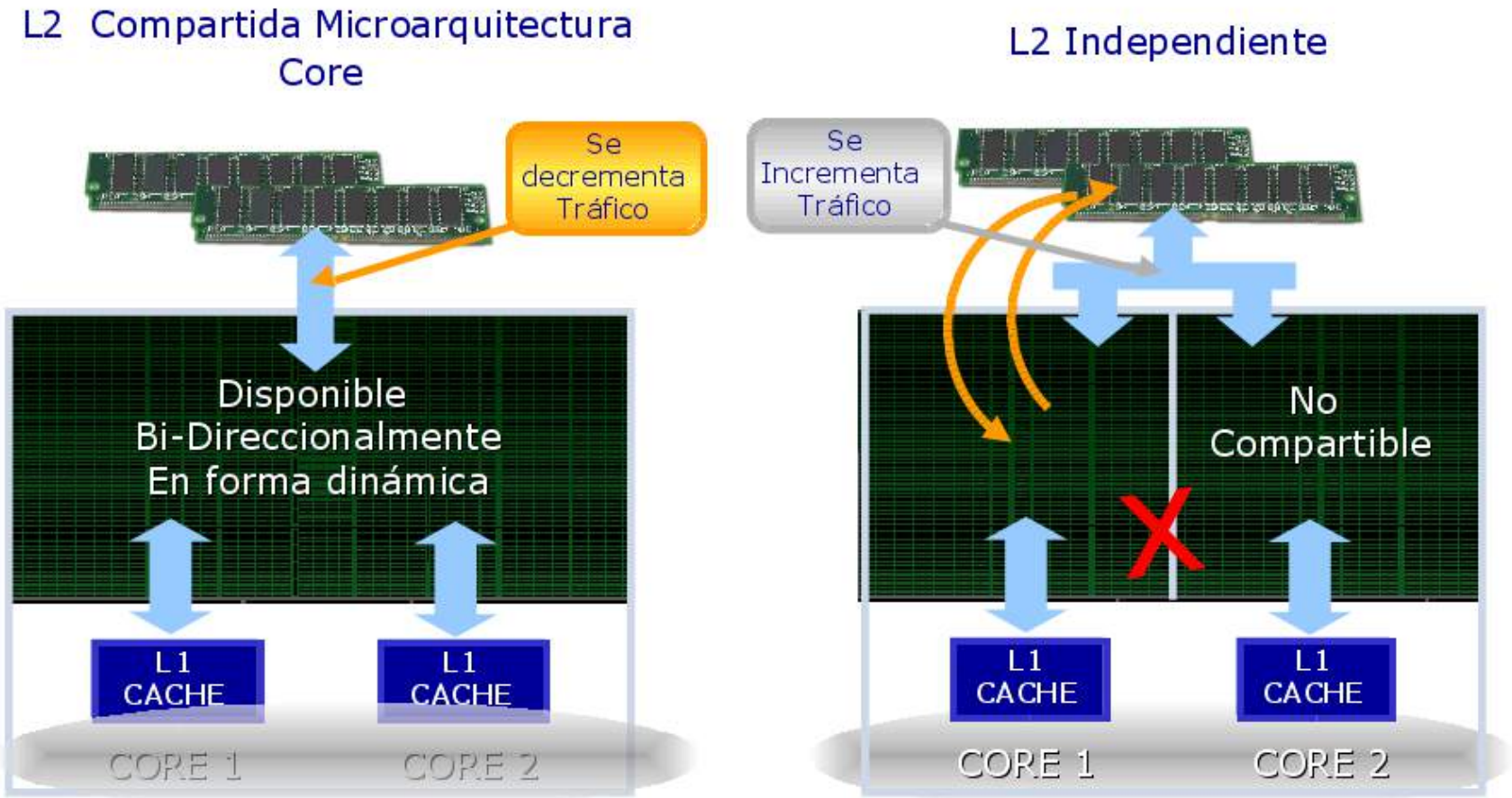
OM19809

Multicore

Intel Core 2 Extreme Quad-core Processor
Intel Core 2 Quad Processor
Intel Xeon Processor 3200 Series
Intel Xeon Processor 5300 Series



Smart Cache


L2 Compartida Microarquitectura Core
Se decrementa Tráfico
Disponible Bi-Direccionalmente En forma dinámica
L1 CACHE
L1 CACHE
CORE 1
CORE 2
L2 Independiente
Se Incrementa Tráfico
No Compartible
L1 CACHE
L1 CACHE
CORE 1
CORE 2

Detalle de Componentes Internos de los ATOM E65XX

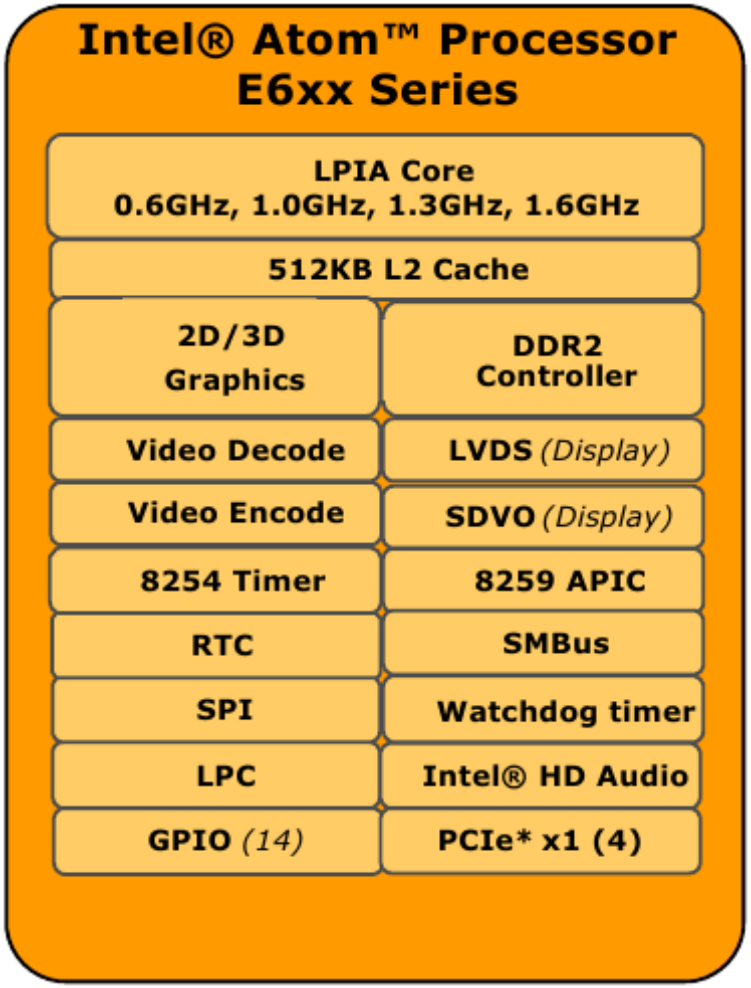


Figura: ATOM. Detalle de componentes Internos. ©Cortesía Intel

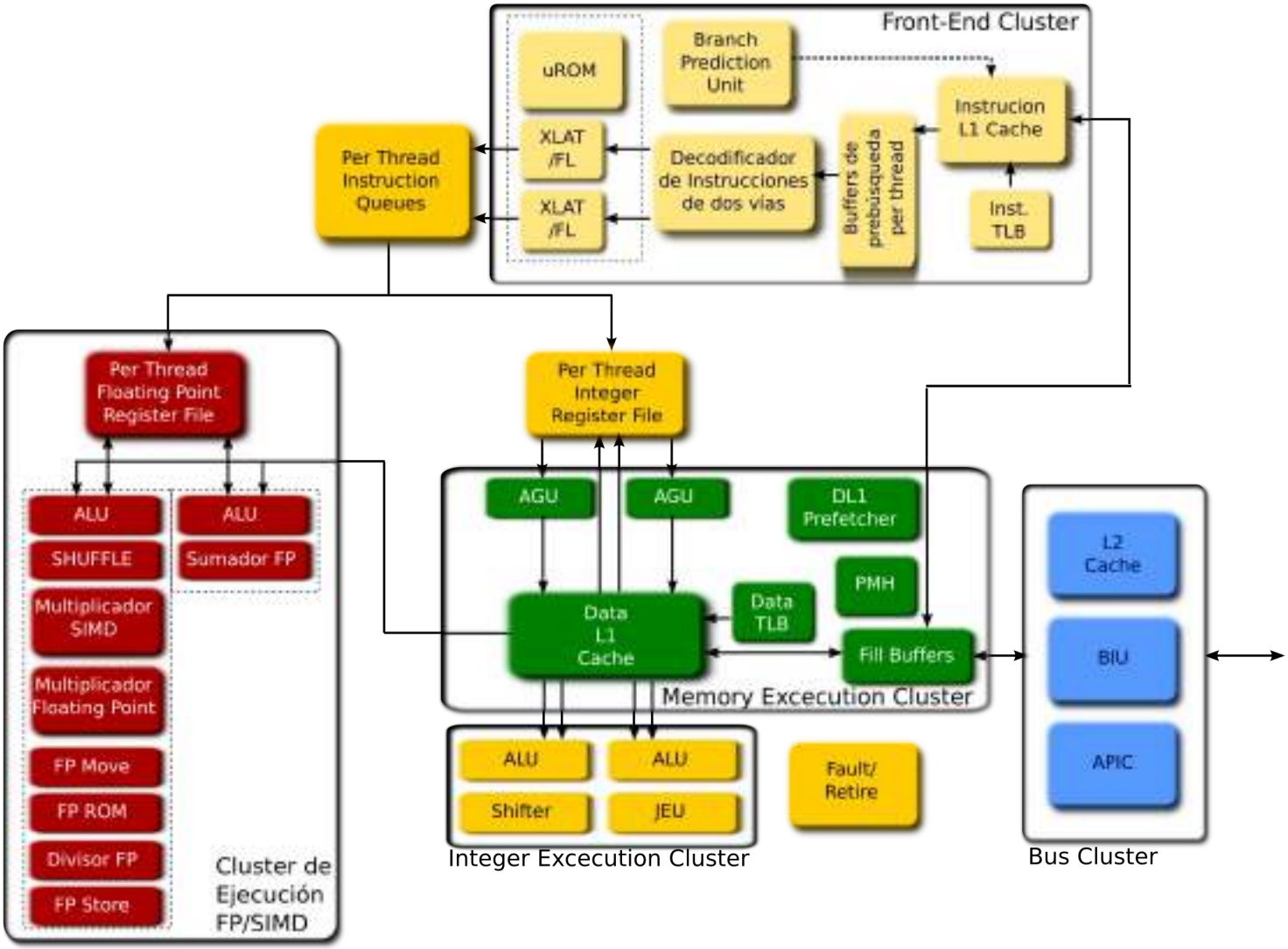
Características principales

- Clock: 0.6GHz a 1.6GHz.
- Proceso de Fabricación 45 y 32 nm high-k metal-gate CMOS.
- 1 Core con SMT (Simulaneous Multi Threadiming) de 2 threads (mediante la clásica Tecnología Hyperthreading).
- Tercer generación de *Enhanced Intel SpeedStep ® technology*. Esta tecnología permite adaptar la frecuencia de clock a las demandas de procesamiento, reduciéndola cuando el procesador no está exigido de modo de minimizar la disipación de calor y bajando por consiguiente los requerimientos de potencia.
- Tecnología de Virtualización por hardware. Idem procesadores IA-32.
- Extensiones de 64 bits. Idem procesadores IA-32, excepto que direcciona solo de 2 a 4 Gbytes de Memoria física.
- Extensiones SIMD de 128 bits (Hasta Tecnología SSSE3). Idem procesadores IA-32.

Advanced Micro-Ops Execution

- Ejecución de Instrucciones que equivalen a una micro operación única desde la decodificación hasta el retiro, incluyendo instrucciones con registro único, carga y almacenamiento.
- Pipeline con ejecución en orden de 16 etapas optimizado para maximizar el rendimiento y reducir el consumo de energía.
- Pipelines dobles para habilitar, decodificar, ejecutar y retirar dos instrucciones por ciclo.
- Stack pointer mejorado para ejecución de ingreso y retorno a funciones.

Diagrama detallado



Subsistema de memoria

Cache

- L1 de Instrucciones de 20 K.
- L1 de Datos de 32 K.
- L2 de Instrucciones y Datos de 512 K, asociativo de 8 vías.
- Bus de datos de 256 bits entre L1 y L2 para maximizar la velocidad de transferencia de datos.
- Prebúsqueda y carga especulativa de datos en los cache L1 y L2, en función de detección inteligente de patrones de acceso a fin de maximizar el hit rate.

Controlador DDR

- Single Channel Memory Controller DDR2. 800 MT/s (Mega Transfer per seconds)
- 8 devices, hasta 2GB
- Bus de 32 bits
- Memory down only

Subsistema Video

Video Engine
- Hardware de Codificación y Decodificación de video
- Formatos de Codificación MPEG4 y H.264.
- Formatos de Decodificación MPEG2, MPEG4, VC1, WMV9, y H.264.

Gráficos Integrados
- Frecuencia de hasta 400 MHz.
- Soporte OpenGL *ES2.0, OpenVG * 1.1

Dual Display
- LVDS y SDVO de 24-bit single channel

Controlador de Gráficos en detalle

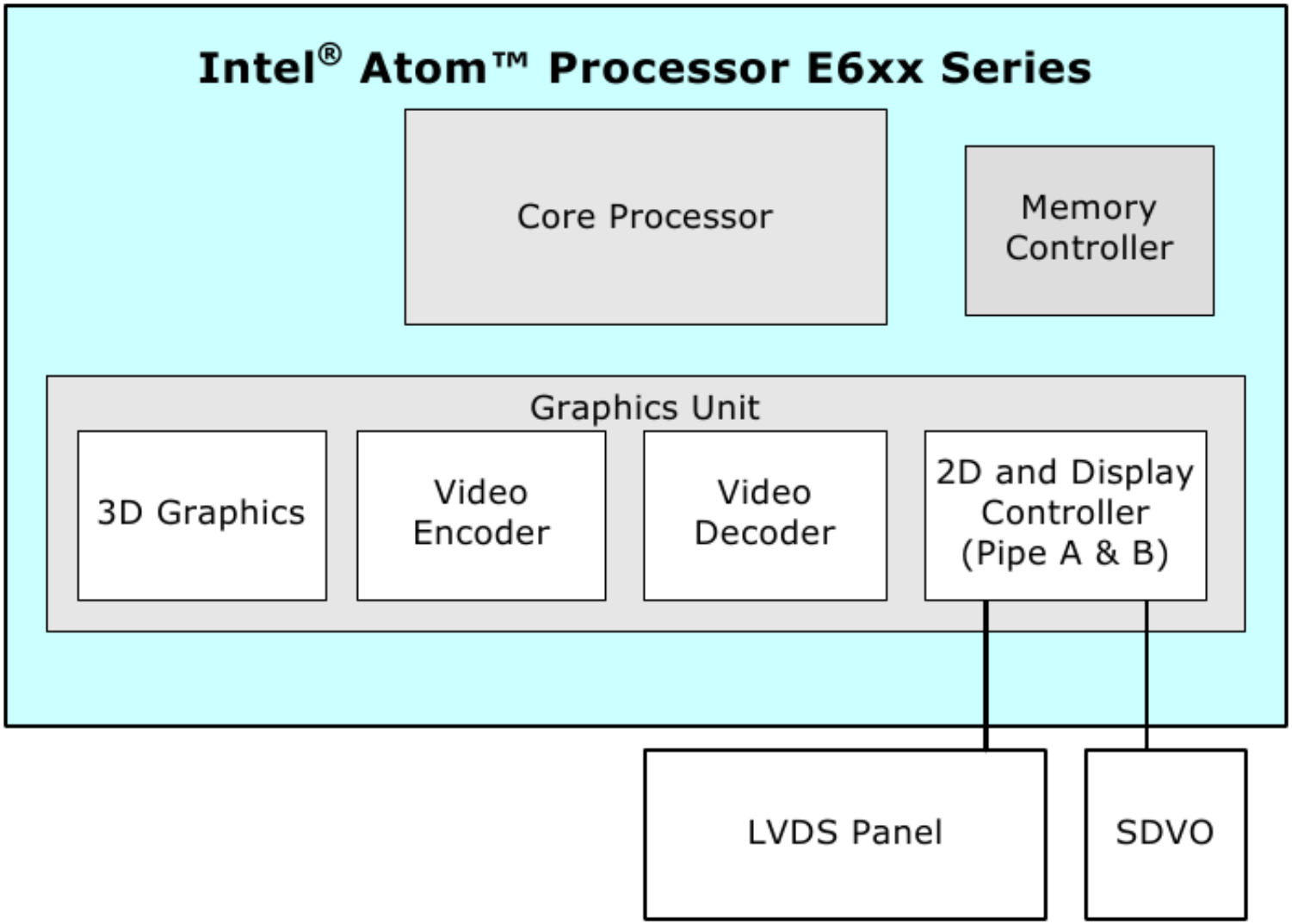
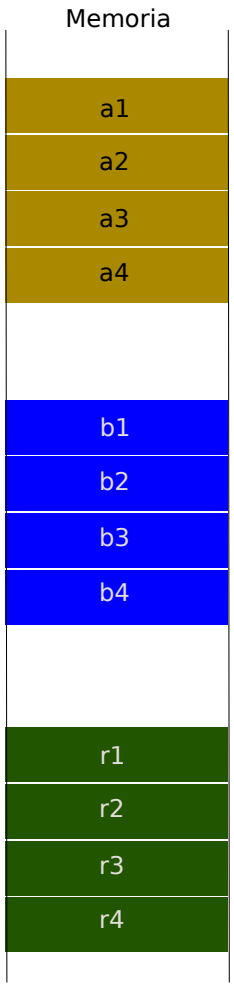


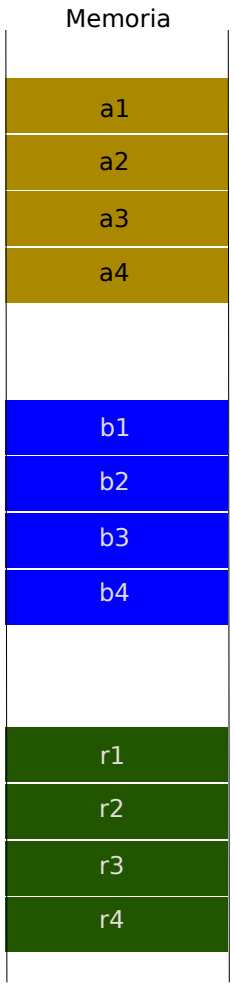
Figura: Subsistema de Video Interno. ©Cortesía Intel

Single Instruction Multiple Data



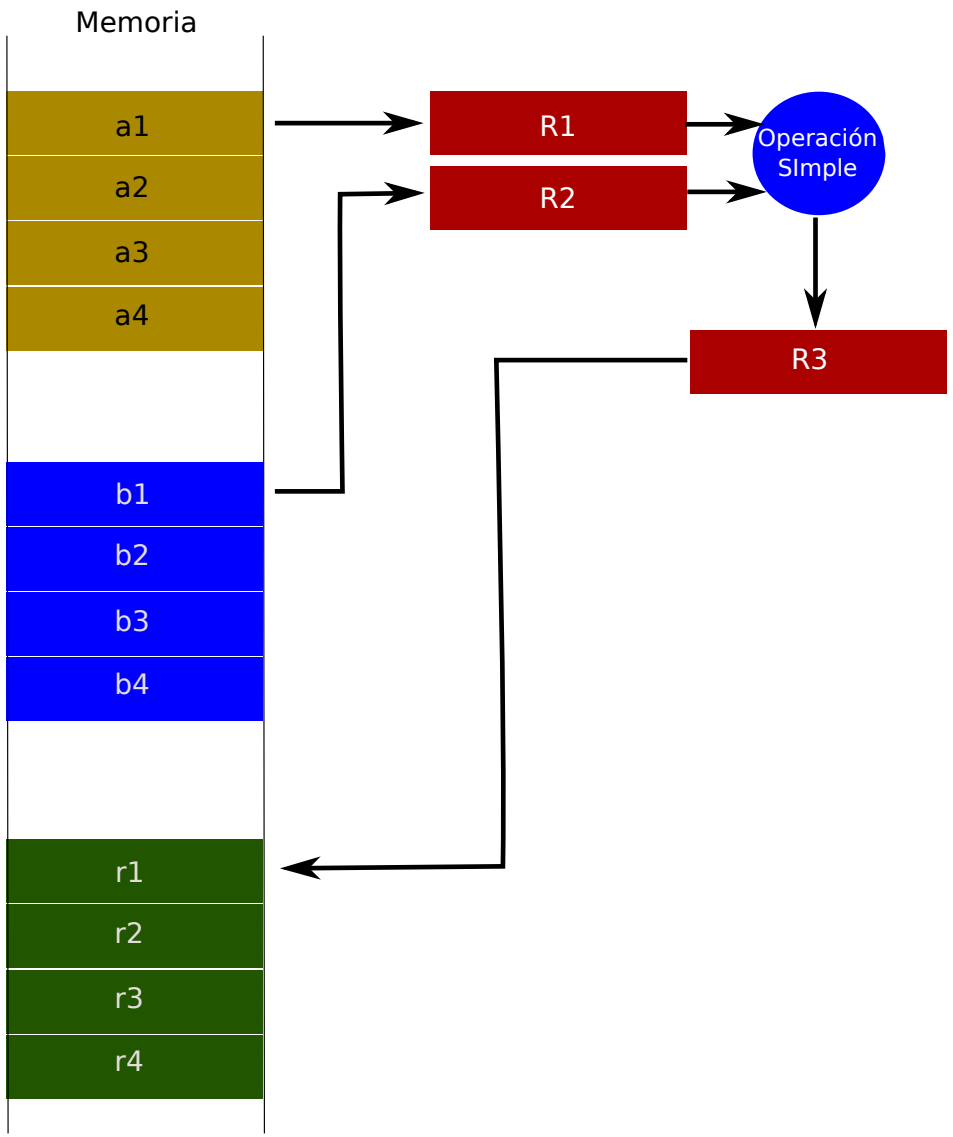
- Se trata de un modelo de ejecución capaz de computar una sola operación sobre un conjunto de múltiples datos.
- Se refiere a esta técnica como paralelismo a nivel de datos.
- Es particularmente útil para procesar audio, video, o imágenes en donde se aplican algoritmos repetitivos sobre sets de datos del mismo formato y que se procesan en conjunto, como por ejemplo en filtros, compresores, codificadores en donde la salida depende de los últimos *n* valores de muestras tomados.
- La figura muestra el layout típico de variables memoria para aplicar este modelo

Como sería la vida sin SIMD?

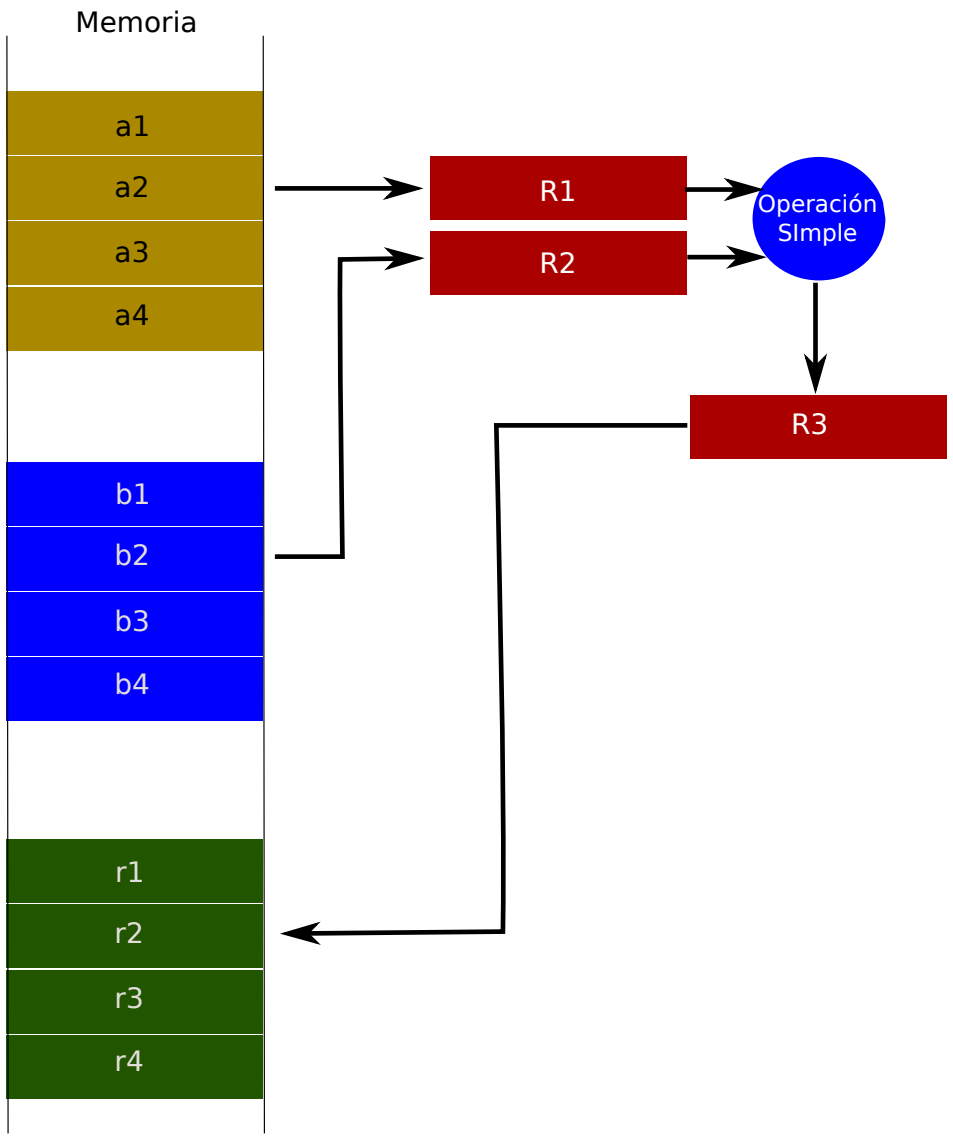


- Considerando el sector de memoria de la figura, nos proponemos realizar una operación aritmética o lógica sobre las cadenas de datos a_n y b_n, almacenando el resultado en r_n
- Si un procesador no dispone de un modelo arquitectural que le permita implementar paralelismo a nivel de datos, como SIMD,esta operación implica un loop.

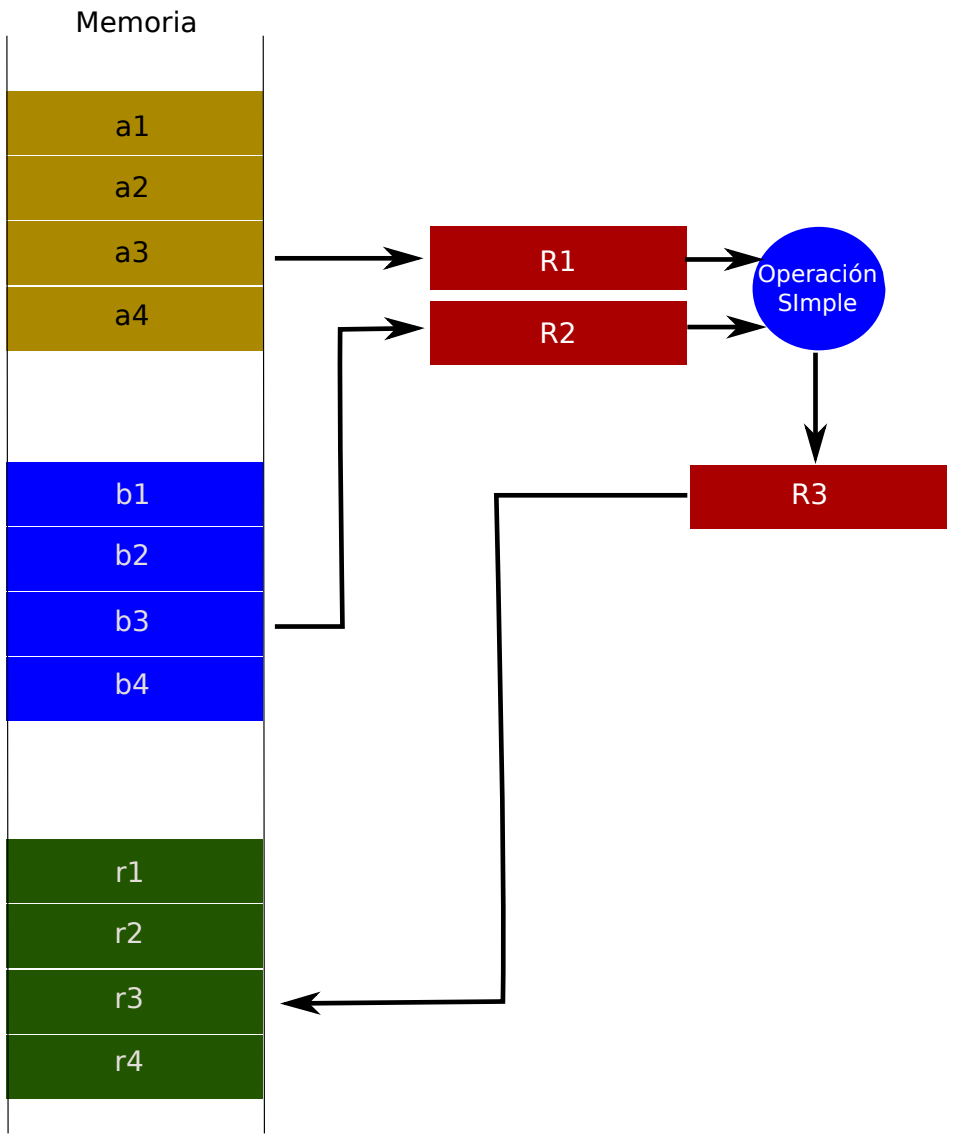
Single Instruction Single Data



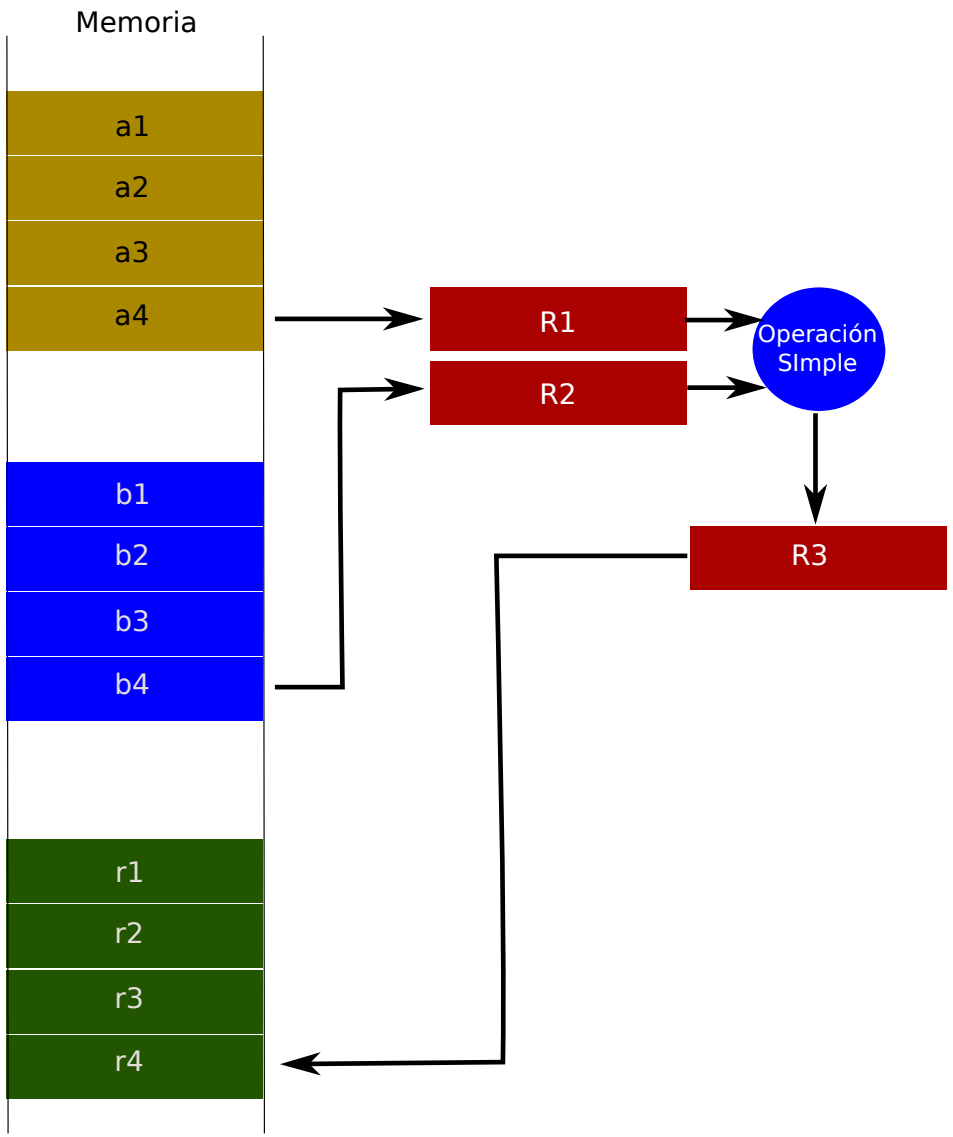
Single Instruction Single Data



Single Instruction Single Data

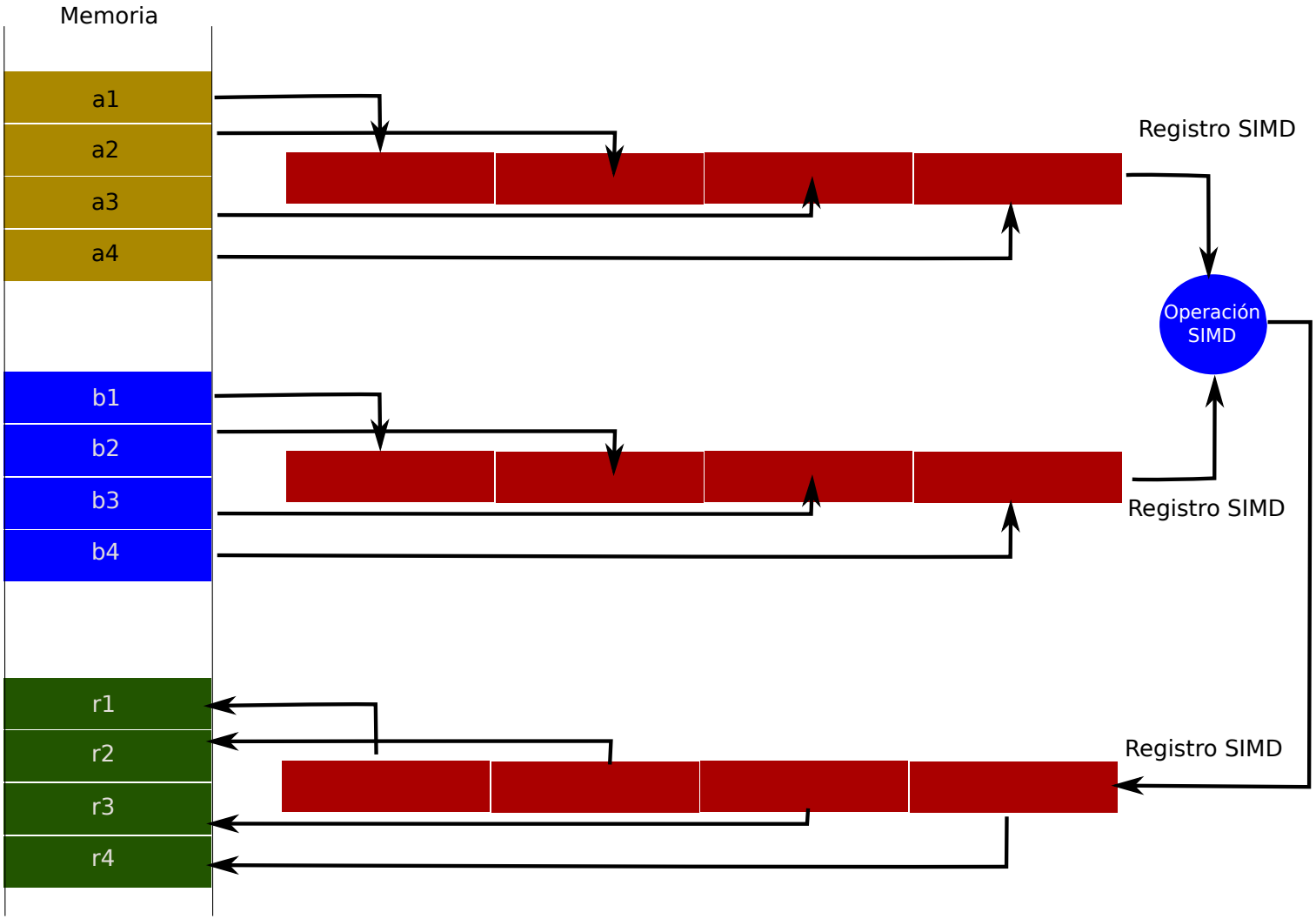


Single Instruction Single Data

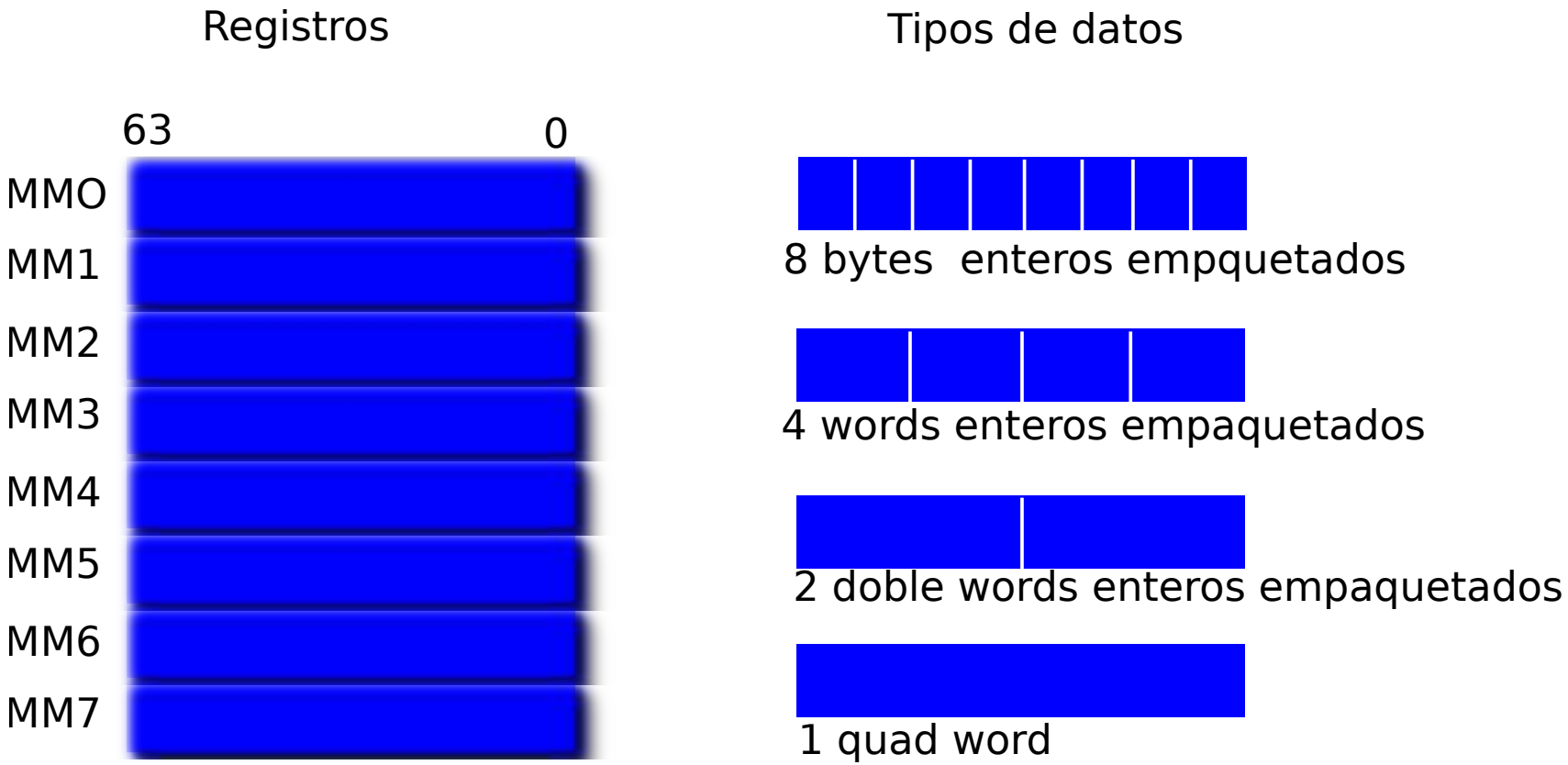


Single Instruction Multiple Data

- Cada Registro se carga en una sola instrucción
- La operación se efectúa en la segunda instrucción

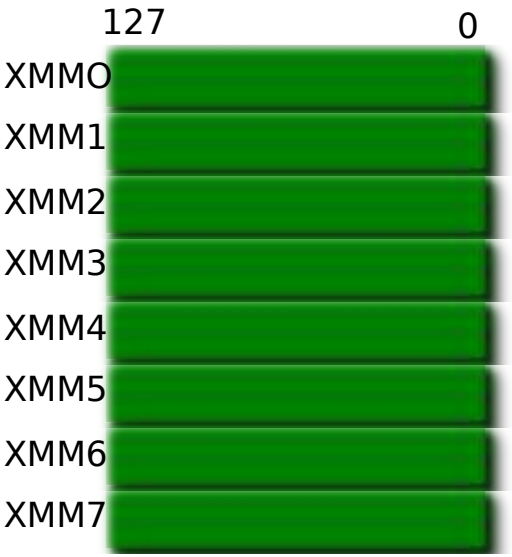


Multimedia extensions - MMX



Streaming SIMD Extension

Registros



Tipos de datos


16bytes enteros empaquetados


8 words enteros empaquetados


4doble words enteros empaquetados

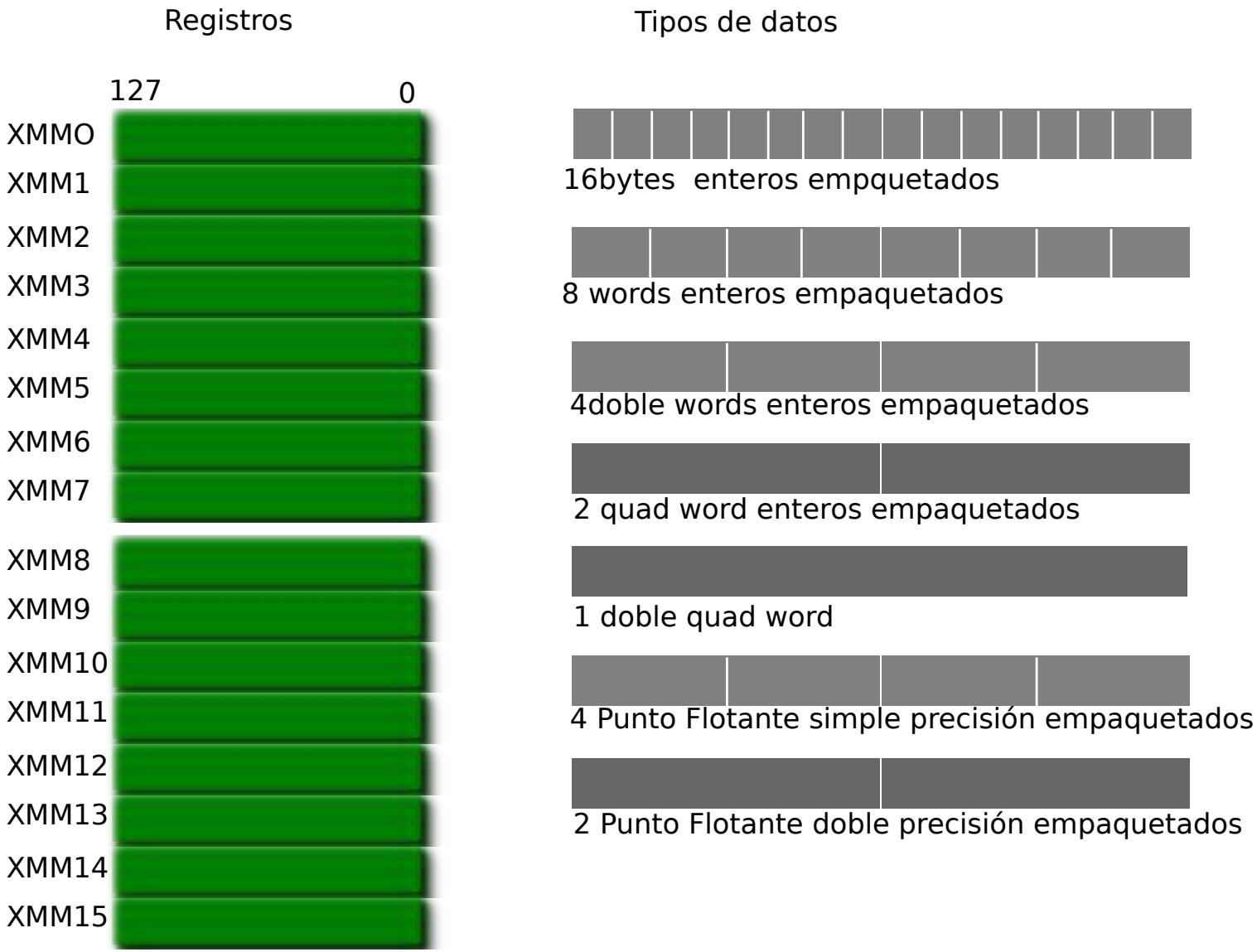

2 quad word enteros empaquetados


1 doble quad word


4 Punto Flotante simple precisión empaquetados


2 Punto Flotante doble precisión empaquetados

Streaming SIMD Extension en Modo 64 bits



Digital Media Boost... Que quiere decir?

- El procesador tiene dos ports para despachar a ejecución instrucciones SIMD.
- La mayoría de las instrucciones SIMD de enteros de 128 bits se ejecutan en un ciclo de clock.
- El core puede ejecutar hasta seis operaciones de punto flotante por ciclo.
- Hasta dos operaciones SIMD de enteros empaquetados en registros de 128 bits por ciclo.
- Safe Instruction Recognition (SIR): función que le permite retirar operaciones de punto flotante fuera de orden con respecto de instrucciones enteras.