Arquitectura de Computadores

Parte 1

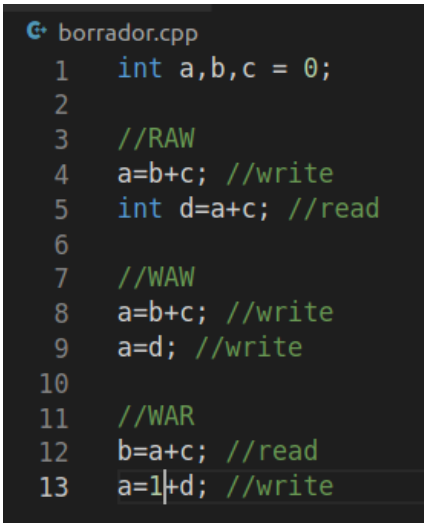
Carmen Azorín Martí

Lección 1: Clasificación del paralelismo implícito en una aplicación

Dependencias de datos

Las condiciones que se deben comulir para que un bloque de código A presente dependencia de datos con respecto a B son:

1. Deben hacer referencia a la misma posición de memoria.
2. B aparece en la secuencia de código antes que A.



Tipos de dependencias de datos:

- RAW (read after write) o dependencia verdadera.
- WAW (write after write) o dependencia de salida.
- WAR (write after read) o antidependencia.

Clasificaciones

Un código secuencial para la aplicación es una descripción estructurada, que presenta la ventaja para el programador de no ser una representación nueva que haya que aprender desde cero. Dentro del código secuencial encontramos **paralelismo implícito** en los siguientes niveles de abstracción:

1. **Nivel de programas**: no existe mucha dependencia entre ellos. Grano grueso.
2. **Nivel de funciones**: las funciones llamadas en un programa se pueden ejecutar en paralelo, siempre que no haya dependencias verdaderas (por ejemplo, RAW). Grano medio.
3. **Nivel de bucle**: se pueden ejecutar en paralelo las iteraciones de un bucle, siempre que no haya dependencias verdaderas. Para detectar las dependencias habrá que analizar las entradas y las salidas de las iteraciones del bucle. Grano medio-fino.
4. **Nivel de operaciones**: Las operaciones independientes (no hay dependencia de datos) se pueden ejecutar en paralelo. Grano fino.

Con granularidad nos referimos al tamaño de los trozos de código que se pueden ejecutar en paralelo.

Este paralelismo es conocido como paralelismo funcional.

Por otra parte, nos encontramos con otra clasificación del paralelismo: el de datos y el de tareas.

1. El **paralelismo de tareas** extrae la estructura lógica de funciones de la aplicación. En esta estructura, los bloques son funciones, y las conexiones entre ellos reflejan el flujo de datos entre funciones. Por ello, equivaldría al paralelismo a nivel de función.
2. El **paralelismo de datos** se encuentra implícito en las operaciones con estructura de datos (vectores y matrices). Las operaciones vectoriales y matriciales engloban varias operaciones que se pueden ejecutar en paralelo. El paralelismo de datos se puede extraer de los bucles.

Thread vs proceso

El hardware es el encargado de gestionar la ejecución de instrucciones. A nivel superior, el SO se encarga de gestionar la ejecución de unidades de mayor granularidad (procesos y hebras).

Cada proceso en ejecución tiene su propia asignación de memoria, razón por la cual, para comunicarse entre procesos hay que usar llamadas al SO. Además, los procesos deben comprender el código del programa, es decir, deben tener su propia tabla de ficheros abiertos, tabla de páginas, contenido de registros y sus propios datos de pila, segmentos, etc.

Una hebra tiene su propia pila y contenido de registros, pero comparte el código, variables globales y otros recursos con las hebras del mismo proceso. Por ello, las hebras tardan menos tiempo en crear procesos, destruir procesos, comunicarse y sincronizarse entre ellas, etc.

Por tanto, las **hebras tienen una granularidad menor que los procesos**.

El **paralelismo** implícito en el código de una aplicación se puede hacer **explícito a** distintos niveles:

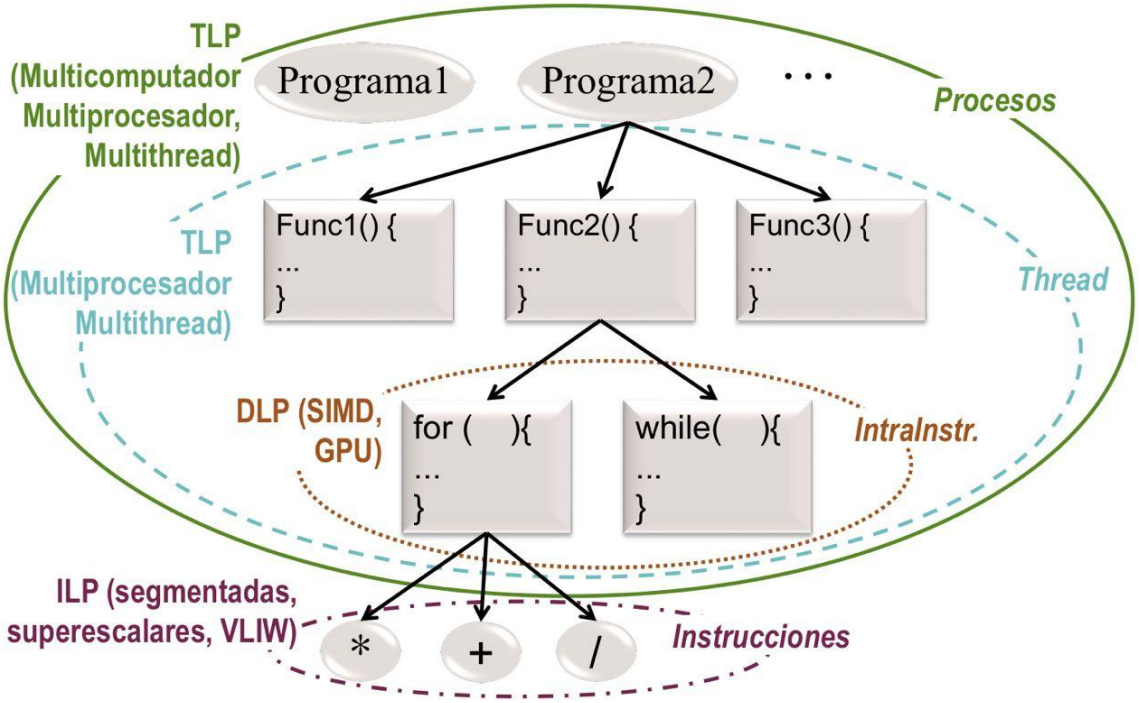
1. **Nivel de instrucciones**: la unidad de control de un procesador gestiona la ejecución de instrucciones por la unidad de procesamiento.
2. **Nivel de hebra**: es la menor unidad de ejecución que procesa el sistema operativo. Y es la menor secuencia de instrucciones que se puede ejecutar en paralelo.
3. **Nivel de proceso**: mayor unidad de ejecución que gestiona el SO. Un proceso consta de una o varias hebras.

Relación entre paralelismo implícito, explícito y arquitecturas paralelas

En el gráfico se relacionan los distintos niveles en los que se encuentra el paralelismo implícito en el código, con los niveles en los que se puede hacer explícito y con arquitecturas paralelas que aprovechan el paralelismo.

- El paralelismo entre programas se aprovecha en arquitecturas con paralelismo a nivel de procesos.

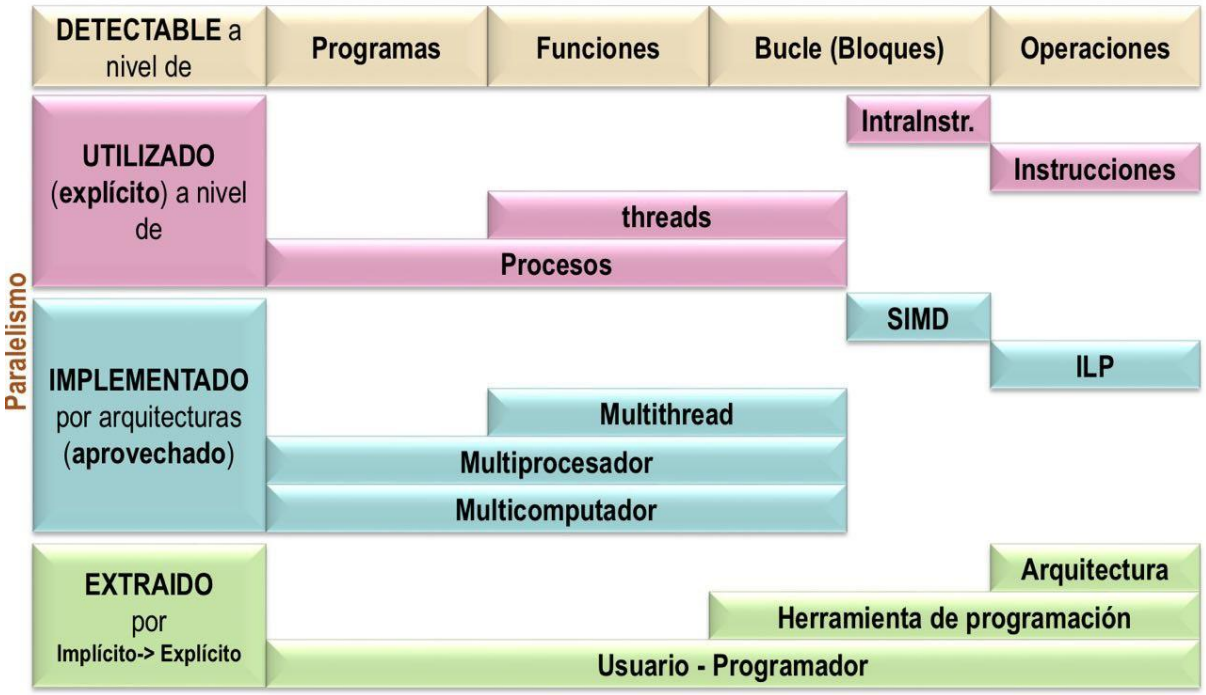
- El paralelismo entre funciones se aprovecha en arquitecturas con paralelismo a nivel de procesos y de hebras.
- El paralelismo en un bucle se puede extraer a nivel de procesos o de hebras. Además, también se puede hacer explícito dentro de una instrucción vectorial.
- El paralelismo entre operaciones se puede aprovechar en arquitecturas con paralelismo a nivel de instrucción.



Detección, utilización, implementación y extracción del paralelismo

- El paralelismo detectable a nivel de programas es utilizado por los procesos, implementado por arquitecturas multiprocesador y multicomputador, y extraído por el programador.
- El paralelismo detectable a nivel de funciones es utilizado por las hebras y los procesos, implementado por arquitecturas multihebra, multiprocesador y muticomputador, y extraído por el programador.
- El paralelismo detectable a nivel de bloques es utilizado por las instrucciones vectoriales, las hebras y los procesos, implementado por arquitecturas multihebra, multiprocesador, multicomputador y SIMD-vectoriales, y extraído por el compilador y el programador.
- El paralelismo detectable a nivel de operaciones es utilizado por las instrucciones, implementado por arquitecturas ILP (superescalares o segmentados), y extraído por la propia arquitectura, el compilador y el

programador (para ello, eliminan dependencias de datos no verdaderas).



Lección 2: Clasificación de arquitecturas paralelas

Computación paralela y computación distribuida

- Computación paralela: estudia los aspectos hardware y software relacionados con el desarrollo y ejecución de aplicaciones en un sistema de cómputo compuesto por **múltiples procesadores** que es visto externamente como **unidad autónoma** (multicores, multiprocesadores, multicomputadores, cluster).
- Computación distribuida: estudia los aspectos hardware y software relacionados con el desarrollo y ejecución de aplicaciones en una **colección de recursos autónomos** situados en **distintas localizaciones físicas**.

Clasificación de computadores según el segmento de mercado

El precio en un nivel del mercado depende decisivamente del número de procesadores que incluye el sistema y del grado de utilización de componentes disponibles comercialmente.

El **nivel más bajo** corresponde al mercado de **computadores de uso personal** (PC de sobremesa, portátiles y estaciones de trabajo). En su mayoría, tienen un procesador o dos (en el caso de configuraciones SMP).
En los **siguientes niveles** de mercado, nos encontramos sistemas a lo que se accede desde distintas localizaciones y que comparten varios usuarios (**servidores**). Los servidores se clasifican en tres grupos: básicos, de gama media y de gama alta.
En el **nivel más alto** se encuentran los **supercomputadores**, que ofrecen mayores prestaciones.

Clasificación de Flynn de arquitecturas
La taxonomía de Flynn divide los computadores en cuatro clases según si el computador procesa una o varias secuencias de instrucciones, que actúan sobre una o varias secuencias de datos:

1. **SISD**: un **único flujo de instrucciones** procesa operandos y genera resultados, definiendo un **único flujo de datos**.
    a. Una única unidad de control (UC) que recibe instrucciones de memoria, las decodifica y genera los códigos que definen la operación correspondiente a cada instrucción que debe realizar la unidad de procesamiento (UP) de datos.
    b. El flujo de datos para las operaciones se sacan de memoria y, posteriormente, se almacenan en memoria los resultados.
2. **SIMD**: un **único flujo de instrucciones** procesa operandos y genera resultados, definiendo **varios flujos de datos**, dado que cada instrucción codifica varias operaciones iguales, cada una sobre operadores distintos.
    a. Los códigos que genera una única unidad de control (UC) actúan sobre varias unidades de procesamiento (UPi). Por tanto, puede realizar varias operaciones similares simultáneas con operandos distintos.
    b. Cada una de las secuencias de operandos y resultados utilizados definen un flujo diferente de datos.
3. **MIMD**: el computador ejecuta **varias secuencias de instrucciones**, y cada una de ellas procesa operandos y genera resultados definiendo un único flujo de datos, de manera que existen **varios flujos de datos** (uno por cada flujo de instrucciones).
    a. Varias unidades de control decodifican las instrucciones correspondientes a distintos programas. Cada uno de esos programas procesa un conjunto de datos distintos, que definen distintos flujos de datos.

4. **MISD**: **varios flujos distintos de instrucciones** se ejecutan actuando sobre el mismo **flujo de datos**.
    a. **No existen** computadores MISD específicos, porque su comportamiento se puede implementar con iguales prestaciones en un computador MIMD.

Multiprocesadores vs multicomputadores

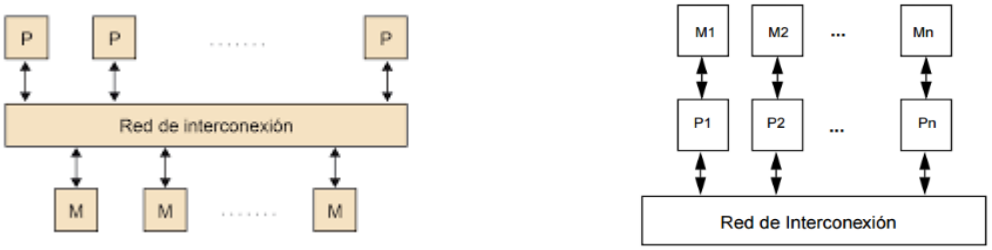
Los multiprocesadores se han clasificado según la organización del sistema de memoria. En particular, los sistemas con paralelismo de alto nivel se han clasificado según de la organización de su espacio de direcciones:

1. **Multiprocesadores (UMA/SMP)**: sistemas en los que todos los procesadores comparten el mismo espacio de direcciones. El programador no necesita conocer dónde están almacenados los datos.
    a. **Mayor latencia**: si añade un procesador, aumenta la latencia porque habrá más competencia a la hora de entrar en la memoria.
    b. **Poco estable**: si se añaden procesadores, no se puede asegurar que se vaya a reducir el tiempo de ejecución.
    c. Comunicación implícita mediante variables compartidas. Datos no duplicados en memoria principal.
    d. Necesita implementar primitivas de sincronización.
    e. Distribución código y datos entre procesadores no necesaria.
    f. Programación más sencilla.

Existen **multi-procesadores NUMA** que tienen la estructura de un multicomputador pero cada procesador puede acceder al espacio de memoria que necesite en cada momento. Esa es la diferencia con el multicomputador, ya que éste tiene la característica de que cada procesador puede acceder únicamente a su memoria. Sin embargo, cabe destacar que en un NUMA el procesador tarda menos en acceder al espacio de direcciones que tiene más cerca, diferencia con los UMA, donde todos los procesadores tardan el mismo tiempo en acceder a todas las posiciones de memoria. En conclusión, un **NUMA tiene más escalabilidad y más nivel de empaquetamiento y conexión que un UMA, pero menos que un NORMA.**

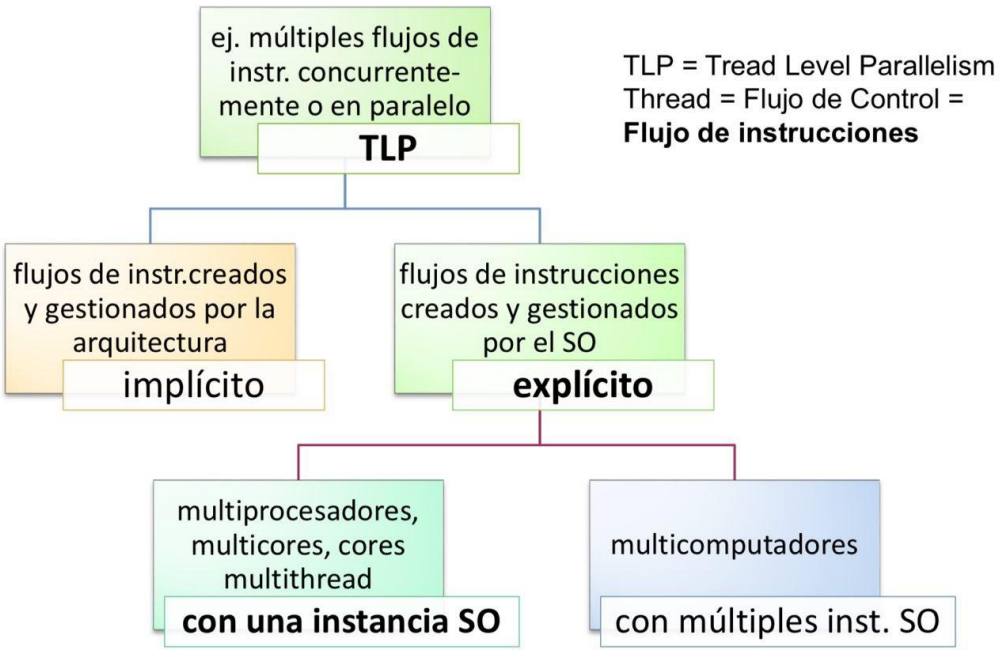
2. **Multicomputadores (NORMA)**: sistemas en los que cada procesador tiene su propio espacio de direcciones particular. El programador necesita conocer dónde están almacenados los datos.
    a. **Menor latencia**: si se añade un procesador, no afecta a la latencia, ya que cada procesador tiene su propio espacio de memoria y no debe pasar por la red.

b. **Más estable**: se pueden añadir procesadores y eso permite disminuir el tiempo de ejecución.
c. Comunicación explícita mediante software para paso de mensajes. Datos duplicados en memoria principal, copia datos.
d. Sincronización mediante software de comunicación.
e. Distribución código y datos entre procesadores necesaria, por tanto, se necesitan herramientas de programación más sofisticadas.
f. Programación más difícil.



En la izquierda vemos un multiprocesador UMA, en el que cada procesador debe pasar por la red de interconexión para acceder a la memoria compartida con el resto de procesadores.
En la derecha tenemos la estructura de un NUMA donde cada procesador puede acceder a su memoria local sin pasar por la red de interconexión. Sin embargo, puede acceder al resto de memorias accediendo a la red.
Un NORMA también tiene la estructura de la derecha. Pero cada procesador puede acceder únicamente a su memoria local. Por eso, los cluster son multicomputadores NORMA.
Un multiprocesador puede ser MISD y MIMD. Un multicomputador puede ser MIMD.

Clasificación de aquitecturas con múltiples flujos de instrucciones

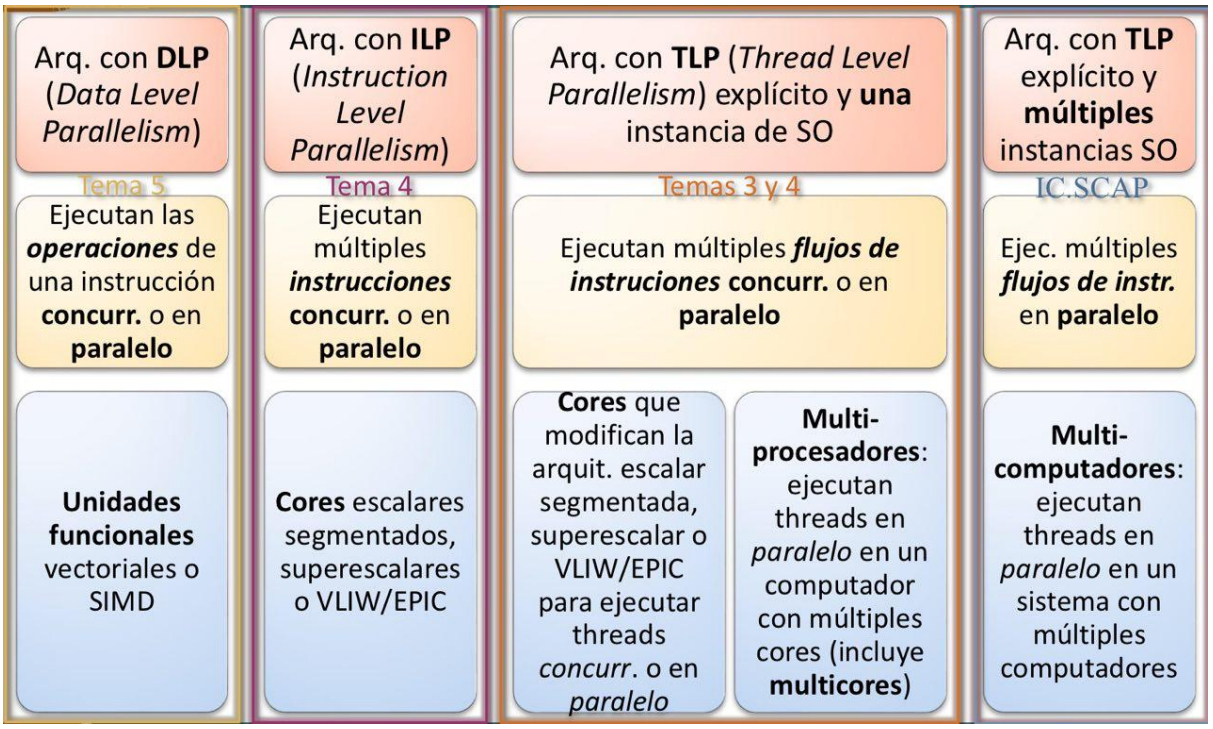


Se debe diferenciar entre concurrencia y paralelismo.
En la **concurrencia** hay más de una instrucción procesándose. Actúan distintos componentes del procesador a la vez. Es decir, cada instrucción esta ejecutándose en una etapa distinta, proceso conocido como **desacoplar**.

Por ejemplo, en la construcción de coches, mientras un coche monta las puertas, el otro coche monta las ruedas…
En el **paralelismo** lo que se hace es replicar el hardware, lo tenemos dos veces. Entonces, hay dos entradas procesándose a la vez.

En la construcción de coches, hay varías líneas de montaje y dos coches pueden estar montando las puertas a la vez.

Cristerios de clasificación de computadores



Lección 3: Evaluación de prestaciones de una arquitectura

Tiempo CPU (sistema y usuario) de Unix vs Tiempo respuesta

- Tiempo(CPU) = system + user -> no incluye todo el tiempo
- Tiempo(real) = elapsed time
    - Con flujo de instrucciones: elapsed >= t(CPU)
- Utilizando el comando **time** en Unix salen 3 tiempos:

- 1º Tiempo de CPU de usuario
- 2º Tiempo de CPU sistema (tiempo en kernel)
- 3º Tiempo transcurrido total != 1º+2º

$$\text{Tiempo de CPU } (T_{CPU}) = \text{Ciclos\_del\_Programa} \times T_{CICLO} = \frac{\text{Ciclos\_del\_Programa}}{\text{Frecuencia\_de\_Reloj}}$$

$$\text{Ciclos por Instrucción (CPI)} = \frac{Ciclos\_del\_Programa}{Numero\_de\_Instrucciones(NI)}$$

$$T_{CPU} = NI \times CPI \times T_{CICLO}$$

$$\text{Ciclos\_del\_Programa} = \sum_{i=1}^{n} CPI_i \times I_i$$

$$CPI = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{NI}$$

En el programa hay I_i instrucciones del tipo i (i=1,...n)

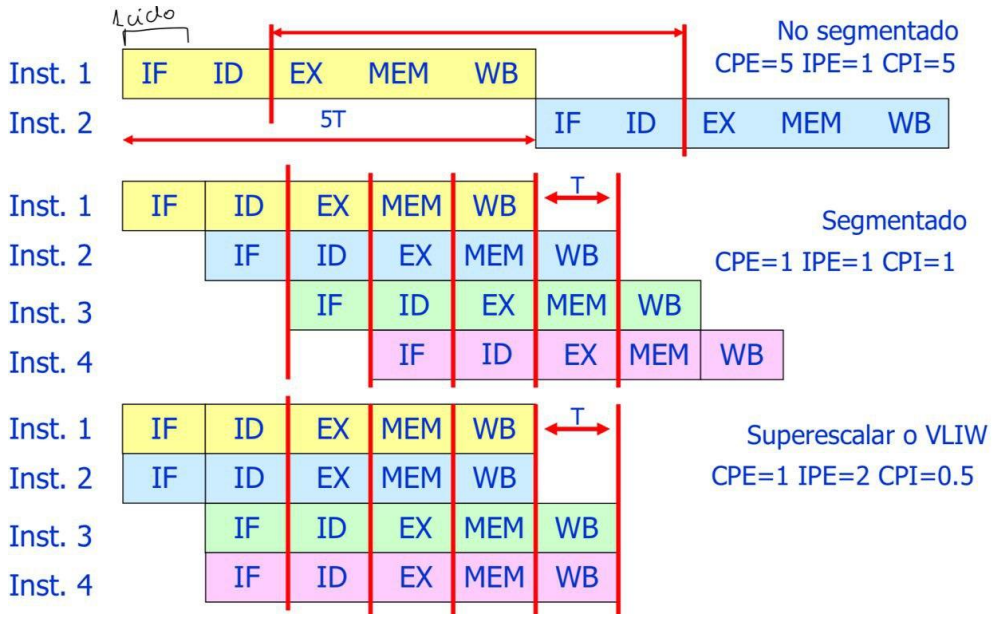
Cada instrucción del tipo i consume CPI_i ciclos

Hay n tipos de instrucciones distintos.

Hay procesadores que pueden lanzar para que empiecen a ejecutar varias instrucciones al mismo tiempo:

1. CPE: número mínimo de ciclos transcurridos entre los instantes en que el procesador puede emitir instrucciones.
2. IPE: instrucciones que pueden emitirse cada vez que se produce dicha emisión.

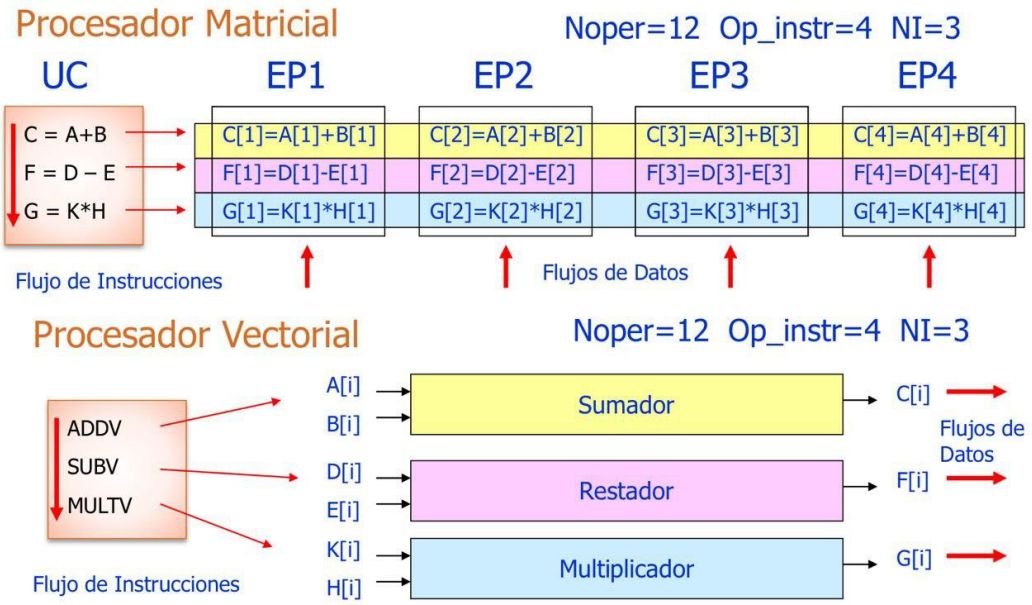
CPI = CPE/IPE



Hay procesadores que pueden codificar varias operaciones en una instrucción:

1. N_Operaciones: número de operaciones que realiza el programa.
2. Op_instr: número de operaciones que puede codificar una instrucción.

NI = N_Operaciones/Op_instr



En el tiempo ha habido mejoras en diferentes aspectos que han permitido mejorar el tiempo de CPU:

MEJORAS	NI	CPI	T(ciclo)
Tecnología			
Estr. y Organ.			
Repertorio instr.			
Compilador			

Calcular Tiempo CPU, GFLOPS y MIPS para medir la productividad

Para medir la productividad podemos utilizar los MIPS, los GFLOPS y el tiempo de ejecución. Sin embargo, a la hora de comparar programas:

1. Tiempo de ejecución: lo mejor para comparar
2. GFLOP: podemos usarlos para comparar, pero no nos dice en qué medida mejorar las prestaciones
3. MIPS: nunca usarlos para comparar

Vayamos viéndolos uno a uno:

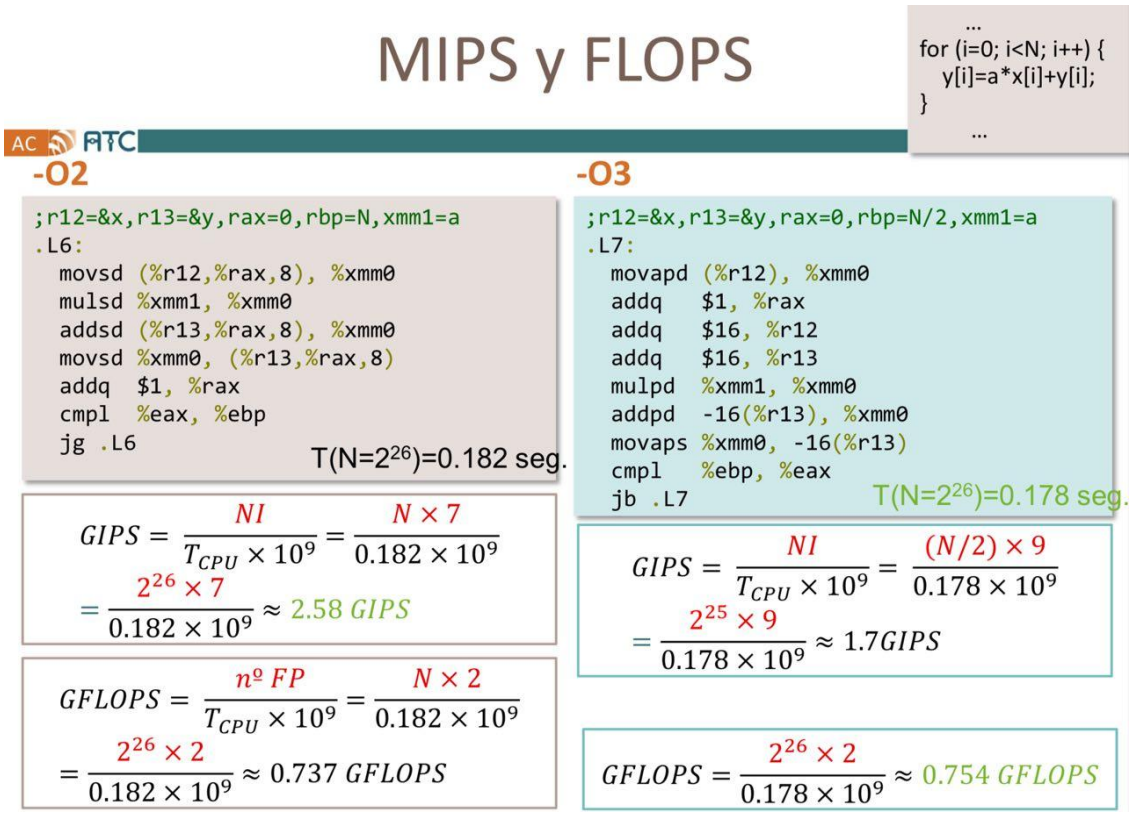
- **MIPS**: millones de instrucciones por segundo
  - Dependen del repertorio de instrucciones -> no sirven para comparar máquinas con distintos repositorios
  - Pueden variar con el programa -> no sirven para comparar máquinas
  - Pueden variar inversamente con las prestaciones -> cuantos más MIPS, mejores prestaciones

$$MIPS = \frac{NI}{T(CPU)*10^6} = \frac{f(Hz)\ (c/s)}{CPI*10^6\ (c/i)}$$

- **GFLOPS**: millones de operaciones en coma flotante por segundo
  - Sólo tiene en cuenta las operaciones en coma flotante -> no muy adecuado para comparar programas
  - Las operaciones en coma flotante no es constante en máquinas diferentes y la potencia de las operaciones no es igual para todas las operaciones -> hay que normalizar las instrucciones en coma flotante
  - cuantos más GFLOPS mejores prestaciones

$$GFLOPS = \frac{Operaciones\ en\ coma\ flotante}{T(CPU)*10^9}$$

- **Tiempo respuesta:**
  - Cuanto menos tiempo de respuesta mejores prestaciones



Ganancia de Prestaciones

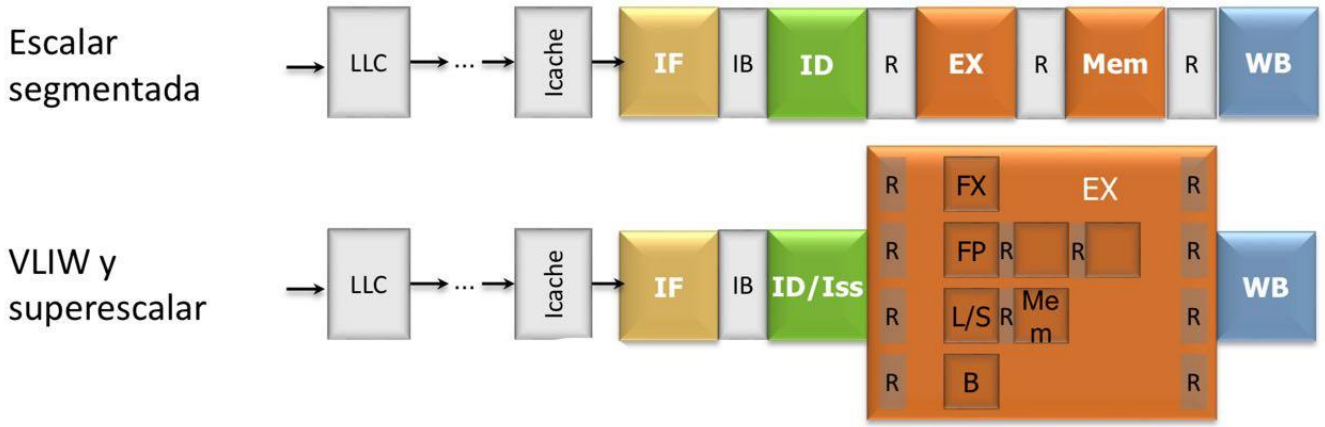
Si se incrementan las prestaciones de un sistema, el incremento en prestaciones (velocidad) que se consigue en la nueva situación (**p**) con respecto a la previa (**b**) se expresa mediante la Ganancia en prestaciones (**S**)

$$S = \frac{Vp}{Vb} = \frac{Tb}{Tp} = \frac{NI_b * CPI_b * Tciclo_b}{NI_p * CPI_p * Tciclo_p}$$

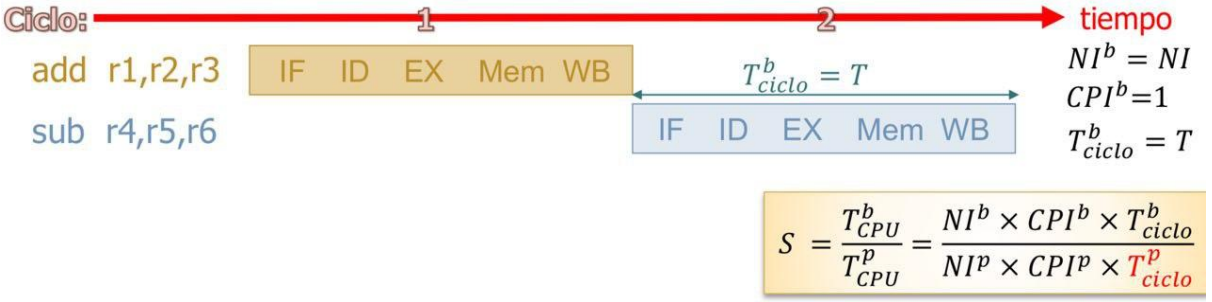
- Arquitecturas con paralelismo a nivel de instrucción (ILP)
  - Escalar segmentada
  - VLIW y superescalar

  Tienen las siguientes etapas de ejecución:
  1. Instruction Fetch: captación de instrucciones
  2. Instruction Decode: decodificación de instrucciones y emisión a unidades funcionales (incluye captura de operandos)
  3. Execution y Memory: ejecución y acceso a memoria
  4. Write-Back: almacenamiento de resultados en registros

  La diferencia entre segmentada y escalar radica en que en la tercera etapa (de ejecución), se pueden ejecutar N instrucciones de distinto tipo en paralelo.

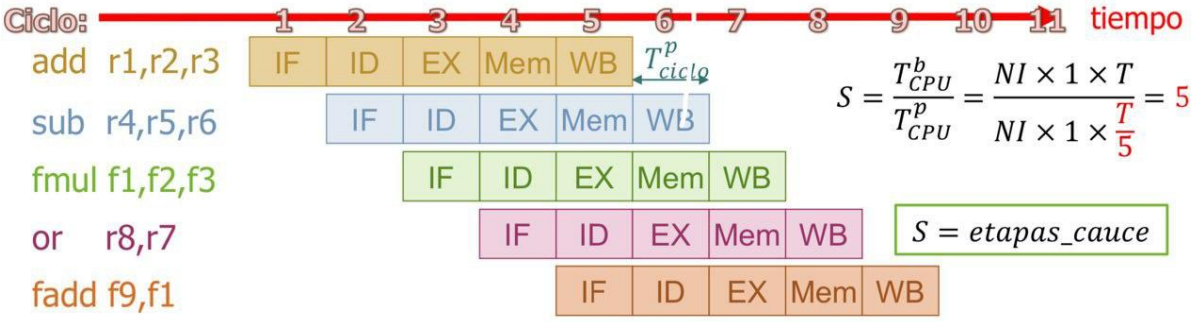


Ejemplo de la mejora en un núcleo de procesamiento:



$$S = \frac{T^b_{CPU}}{T^p_{CPU}} = \frac{NI^b \times CPI^b \times T^b_{ciclo}}{NI^p \times CPI^p \times T^p_{ciclo}}$$

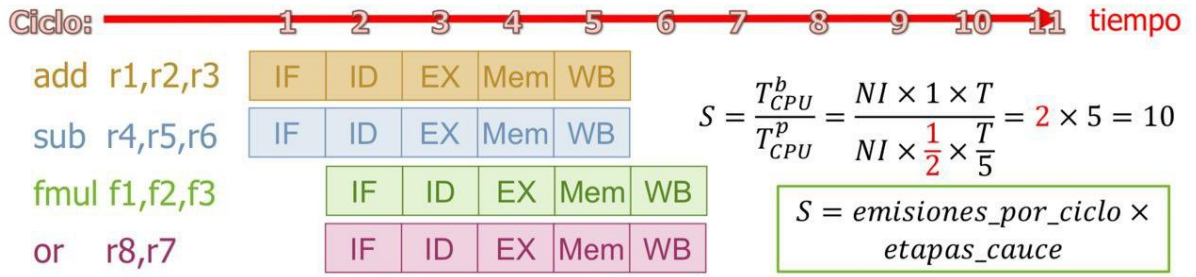
Núcleo **segmentado** en **5 etapas**:



$$S = \frac{T^b_{CPU}}{T^p_{CPU}} = \frac{NI \times 1 \times T}{NI \times 1 \times \frac{T}{5}} = 5$$

$$S = etapas\_cauce$$

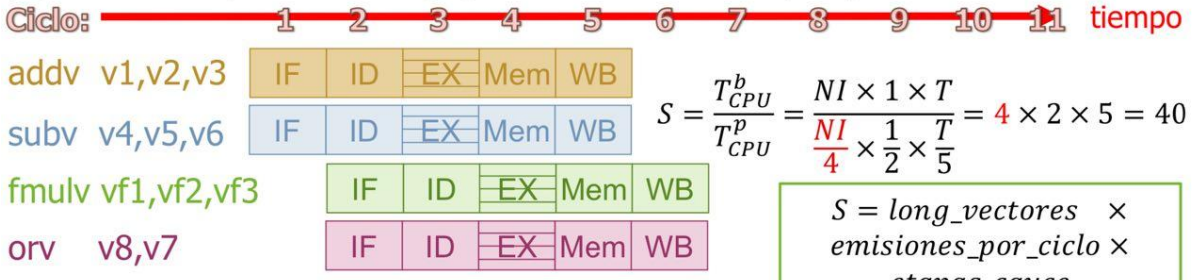
Núcleo **superescalar** con **2 emisiones por ciclo** y **5 etapas**:



$$S = \frac{T^b_{CPU}}{T^p_{CPU}} = \frac{NI \times 1 \times T}{NI \times \frac{1}{2} \times \frac{T}{5}} = 2 \times 5 = 10$$

$$S = emisiones\_por\_ciclo \times etapas\_cauce$$

Núcleo **superescalar** con **2 emisiones por ciclo** y **5 etapas**, y
unidades funcionales SIMD (vectoriales) que procesan **vectores de 4 componentes**
(suponemos el mejor caso: el código genera sólo instrucciones que operan con vectores de 4 com



$$S = \frac{T^b_{CPU}}{T^p_{CPU}} = \frac{NI \times 1 \times T}{\frac{NI}{4} \times \frac{1}{2} \times \frac{T}{5}} = 4 \times 2 \times 5 = 40$$

$$S = long\_vectores \times emisiones\_por\_ciclo \times etapas\_cauce$$

Ley de Amdahl

La mejora de velocidad (**S**), que se puede obtener cuando se mejora un recurso de una máquina en un factor **p** está limitada por:

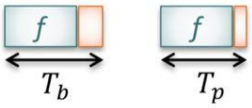
$$S = \frac{Vp}{Vb} = \frac{Tb}{Tp} \leq \frac{1}{f+\frac{1-f}{p}} = \frac{p}{1+f(p-1)}$$

Ésta última fracción tiende a $\frac{1}{f}$ cuando $x \to \infty$ y tiende a p cuando $f \to 0$

Donde **f** es la fracción de tiempo de ejecución del sistema sin la mejora durante el que no se usa dicha mejora.

- f no puede ser mayor que 0

Ejemplo: Si un programa pasa un 25% de su tiempo de ejecución en una máquina realizando instrucciones de coma flotante, y se mejora la máquina haciendo que estas instrucciones se ejecuten en la mitad de tiempo, entonces ***p*=2, *f*=0.75 y**



$$S = \frac{T_b}{T_p} = \frac{1}{0.75 + \frac{0.25}{2}} \approx 1.14$$

Habría que mejorar el caso más frecuente (lo que más se usa)

Benchmark

Propiedades exigidas a medidas de prestaciones:

1. Fiabilidad: representativas, evaluar diferentes componentes del sistema y reproducibles
2. Permitir comparar diferentes realizaciones de un sistema o diferentes sistemas: aceptadas por todos los interesados (usuarios, fabricantes, vendedores)

Interesados:

1. Vendedores y fabricantes de hardware o software
2. Investigadores de hardware o software
3. Compradores de hardware o software

Tipos de Benchmark:

1. De bajo nivel o microbenchmark
2. Kernels
3. Sintéticos
4. Dhrystone y Whetstone
5. Programas reales
6. Aplicaciones diseñadas

El LINPACK es un núcleo para clasificar las máquinas del Top500 en función de sus prestaciones.