

AC 21/22 - Tema 1: Arquitecturas paralelas, clasificación y prestaciones

1. Introducción

Tanto el diseño de nuevos procesadores como el desarrollo de programas eficientes requieren de evaluar las prestaciones de un sistema. En este capítulo vamos a ver formas de obtener esas métricas, que ponen de manifiesto las relaciones existentes entre el tiempo de la CPU y parámetros como el número de instrucciones, capacidad de la microarquitectura o número de instrucciones por ciclo. También estudiaremos los cuellos de botella y el uso de la Ley de Ahmdal para razonar acerca de éstos.

2. Tiempo de CPU

Tiempo de respuesta y de CPU

Tiempo de respuesta: tiempo transcurrido entre el inicio de la ejecución de un programa y la obtención de los resultados

- Tiempo de CPU: contenido en el tiempo de respuesta, es el tiempo que el procesador pasa ejecutando instrucciones del programa.

El resto del tiempo de respuesta, no incluido en el Tcpu es, por ejemplo, tiempo de E/S o tiempo que el procesador dedica a otros programas distintos durante la ejecución.

Es decir: el tiempo de respuesta puede ser un indicador de las prestaciones de un ordenador, pero puede verse afectado por operaciones de E/S o por la carga del procesador en el momento de la ejecución. Nos centraremos en el Tcpu, considerando siempre Te/s despreciable.

$$T_{CPU} = \text{Ciclos del programa} \cdot T_{ciclo} = \text{Ciclos del programa} / \text{frecuencia}$$

Es decir, Tciclo es el periodo y frecuencia es frecuencia y; como sabemos, $f = 1/T$

- Ciclos de programa: número de ciclos de reloj del procesador que tarda el programa en ejecutarse.

$$\text{Ciclos de programa} = \text{Num instr} \cdot \text{Ciclos por instrucción}$$

- Frecuencia: frecuencia del reloj

$$f = 1/T_{ciclo}$$

Por tanto, otra expresión para el Tcpu sería:

$$T_{CPU} = CPI \cdot NI \cdot T_{ciclo} = (NI \cdot CPI) / f$$

Especificación

En una máquina en la que todas las instrucciones requieren el mismo número de ciclos, su CPI medio puede calcularse de la siguiente manera:

$$\text{CPI} = (\text{sumatorio from i:0 to W: num instr distintas } [NI \cdot CPI]) / NI$$

Normalmente las instrucciones del programa se codifican en una sola instrucción máquina. Pero para repertorios que aprovechan el paralelismo de datos esto no es necesariamente cierto, así que podemos calcular su NI de la siguiente manera:

$$NI = \text{Num operaciones del programa} / \text{Num operaciones que puede codificar cada instr}$$

$$NI = Noper / OPI$$

El CPI suele depender de las características de la arquitectura y su disposición física. Pero hay ocasiones (procesador segmentado NO superescalar) en las que el compilador es decisivo para el aprovechamiento de los recursos, en cuyo caso nos interesaría expresar el CPI a partir de las características de la microarquitectura.

Podemos usar el num medio de ciclos que tienen que transcurrir desde que se emiten una o varias instr, hasta que se puede realizar otra emisión (llamado CPE); y el número medio de instr que se emiten (IPE), de forma que:

$$\text{CPI} = \text{num medio ciclos entre emisión de una instr hasta sig emisión} / \text{num medio instr emitidas}$$

$$\text{CPI} = \text{CPE} / \text{IPE}$$

- En un procesador no segmentado (las instr se ejecutan de una en una), CPE es igual al número medio de ciclos de reloj que tarda en procesarse la instr; y IPE es 1 porque las instr van de una en una.
- En un procesador segmentado, el valor máximo de CPE es 1 porque cada ciclo podrían empezar a emitirse instrucciones. Si el procesador segmentado sólo puede empezar a ejecutar una instrucción por ciclo, se tiene:

$$\text{CPI} = 1/\text{IPE} = 1/1 = 1$$

Lo cual sería un caso ideal, dado que en los procesadores segmentados normalmente no es posible emitir una instrucción por ciclo (por problemas con dependencias, por ejemplo). Así que, normalmente, $\text{CPE} > 1$ y por tanto $\text{CPI} > 1$. Cuanto más cercano a 1 sea el CPI, mejor estamos aprovechando el cauce del procesador.

En el caso de procesadores superescalares o VLIW, donde se pueden emitir varias instr por ciclo, el CPI puede ser menor que 1.

Conclusión final tras estas aclaraciones:

$$T_{cpu} = NI \cdot CPI \cdot T_{ciclo} = (Noper/OPI) \cdot (CPE/IPE) \cdot T_{ciclo}$$

3. Medidas de velocidad de procesamiento y benchmarks

Referidas a la velocidad de ejecución de instrucciones

MIPS: Millones de instrucciones por segundo

$$\text{MIPS} = \text{NI} / (\text{Tcpu} * 10^6) = \text{NI} / (\text{NI} * \text{CPI} * \text{Tciclo} * 10^6) = 1 / (\text{CPI} * \text{Tciclo} * 10^6) = \text{F} / \text{CPI} * 10^6$$

Con la tecnología actual, no es raro sustituir el 10^6 por 10^9 para obtener GIPS, y así no tener que manejar MIPS tan grandes.

Cuando trabajamos con MIPS hay que tener en cuenta que es un cálculo específico al repertorio: una máquina de repertorio RISC necesita muchas más instr para ejecutar el mismo programa que una CISC así que, incluso si el Tcpu de las dos máquinas fuera el mismo para un mismo programa (ambas máquinas son igualmente eficientes), el MIPS de la máquina RISC sería mayor y caeríamos en error si dijéramos que por ello es mejor. Los MIPS miden la velocidad con la que cada procesador ejecuta las instrucciones de su repertorio - por ello muchas veces se refiere a ellos como la velocidad pico del procesador.

MFLOPS: Millones de operaciones float por segundo

$$\text{MFLOPS} = \text{operaciones float} / \text{Tcpu} * 10^6$$

De nuevo, es común sustituir el 10^6 por 10^9 y obtener GFLOPS; por 10^{12} , TFLOPS; por 10^{15} , PFLOPS; por 10^{18} , EFLOPS...

Benchmarks

Tanto cuando evaluamos MIPS como FLOPS, si nos referimos a los valores pico de esas magnitudes hay que emplear un programa o cjto de programas sobre los que hacer medidas de tiempo y NI o NIfloat.

Para tener un marco común de programas de pruebas para evaluar prestaciones, tenemos benchmarks. Estos varían según el fin de la evaluación o el tipo de recurso que se va a evaluar. Algunos ejemplos conocidos son SPEC, TPC, Linpacks.

4. Estimación de tiempo de CPU

Hay veces que es necesario estimar el Tcpu que puede necesitar un programa dadas las características del computador en el que se ejecuta, el cjto de operaciones, el patrón de acceso a memoria del programa...

Las expresiones dadas hasta ahora asumen que los datos a los que se accede están en el nivel 1 de la caché - por eso el CPI suele ser pequeño. Pero si los datos están en memoria o más profundos en la caché el tiempo puede aumentar considerablemente. Podemos obtener una expresión del Tcpu mínimo con la expresión del Tcpu: ese es el tiempo mínimo que va a necesitar el programa, sin fallos de caché y con el procesador a pleno rendimiento.

Pero si tenemos información de las características de la jerarquía de memoria de la máquina, podemos tener una mejor estimación del tiempo mínimo de CPU, porque puede existir un solapamiento casi total entre el tiempo de ejecución de instrucciones en el procesador, y el tiempo de acceso a memoria principal (para load o store, para los que se producen los fallos de caché): es decir, el tiempo mínimo será el mayor de estos dos tiempos:

$$T_{ejec_min} = \max(T_{cpu}, T_{memoria})$$

$$T_{memoria} = \text{Num accesos a memoria} \cdot \text{tiempo medio de acceso a memoria}$$

$$T_{memoria} = N_{acc} \cdot t_{mem}$$

El tiempo de acceso a memoria en una caché look-through (mira toda la caché y si no está ahí lo que quiere, va a memoria) se calcula:

$$t_{mem} = a_1 \cdot t_1 + (1 - a_1) \cdot (t_1 + t_m)$$

- a_1 - tasa de aciertos de la caché, la probabilidad de que un acceso a memoria esté en caché
 - Para estimar la tasa de aciertos se determina el número de accesos a memoria y en cuántos de ellos falla
- t_1 - tiempo de acceso a caché L1
- t_m - tiempo de acceso a memoria
 - Determinado por la arquitectura

Si además del acceso tenemos en cuenta el tiempo necesario para reemplazar una línea de caché con el contenido que traemos desde memoria, la fórmula queda:

$$t_{mem} = a_1 \cdot t_1 + (1 - a_1) \cdot (t_1 + t_m + p_{reemplazo} \cdot t_{linea})$$

- $p_{reemplazo}$ - probabilidad de que, cuando se produzca una falta, haya que cambiar una línea de la caché actualizando en memoria la línea reemplazada
 - A partir del patrón de accesos de memoria y las características de la caché se puede determinar también la probabilidad de reemplazo
- t_{linea} - tiempo necesario para actualizar la línea
 - Determinado por la arquitectura

Para mejorar las prestaciones de la jerarquía de memoria se puede:

- Reducir los tiempos de acceso a los distintos niveles de caché
- Reducir tasa de fallos a niveles de caché
- Reducir la penalización cuando se producen fallos
 - Técnicas basadas en inclusión de recursos (cachés de víctimas o pseudo-asociativas)
 - Procedimientos a nivel de programación como precaptación u optimización de código (mezcla de arrays, fusión de bucles, operaciones con submatrices (blocking))

5. Ganancia de velocidad y ley de Amdahl

Para medir el resultado de una mejora se utiliza la ganancia de velocidad, que compara la velocidad de una máquina antes y después de los cambios. La expresión a ganancia de velocidad es:

$$Sp = V_p / V_1 = (W/T_p) / (W/T_1) = T_1 / T_p$$

- V_1 - velocidad previa a la mejora
- V_p - velocidad posterior a la mejora
- W - carga de trabajo que se aplica, es la misma post y pre mejora
- T_1 - Tcpu previo a la mejora
- T_p - Tcpu posterior a la mejora

La ley de Amdahl establece una cota superior a la ganancia que se puede conseguir mejorando alguno de los recursos en un factor p y según la frecuencia con la que se utiliza ese recurso:

LEY DE AMDAHL

$$Sp \leq 1 / (1 + f \cdot (p-1))$$

- Sp - ganancia
- f - fracción del tiempo de ejecución ANTES DE APLICAR LA MEJORA en un recurso QUE NO EMPLEA LA MEJORA
 - O sea, si hay una mejora en el 25% del código, $f=0.75$
 - Así, si $f=1$ es que no se aprovecha la mejora en ningún punto del código y Sp será menor o igual a 1, así que no hay mejora de velocidad
 - Si $f=0$ es que la mejora se aprovecha en todo el código y Sp puede alcanzar un valor igual al factor p

AC 21/22 - Tema 2: Programación paralela

1. Introducción

En este capítulo abordaremos los siguientes temas relacionados con la programación paralela:

1. Las herramientas existentes para escribir código paralelo y el trabajo extra que implica para las herramientas y el programador la paralelización
2. Las estructuras típicas de flujos de instr en códigos paralelos
3. Cómo evaluar las prestaciones del código paralelo implementado y qué prestaciones se pueden esperar

2. Herramientas de programación paralela

2.1. Trabajo a realizar por las herramientas de programación paralela o por el programador

La programación paralela requiere que la herramienta de programación utilizada, el programador o ambos realicen el siguiente trabajo, no requerido en la programación secuencial:

- **Localización del paralelismo implícito** en una aplicación, es decir, descomponer la aplicación en unidades de cómputo independientes, o **tareas**.
Es recomendable terminar la descomposición con un grafo dirigido en el que los vértices sean tareas y los arcos dependencias entre ellas. Nos permitirá ver fácilmente el número máximo de tareas que se pueden ejecutar en paralelo, o **grado de paralelismo** (número de flujos de instr o procesadores máximo necesario para la parallelización).
- **Asignación de las tareas o carga de trabajo** a flujos de instrucciones, y asignación de flujos de instr a procesadores. No suele ser rentable usar más flujos que procesadores. Los flujos pueden estar asociados a los procesadores previa asignación. La asignación puede ser:
 - Estática o dinámica
 - Con una asignación estática el reparto de tareas es siempre el mismo; mientras que con la dinámica el reparto puede variar y solo se conoce al final de la ejecución.

La asignación dinámica implica código extra y sincronización/comunicación entre hebras, lo cual introduce retardos adicionales. Pero es la única opción cuando no conocemos el número de tareas previamente.

- Implícita (herramienta) o explícita (programador)
- **Comunicación/sincronización** entre los flujos de instr. Los flujos tienen que colaborar, enviarse los resultados que producen...

(a) Cálculo de Pi con C

```

main(int argc, char **argv) {
    double ancho, sum=0;
    int intervalos, i;
    intervalos = atoi(argv[1]);
    ancho = 1.0/(double) intervalos;
    for (i=0;i< intervalos; i++){
        x = (i+0.5)*ancho;
        sum = sum + 4.0/(1.0+x*x);
    }
    sum* = ancho; }
    ...
}

#include <omp.h>
#define NUM_THREADS 4
main(int argc, char **argv) {
    double ancho,x, sum=0; int intervalos, i;
    intervalos = atoi(argv[1]);
    ancho = 1.0/(double) intervalos;
    omp_set_num_threads(NUM_THREADS); →Crear y Terminar
#pragma omp parallel →Comunicar y sincronizar
Localizar{
    #pragma omp for reduction(+:sum) private(x)
    for (i=0;i< intervalos; i++) {
        x = (i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x);
    }
    sum* = ancho;
}
    ...

(b) Cálculo de Pi con OpenMP/C
(c) Cálculo de Pi con MPI/C

```

Grafo de dependencias entre tareas

0,1,...,intervalos-1

Figura 1. (a) Código secuencial C para el cálculo de Pi y códigos paralelos en C usando (b) OpenMP (API+Directivas+Funciones), estándar de-facto para programación con el estilo de variables compartidas y (c) MPI (API+funciones), estándar de-facto para el estilo de paso de mensajes. Las modificaciones realizadas al código secuencial (a) para obtener código paralelo se han destacado en negrita en (b) y (c)

Las herramientas permiten, de forma implícita (lo hace la herramienta) o explícita (lo hace el programador), las siguientes **LABORES**:

1. Localizar/Detectar paralelismo, o sea, descomponer la aplicación en tareas independientes (*decomposition*).
2. Asignar las tareas, la carga de trabajo (instr+datos) a flujos (*scheduling*).
3. Crear y terminar flujos de instr, o enrolar y desenrolar en un grupo flujos previamente creados.
4. Comunicar/Sincronizar flujos de instr.
5. Asignar flujos de instr a unidades de procesamiento (*mapping*).

La última labor la puede hacer el SO o el hardware, depende del sistema

2.2. Nivel de abstracción en que sitúan al programador las herramientas

Cuanto mayor sea el nivel de abstracción, menos labores de las enunciadas anteriormente dependerán del programador. Hay herramientas que permiten escoger el nivel de abstracción. La labor más difícil para la herramienta es la primera, localizar el paralelismo. Cuantas más labores haga el programador, mayores prestaciones se pueden llegar a conseguir si el programador conoce la arquitectura del sistema de cómputo.

Las herramientas de paralelismo se pueden clasificar según su nivel de abstracción, de menos a más:

- **Compiladores paralelos:** Ej. compiladores de Intel. Generan código paralelo a partir de código secuencial. El programador sólo tiene que implementar el código secuencial, no realiza ninguna de las labores previas.
- **Lenguajes paralelos y APIs de funciones y directivas:** en este nivel, el programador debe al menos detectar el paralelismo (labor 1). El programador no hace el reparto (labor 2), y pueden librarse al programador de asignar flujos a procesadores (labor 5), saber como se crean/terminan flujos (labor 3) o detalles de comunicación y sincronización (labor 4).
 - Lenguajes paralelos, como Occam, Ada, Java; tienen construcciones específicas y bibliotecas de funciones. Requieren un compilador exclusivo.
 - APIs de func y directivas, como OpenMP, OpenACC; constan de directivas y una biblioteca de funciones que se añaden a un compilador de un lenguaje secuencial normal, como C/C++ o Fortran.
- **APIs de funciones:** Ej. pthreads, MPI. Consisten en una biblioteca de funciones que se añaden a un compilador de lenguaje secuencial. El programador debe hacer explícitamente la asignación de tareas (labor 1), y participar en todas las labores menos, probablemente, la última.
- **Lenguajes paralelos para arquitecturas de propósito específico:** ej. CUDA. Consisten en construcciones de lenguaje y bibliotecas de funciones. Requieren un compilador exclusivo. El programador debe participar en todas las labores excepto, probablemente, la última; y debe tener conocimientos mayores sobre la arquitectura de destino para poder escribir el código paralelo.

2.3. Estilos de programación paralela

Cada herramienta ofrece al programador un modelo de programación particular. Las arquitecturas paralelas se caracterizan por el estilo de programación más cercano a su implementación hardware: paso de mensajes para multicomputadores (Ej. MPI); variables compartidas para multiprocesadores y núcleos multithread (Ej. OpenMP); y paralelismo de datos para procesadores que ejecutan una instrucción en paralelo en múltiples unidades de procesamiento (SIMD, Single Instruction Multiple Data) (Ej. HPF). Teniendo eso en cuenta, es fácil ver qué caracteriza a cada uno de estos modelos de programación:

- **Paso de mensajes:** el programador debe tener en cuenta que los flujos de instrucción no comparten memoria, cada uno de ellos tiene su espacio de direcciones en particular. La herramienta debe ofrecer medios para copiar datos de un espacio de direcciones a otro.
- **Variables compartidas:** como los flujos comparten memoria, la información puede pasarse a través de variables. La herramienta debe ofrecer (si no lo hace ya implícitamente) medios para sincronizar los flujos con el fin de que las comunicaciones a través de variables no tengan problemas: la sincronización debe conseguir que el consumidor de un dato lo lea después de que el productor lo escriba en la variable compartida, no antes.
- **Paralelismo de datos:** el programador debe tener en cuenta que todos los flujos de datos ejecutan a la vez la misma instrucción con datos distintos.

2.4. Comunicación/Sincronización en herramientas de programación paralela

Las herramientas de comunicación implementan comunicación uno-a-uno, es decir, envío de datos desde un emisor hasta un receptor. También pueden ofrecer comunicaciones colectivas de distinto tipo, que evitan tener que implementar estas comunicaciones con múltiples envíos uno-a-uno.

Las comunicaciones colectivas se pueden clasificar así:

- **Comm uno-a-todos:** uno envía y todos reciben (el que envía puede estar incluido con los que reciben). Difusión o dispersión (*Broadcast, Scatter*).
- **Comm todos-a-uno:** todos envían y uno recibe (el que recibe puede estar incluido con los que envían). Acumulación o reducción (*Gather, Reduction*).
- **Comm todos-a-todos:** todos difunden (*all-broadcast*), todos dispersan (*all-scatter*)
- **Comm múltiples uno-a-uno:** permutaciones (rotación, barajar...), todos los flujos envían a uno y todos reciben de uno, desplazamientos...
- **Comm colectivas compuestas de varias de las anteriores:**
 - Todos combinan (*all reduction*)
 - Recorrido (*scan*)
 - Barreras (*barrier*)
 - Punto en el código en el que todos los flujos se esperan entre sí. Cuando todos llegan a la barrera, continúan a la vez con la ejecución. Se compone de una reducción y una difusión.

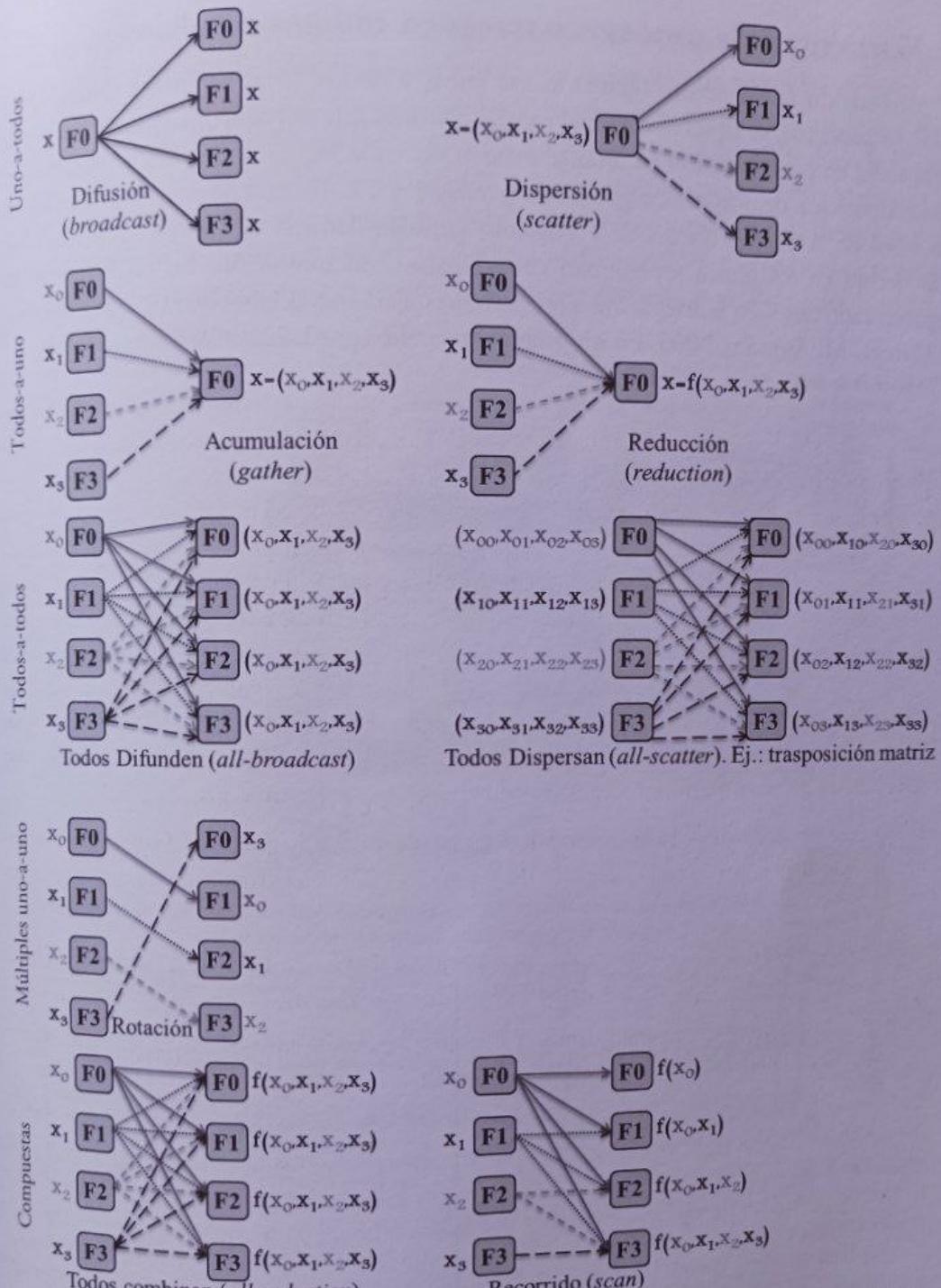
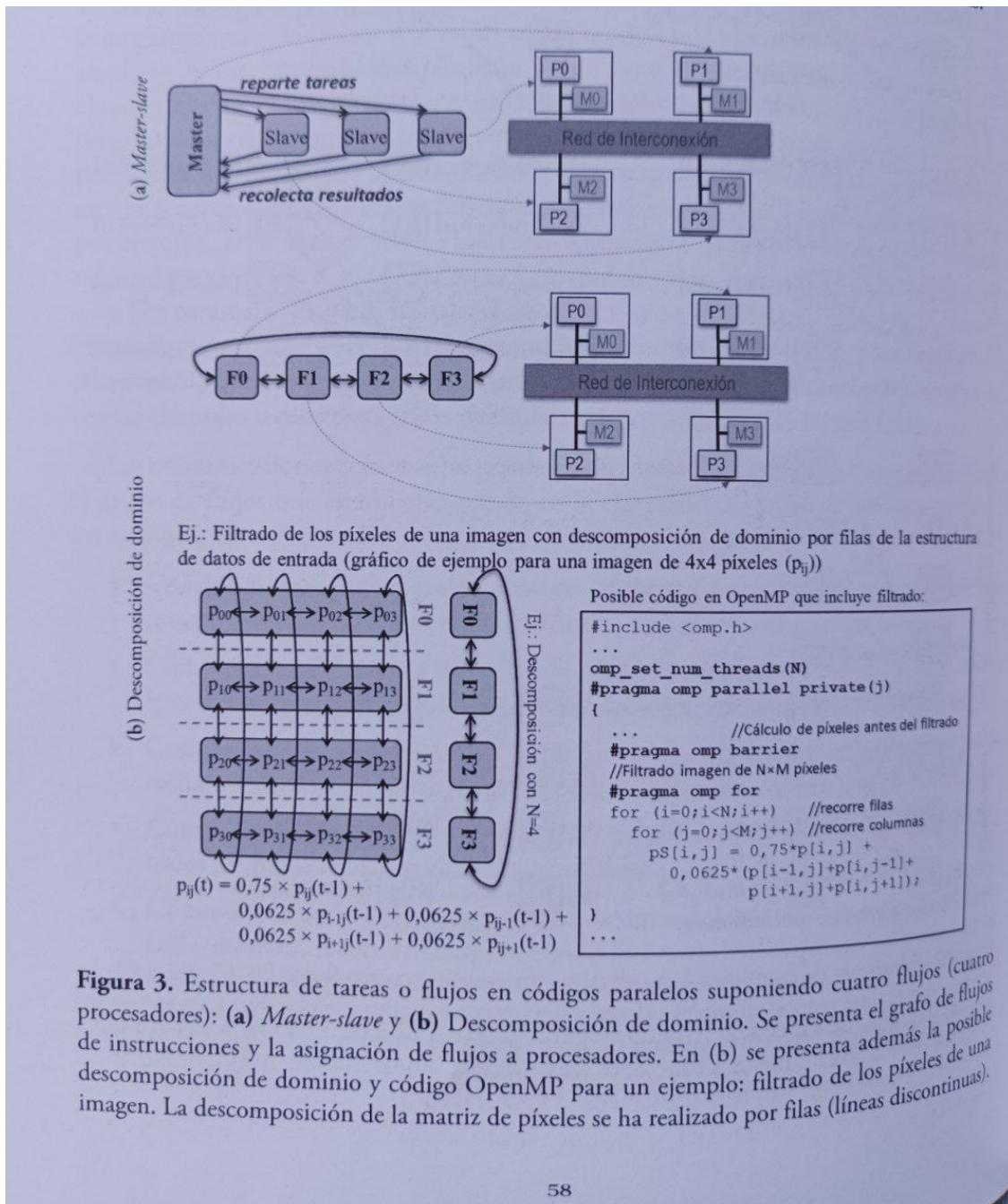


Figura 2. Funciones de comunicación colectivas. Intervienen cuatro flujos F0, F1, F2 y F3. La operación $f()$ que reduce múltiples valores a uno es commutativa y asociativa. Cuando en los ejemplos un flujo recibe o envía un dato (x_i o x_{ij}) se podría también recibir o enviar múltiples datos (x_i o x_{ij} podrían ser vectores).

3. Estructuras de procesos o tareas en códigos paralelos

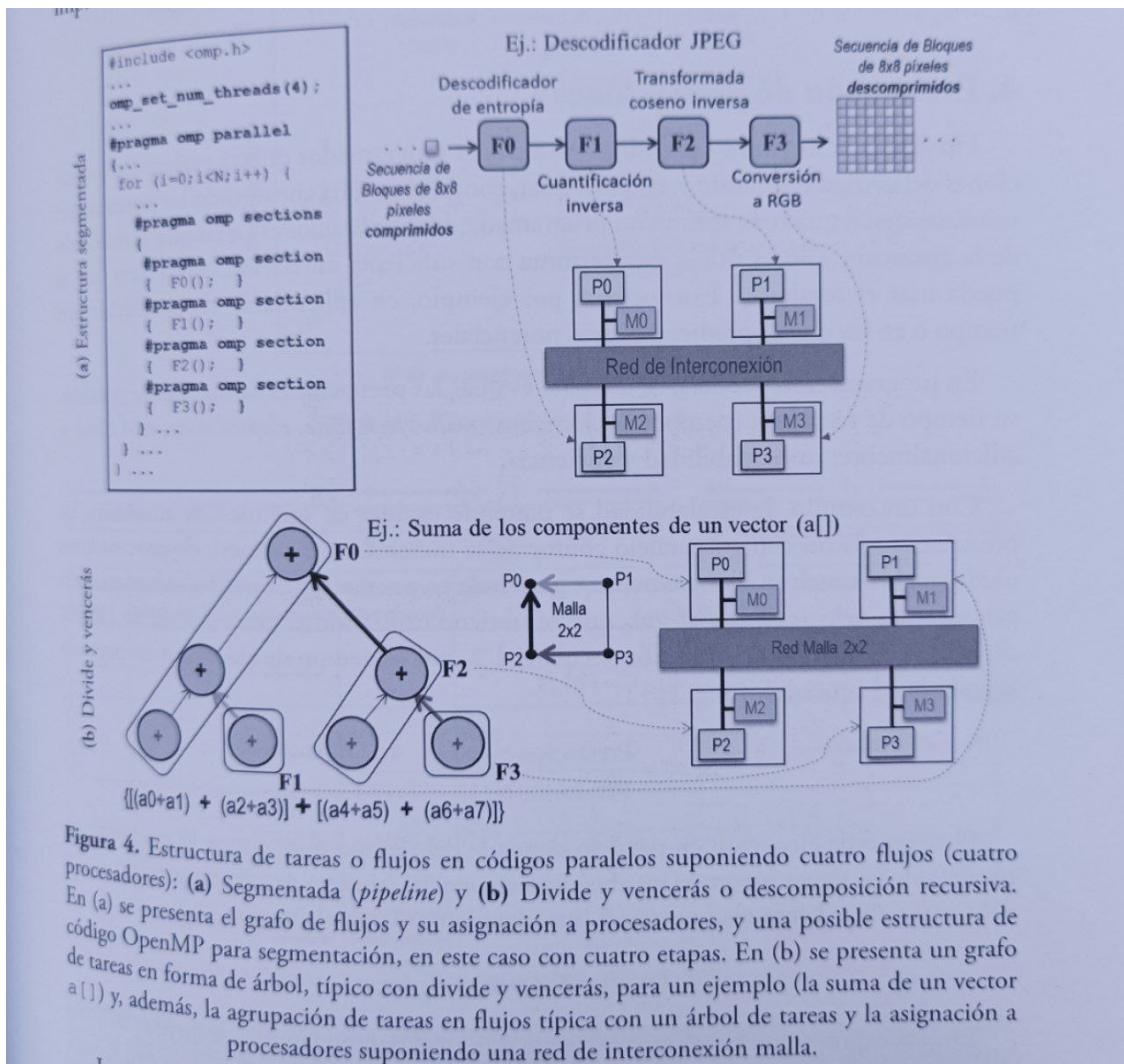
Analizando la estructura o grafo de las tareas y de los flujos de instr de los códigos paralelos, podemos ver ciertos patrones que se repiten. En un programa paralelo puede haber varias estructuras. Las estructuras más comunes son:

- **Descomposición del dominio y Maestro-Esclavo:** el trabajo a realizar por cada proceso se determina dividiendo las estructuras de datos de entrada o las de salida en trozos, tanto como flujos (descomposición del dominio), y luego cada flujo gestiona sus tareas y reporta el resultado a una hebra master.
- Si el flujo se divide en entradas, el trabajo a asignar al flujo i será el que utilice las entradas del flujo i .
- Si el flujo se divide en salidas, el trabajo a asignar al flujo i será el que genere las salidas del trozo i .



- **Cauce segmentado (*pipeline*):** se utiliza cuando se aplica a un flujo de datos de entrada la misma secuencia de instrucciones, una detrás de otra, como una cadena de montaje.

- **Divide y vencerás:** se usa cuando el problema a resolver se puede dividir en subtareas que son instancias más pequeñas del problema original, de forma que combinando los resultados de las subtareas se resuelve el problema original. Las subtareas también se pueden dividir en subtareas, dando lugar a una estructura en forma de árbol.



4. Evaluación de prestaciones

Tras escribir un código paralelo, hay que comprobar que se cumplen los requisitos de tiempo. Para evaluar las prestaciones del código, se evalúa su tiempo de respuesta (desde el inicio de ejecución hasta obtener los resultados).

Un estudio de escalabilidad analiza en qué medida aumentan las prestaciones del código conforme aumenta el número de procesadores. La ganancia en prestaciones (o ganancia en velocidad) se calcula:

$$S(\text{num procesadores}) = \text{Prestaciones}(\text{num procesadores}) / \text{Prestaciones}(1 \text{ procesador})$$

$$S(\text{num procesadores}) = \text{tiempo ej. secuencial} / \text{tiempo ej. paralelo con } p \text{ procesadores} = T_s / T_p(1)$$

El tiempo de ejecución en paralelo se calcula:

$$T_p(p, \text{tam problema } n) = \text{Tiempo calculo paralelo}(p, n) + \text{Tiempo de sobrecarga/overhead }(p, n)$$

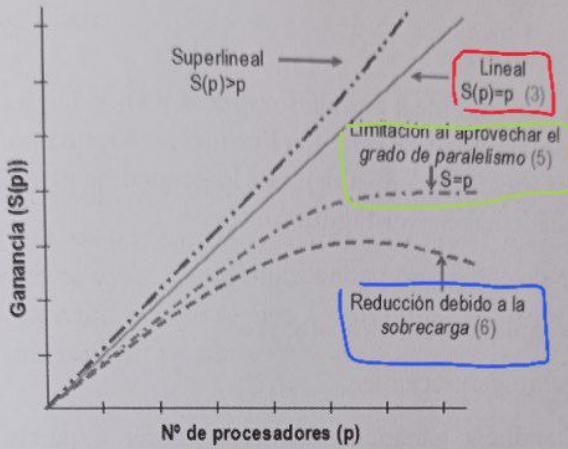
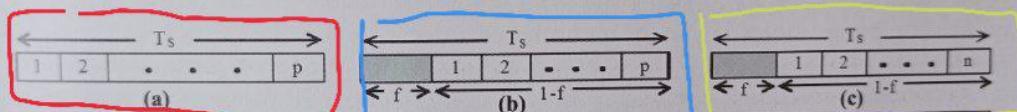


Figura 5. Curvas de escalabilidad



Modelo código	Fracción no paral. en T_s	Grado paralelismo	Sobrecarga (Overhead)	Ganancia en función del número de procesadores p con T_s constante
(a)	0	ilimitado	0	$S(p) = \frac{T_s}{T_p(p)} = \frac{T_s}{T_s/p} = p$ Ganancia lineal (3)
(b)	f	ilimitado	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p \rightarrow \infty} \frac{1}{f}$ Ley de Amdahl (4)
(c)	f	n	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p=n} \frac{1}{f + \frac{(1-f)}{n}}$ (5)
(b)	f	ilimitado	Incrementa linealmente con p	$S(p) = \frac{1}{f + \frac{(1-f)}{p} + \frac{T_0(p)}{T_s}} \xrightarrow{p \rightarrow \infty} 0$ (6)

Figura 6. Ganancia en prestaciones para diferentes modelo de código: (a) Todo el código se puede repartir por igual entre los p procesadores disponibles, sea cual sea p (i.e. el grado de paralelismo es ilimitado), (b) hay una fracción f del código secuencial (del tiempo de ejecución secuencial T_s) que no se puede paralelizar y el resto ($1-f$) se puede repartir entre los p procesadores disponibles, sea cual sea p , (c) hay una fracción f del código secuencial que no se puede paralelizar y el resto ($1-f$) se puede dividir hasta n trozos (como ocurre en un trozo de código con un bucle de n iteraciones).

61

En el mejor caso se esperaría obtener un tiempo de ejecución del tiempo secuencial, dividido entre el número de procesadores. Para llegar a este tiempo, se debe...

1. Poder repartir todo el código entre los procesadores disponibles, sea cual sea este número (paralelismo ilimitado)
2. Hacer el reparto asignando a cada procesador la misma cantidad de trabajo (carga equilibrada)
3. No añadir sobrecarga

La escalabilidad en tal caso sería lineal (en rojo en la figura anterior). En la práctica, la curva de escalabilidad no será una línea recta, pero puede acercarse para un rango de p . Por lo general, la ganancia está debajo de la ideal por...

1. Una fracción f no es paralelizable
2. El reparto de trabajo no está equilibrado
3. Existe sobrecarga no despreciable

La ganancia además deja de crecer a partir de un cierto número de p (en verde en la figura), de hecho si añadimos demasiados podemos generar mucha sobrecarga y afectar negativamente al rendimiento (azul en la figura).

Pero incluso si el grado de paralelismo es ilimitado y la sobrecarga no existe, **la ganancia está limitada** (caso b, segunda fila de la tabla), porque la limita la fracción no paralelizable f, que se mantendrá siempre constante. El tiempo de ejecución nunca podrá ser menor que el tiempo de ejecución secuencial de ese fragmento no paralelizable.

$$1/\text{fracción no paralelizable} = \text{tiempo secuencial} / (\text{f no paral.} * \text{tiempo secuencial})$$

La ley de Amdahl (fórmula en la segunda fila de la tabla) dice que la ganancia de prestaciones que se puede conseguir aplicando paralelismo está limitada por la fracción no paralelizable del mismo. Ahmdal da una visión pesimista (la ganancia es limitada) porque no creía en la viabilidad del paralelismo masivo pero, en la práctica, la fracción de código secuencial no paralelizable **se puede reducir si aumenta mucho el tamaño del problema** - Aumentando el tamaño se mejora la calidad de los resultados. Esto nos lleva a pensar que la paralelización de un código puede tener dos objetivos:

- Disminuir el tiempo de ejecución (nuestro enfoque hasta ahora)
- Aumentar el tamaño del problema a resolver para mejorar la calidad del resultado

Si el objetivo al aumentar p es mejorar la calidad de los resultados mediante aumento de tamaño y no el tiempo, se puede obtener una ganancia de prestaciones que **no está limitada** y depende linealmente de p:

$$S(p) = Ts / Tp = (f \cdot Tp + p \cdot (1-f) \cdot Ts) / Tp = f + p \cdot (1-f) = Ts / (f * Ts + ((1+f)*Ts) / p)$$

Esta expresión es la **ganancia estable**, de Gustafson. Mientras que Amdahl asume que el Ts se mantendrá constante aun si incrementamos la p, concluyendo pues que la ganancia está limitada por la f, Gustafson mantiene cte el Tp y muestra que la ganancia puede crecer con pendiente constante relativa al número de procesadores p.

Para evaluar en qué medida se aproximan [las prestaciones que ofrece una máquina para un programa paralelo] a las [prestaciones máximas que idealmente ofrecería esa máquina, según su num de procesadores], se usa la siguiente expresión de **eficiencia**:

$$E(p) = \text{prestaciones}(p, n) / (p * \text{prestaciones}(1, n)) = S(p) / p$$

$$S(p) = \text{tiempo ej. secuencial} / \text{tiempo ej. paralelo con } p \text{ procesadores} = Ts / Tp(p)$$

AC 21-22, tema 4: arquitecturas con paralelismo a nivel de instrucción

1. Introducción

Mejora de la tecnología nos lleva a mejoras en los transistores, más pequeños y rápidos. La velocidad en microprocesadores llega de la mano de dos estrategias: aumento de la frecuencia del reloj, y diseño de microarquitecturas capaces de completar varias instrucciones por ciclo (superescalares).

EXPRESIONES VELOCIDAD DE PROCESAMIENTO:

$$V_{cpu} = NI / T_{cpu} = NI / (NI \cdot CPI \cdot T_{ciclo}) = 1 / (CPI \cdot T_{ciclo}) = \text{instr por ciclo} \cdot \text{freq}$$

La frecuencia F y el num instr ciclo IPC han permitido aumentar la velocidad exponencialmente, como dicta la Ley de Moore, hasta los primeros años del sXXI donde se empieza a estancar. En ese momento surgen los primeros sistemas multicore.

En este tema vamos a analizar como estos sistemas multicore nos permiten el **paralelismo entre instrucciones ILP**. Los sistemas multicore se basan en la segmentación de cauce o pipelining, según la cual el procesador sigue varias etapas que procesan independientemente las distintas fases de una instrucción.

Ejemplo: etapa de captación IF (instruction fetch); decodificación ID que toma lo obtenido de la fase previa y obtiene sus operandos; etapa de ejecución EX y así.

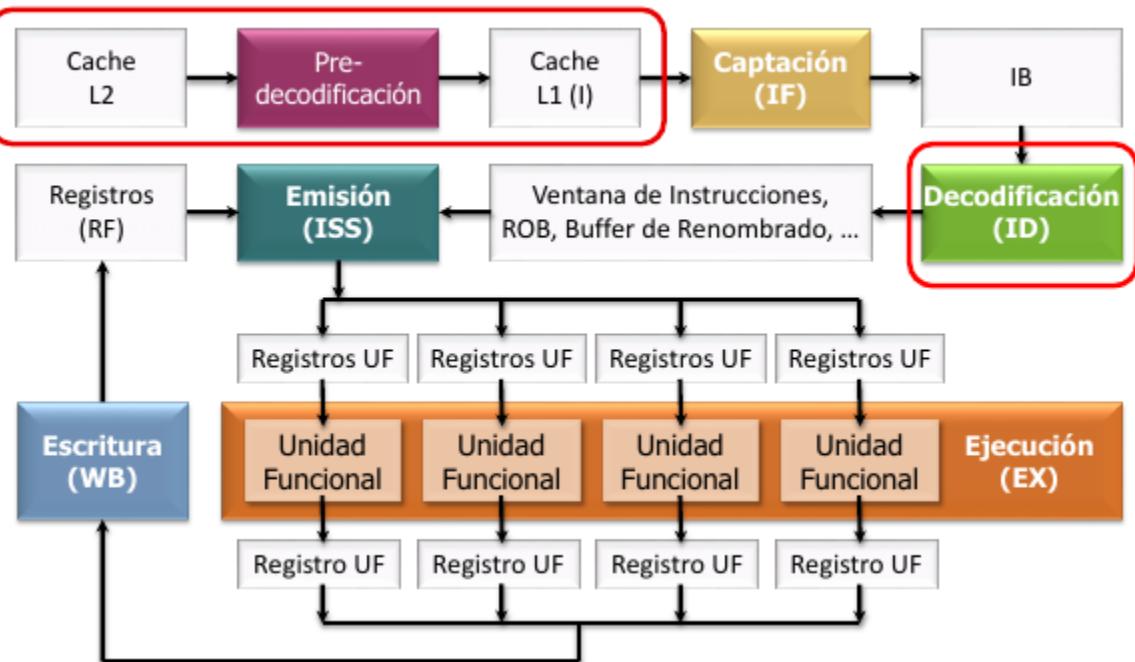
Las etapas definen un cauce en el que cada etapa trabaja con la salida de la anterior y genera resultados para la posterior. Así, se puede aprovechar el ILP, procesando distintas etapas de distintas instrucciones simultáneamente. Pero hay que tener cuidado con las dependencias de datos, dependencias de control (salto incondicional) y colisiones entre instrucciones que necesitan el mismo recurso a la vez. Estos problemas generan esperas que reducen el rendimiento del cauce.

Las arquitecturas que nos van a permitir segmentación de cauce son dos:

- Superescalares, mediante el hardware el procesador se encarga de ordenar instrucciones, renombrar operandos, predecir saltos... para el mejor uso posible del cauce y obtener el máximo IPC posible para la arquitectura.
- Very Long Instruction Word VLIW, mediante el compilador se ordenan las instrucciones para evitar dependencias y mejor uso del cauce. Dado que en el momento de compilar no contamos con toda la información del procesamiento de las instrucciones, se han desarrollado múltiples estrategias para aumentar el rendimiento, mayormente basadas en el procesamiento especulativo.

2. Procesadores superescalares

Un procesador superescalar es un procesador segmentado que puede finalizar más de una instrucción por ciclo mediante recursos hardware para extraer el paralelismo entre instrucciones dinámicamente.



Todas las etapas de un procesador superescalar pueden procesar varias instr por ciclo.

2.1. Procesamiento de instr en un cauce superescalar

Para explicar las distintas etapas vamos a basarnos en la siguiente tabla de ejemplo.

Instrucción	Significado	Ciclos para ejecución EX
1. add r3, r1, r2	$r3=r1+r2$	1
2. mul r3, r3, r4	$r3=r3*r4$	3
3. sw (r5), r3	$M(r5)=r3$	1
4. lw r5, (r6)	$r5=M(r6)$	2
5. add r6, r5, r4	$r6=r5+r4$	1
6. add r5, r3, r4	$r5=r3+r4$	1

- **Precaptación:** añade algunos bits a instrucciones cuando pasan desde la memoria o niveles de caché inferiores (L2 L3...) a la memoria caché de primer nivel L1. Con ello aceleramos la decodificación o algunas instr complejas empiezan a ser procesadas desde su captación.
- **Captación IF:** toma instr de la caché y las pasa a la **cola de instr**. Estas se captan en el orden que traen desde memoria (el que tienen en el código).
- **Decodificación ID:** desde la cola de instr se decodifican en orden. De ahí pasan a la **ventana de instr/estación de reserva**.

En la figura siguiente vemos el progreso de las instr de la tabla hasta ahora durante los cuatro primeros ciclos.

Instrucción			1	2	3	4	..
1	add	r3,r1,r2	IF	ID			..
2	mult	r3,r3,r4	IF	ID			..
3	sw	(r5),r3	IF		ID		..
4	lw	r5,(r6)		IF	ID		..
5	add	r6,r5,r4		IF		ID	..
6	add	5,r3,r4		IF		ID	..

Cola de instrucciones

1	add	r3,r1,r2	sw	(r5),r3	add	r6,r5,r4
2	mult	r3,r3,r4	lw	r5,(r6)	add	r6,r5,r4
3	sw	(r5),r3	add	r6,r5,r4		
4			add	r5,r3,r4		
5						

Tras ciclo 1

Tras ciclo 2

Tras ciclo 3

Figura 2. Captación y decodificación de instrucciones y evolución de la cola de instrucciones

Las instrucciones se emitirán desde la ventana de instrucciones a sus unidades funcionales correspondientes, donde se ejecutarán si sus operandos están disponibles y la unidad funcional está ociosa.

Ventana de instrucciones

Cada línea de la ventana de instrucciones tiene varios campos: el código de la operación (del que se saca la unidad funcional que le corresponde), el destino en el que se almacenará el resultado y dos campos por operando, que dependen de sus respectivos campos "OK": el campo operando contendrá el valor correcto si el campo OK está a 1. Si no, contendrá el sitio al que acceder para hacerlo cuando el campo OK deje de ser 0.

Vamos a ver la ventana de instrucciones para la secuencia de instrucciones planteada en la primera tabla:

#	Cod op	Reg dest	Oper 1	OK1	Oper2	OK2
1	add	r3	[r1]	1	[r2]	1
2	mult	r3	r3	0	[r4]	1
...						

Esta tabla de la ventana de instrucciones corresponde al final del ciclo 2 en la figura superior. Observamos como el OK1 de la operación mult está a 0, por que el r3 no será válido hasta que no se ejecute la suma previa.

#	Cod op	Reg dest	Oper 1	OK1	Oper2	OK2
1	mult	r3	[r3]	1	[r4]	1
2	sw	<i>no hay destino pq va a memoria</i>	[r5]	1	r3	0
3	lw	r5	[r6]	1	<i>solo 1 operando</i>	/
	...					

La ventana de instrucciones tras el ciclo 3. La instrucción de suma ha pasado a ejecución porque sus operadores y su unidad funcional estaban disponibles, así que se elimina y la multiplicación queda la primera de la cola. La operación de suma tarda un ciclo, así que hasta el final del ciclo 3 no conoceremos el resultado de r3. La multiplicación se ejecutará en el ciclo 4 porque tendrá todos sus operadores disponibles.

#	Cod op	Reg dest	Oper 1	OK1	Oper2	OK2
1	sw	<i>no hay destino pq va a memoria</i>	[r5]	1	r3	0
2	lw	r5	[r6]	1	<i>solo 1 operando</i>	/
3	add	r6	r5	0	[r4]	1
4	add	r5	r3	0	[r4]	1

La ventana de instrucciones tras el ciclo 4. Mult se está ejecutando y sabremos su resultado al final de este mismo ciclo, se han introducido las dos últimas sumas (que antes se estaban descodificando).

Emisión ordenada/desordenada

Como vemos, las instrucciones se ejecutan en orden, es una **emisión ordenada**. Es por ello que la instrucción de carga lw no se ejecuta, incluso si sus operandos y unidad están disponibles. Si se permitiera que lw se ejecutara, se trataría de emisión desordenada. Hay que destacar un supuesto:

- sw escribe en [r5]
- lw lee de [r6]
- Si sucede que [r5] = [r6], la instrucción lw debería esperar a sw, porque si no se cometería una dependencia tipo RAW.
 - Esto puede evitarse bien porque se conoce lo que [r5] y [r6] o bien asumiendo que la posibilidad de que [r5] = [r6] es baja, en cuyo caso estaríamos cayendo en un **adelantamiento especulativo**.
 - En este último caso, es obvio que el sistema debe contar con recursos para deshacer el efecto de la instrucción lw adelantada, por si se diera el caso de que,

finalmente, las direcciones sí son iguales. Lo describiremos con más detalle más adelante.

Estaciones de reserva

Hasta ahora hemos hablado de ventana de instrucciones, pero lo más común es contar con las llamadas **estaciones de reserva**: cumplen la misma función, pero dividiendo el contenido de la ventana en estructuras más pequeñas. Lo más común es contar con estaciones de reserva según la unidad funcional a la que vamos a mandar la instrucción. Tras la decodificación, la instrucción manda a la reserva correspondiente. Es frecuente un esquema como:

- Dos estaciones de reserva para el procesador: uno para las operaciones enteras y otro para las operaciones en coma flotante
- Dos estaciones de reserva para la memoria: una para la lectura de datos desde memoria y otra para la escritura

De hecho, la ventana de instrucciones corresponde al caso de **una única estación de reserva centralizada**.

Cuando tenemos un esquema basado en múltiples estaciones de reserva, a la decodificación ID hay que añadirle un paso extra, será una operación de decodificación y emisión (issue), ID/ISS. El inicio de ejecución desde una reserva se denomina "envío" o *dispatch*.

Ejemplo - emisión ordenada/desordenada

Volviendo a la tabla de ejemplo, estas son las trazas de una emisión ordenada y una emisión desordenada. Asumimos que las unidades funcionales siempre están disponibles y que hay suficientes para una ejecución sin colisiones (hay dos sumadores, pero solo una ud. de acceso a memoria).

Emisión ordenada:

instr	1	2	3	4	5	6	7	8	9	10	...
1. add r3, r1, r2	IF	ID	EX								
2. mul r3, r3, r4	IF	ID		EX	EX	EX					
3. sw (r5), r3	IF		ID				EX				
4. lw r5, (r6)		IF	ID					EX	EX		
5. add r6, r5, r4		IF		ID						EX	
6. add r5, r3, r4		IF		ID						EX	

Emisión desordenada: cambios en negrita

instr	1	2	3	4	5	6	7	8	9	10	...
1. add r3, r1, r2	IF	ID	EX								
2. mul r3, r3, r4	IF	ID		EX	EX	EX					
3. sw (r5), r3	IF		ID				EX				
4. lw r5, (r6)		IF	ID	EX	EX						

instr 5. add r6, r5, r4	1	2 IF	3	4 ID	5	6 EX	7	8	9	10	...
6. add r5, r3, r4		IF		ID		EX					

Dependencias, buffer de renombramiento

Como vemos, en la emisión ordenada la ejecución de una instrucción comienza, como muy pronto, simultáneamente a la finalización de una instrucción que la precede en orden. La emisión desordenada termina en menos ciclos, pero para permitir estos adelantos el procesador ha seguido algunas estrategias:

- Primero, las dependencias de datos RAW (Read after write, una instrucción necesita un operando después de que este sea generado). Las instrucciones 2,1; 3,2; y 5,4 la presentan. Se soluciona como ya hemos comentado con el campo OK.
 - Las dependencias WAR y WAW no son realmente dependencias porque simplemente son resultado de usar el mismo registro para guardar varias cosas a lo largo de la ejecución, podría solucionarse simplemente usando otro registro como destino. Esta técnica se denomina **renombramiento** de registros.
 - El renombramiento puede aplicarlo el compilador o el hardware. Esto es lo usual en el caso de procesadores superescalares, donde se incluye una estructura de **buffers de renombramiento**. Por ejemplo:
 - Entrada válida, si está a 1 marca que en esa entrada se está utilizando para hacer un renombrado
 - Registro de destino indica el registro que se ha renombrado en esa línea
 - Valor es el espacio reservado para el renombrado, la nueva ubicación
 - Valor válido indica si lo guardado en el nuevo valor es válido
 - Último indica si en la línea está contenido el nuevo renombramiento que se ha hecho para el registro

El buffer de renombramiento se consulta al captar los operandos para pasarlo a la ventana de instrucción/estación de reserva: antes de comprobar si los operandos están en el banco de registros, se comprueba si se ha hecho algún renombramiento en el buffer de renombramiento. Si no hay renombramiento del registro, el operando se toma del banco de registros.

#	Entrada válida	Reg destino	Valor	Valor válido	Último
1	1	r3	351	1	0
2	1	r3	-	0	1
3	1	r5	-	0	1
4					

"Recordatorio" de la traza de la ejecución, para estudiar más cómodamente su interacción con el buffer de renom:

instr	1	2	3	4	5	6	7	8	9	10	...
r3, r3, r4	IF	ID		EX	EX	EX					
3. sw (r5), r3	IF		ID				EX				
4. lw r5, (r6)		IF	ID					EX	EX		
5. add r6, r5, r4		IF		ID						EX	
6. add r5, r3, r4		IF		ID						EX	

Se muestra el estado del buffer de renom tras el ciclo 3. Al inicio de la ejecución se encontraba vacío:

1. La instr 1 escribe sobre r3 y, al estar vacío el buffer de renom, toma el r2 y r1 desde el banco de registros. Al final del ciclo 3, se ha hecho la suma y ha dado (por ejemplo) 351.
2. En valor metemos 351, y ponemos valor válido a 1.
3. La inst2 termina de decod en el ciclo 2 y da lugar a un renombramiento de r3 (línea 2 del buffer renom). Como es el último renom a r3, cambiamos Último de la línea 2 a 1, y el antiguo de la línea 1 se pone a 0. El valor de r4 se toma del banco de regs, y como el valor de r3 no está disponible aún, la instr mul se quedará en la ventana de instr.
4. Cuando el campo OK de la ventana de regs de la operación mul pase a ser 1, podemos continuar con la operación. Esto pasa al final del ciclo 3.
5. Al final del ciclo 3 se han descod las instr 3 y 4 - La instr 3 sw no dará lugar a renom porque escribe directamente en memoria, y como r5 no se había usado antes se toma desde el banco de registros. Pero r3 está ahora mismo en la línea 2 del buffer de renom, y no está marcado como válido. Hasta que no termine la multiplicación iniciada antes, no tenemos ese valor.
6. La instr 4 lw accede a r6 desde banco de registros, y da lugar a renom de r5. Se añade al buffer. Si la emisión es desordenada, se podría emitir la instr.

Modelo de consistencia, ROB

Quedaría por hablar de la forma en la que, cuando una inst termina su ejecución, actualiza con sus resultados el banco de registros. Existen dos alternativas para el **modelo de consistencia del procesador**:

- Modelo con finalización desordenada: al igual que las instr terminan de ejecutarse de forma desordenada (porque unas tienen más ciclos que otras, independientemente de su orden en el programa), su resultado se escribe en el banco de regs desordenadamente.
- Modelo con finalización ordenada: actualiza con los resultados el banco de regs en el orden en el que las instr aparecen en el programa, independientemente de cuándo se generen.

- Este modelo es más común en procesadores modernos, porque se ha demostrado que la mejor forma de aprovechar el paralelismo es emisión desordenada (según estén disponibles las instr) y almacenado de resultados en orden.

El modelo ordenado se puede implementar gracias al **buffer de reordenamiento ROB**, una estructura relativamente sencilla y versátil, pero un poco exigente con los recursos hardware. El ROB se puede usar, además de para finalización ordenada, como buffer de renom.

El ROB es una cola circular (la línea 1 va después de la 10, y la 10 va antes de la 1) con dos punteros marcados con **: el de la línea 9 es donde se escribe la siguiente instr que le llegue desde decod, y el de la línea 3 es a la que le corresponde escribir su resultado en el banco de registros.

Las instr se escriben ordenadamente en el ROB desde ID o ID/ISS y también se retiran ordenadamente tras escribir sus resultados en los registros. Se podrían introducir tantas IPC como instr se decodifiquen en el susodicho ciclo. El número de registros que se pueden escribir cada vez es un parámetro dado por el procesador, y nos da una idea de su velocidad pico:

El número de instr que pueden retirarse por ciclo * freq reloj = num max de instr que se pueden procesar por ud. de tiempo

Si se llena el ROB, se detienen las decodificaciones hasta que queden líneas libres.

El significado de los campos es:

- CodOP: código de la operación
- RegDest: registro del banco de regs donde vamos a escribir el resultado
- Valor: campo donde se almacena temporalmente el resultado, hasta que lo podamos escribir y se retire
- OK: indica si el valor almacenado en Valor es válido
- Unidad: unidad funcional necesaria
- Pred: indica si se introdujo la línea al ROB como resultado de una predicción
- Flush: indica si el resultado se ha retirado del ROB sin almacenar el resultado en el registro
- Estado: f-> finalizado, x-> ejecutándose, i-> estación de reserva y esperando para su ejecución

Para que una instr pueda retirarse, tanto la propia instr como las de las líneas previas a ésta deben tener el campo OK a 1 (y estado F); o el campo flush a 1.

Ejemplo - ROB

No vamos a tener en cuenta los campos pred y flush porque pertenecen más bien al ámbito especulativo y eso lo vamos a ver más adelante.

#	Cod Op	Reg destino	Valor	OK	Unidad	Pred	Flush	Estado
1								
2								
3**	add	r3	351	1		0	0	f
4	mult	r3	--	0	mult	0	0	x
-	-	[no hay, va		-		-	-	:

5 #	SW Cod Op lw	Reg [mem] destino r5	-- --	0 OK	Unidad carga(load)	0 Pred	0 Flush	I Estado x
6				0		0	0	
7	add	r6	--	0		0	0	i
8	add	r5	--	0		0	0	i
9**								
10								

Este es el buffer ROB tras el ciclo 4 de la traza de emisión desordenada, la añado aquí abajo otra vez para más facilidad:

instr	1	2	3	4	5	6	7	8	9	10	...
1. add r3, r1, r2	IF	ID	EX								
2. mul r3, r3, r4	IF	ID		EX	EX	EX					
3. sw (r5), r3	IF		ID				EX				
4. lw r5, (r6)		IF	ID	EX	EX						
5. add r6, r5, r4		IF		ID		EX					
6. add r5, r3, r4		IF		ID			EX				

Al finalizar el ciclo 4, la instrucción add (línea 3 del ROB) escribirá su resultado en Valor, y podrá retirarse al siguiente ciclo.

La instr mul ha comenzado a ejecutarse y está en la línea 4 del ROB. La inst sw está en la línea 5 del ROB y está esperando en la ventana de inst/estación de reserva al resultado de la multiplicación.

La instr de lectura lw como sabemos puede adelantar a la de almacenamiento sw, y se encuentra en la línea 6 del ROB.

Las dos sumas finales están esperando en la ventana a recibir info sobre sus operandos, y por tanto en el ROB aún no tienen marcada la unidad que van a utilizar cuando se ejecuten.

#	Cod op	Reg dest	Oper 1	OK1	Oper2	OK2
1	sw	no hay destino pq va a memoria	[r5]	1	4	0
3	add	7	6	0	[r4]	1
4	add	8	4	0	[r4]	1
...	...					

Ventana de instr al final del ciclo 4, el mismo momento en el que hemos visto el ROB un par de tablas más arriba: en reg dest de las operaciones de suma **se indican las líneas que les corresponden en el ROB**. En los operandos cuyo OK es 0, se indica **la línea del ROB que corresponde a las instrucciones que producirán esos resultados**.

instr	1	2	3	4	5	6	7	8	9	10	...
1. add r3, r1, r2	IF	ID	EX	ROB	WB						
2. mul r3, r3, r4	IF	ID		EX	EX	EX	ROB	WB			
3. sw (r5), r3	IF		ID				EX	WB			
4. lw r5, (r6)		IF	ID	EX	EX	ROB		WB			
5. add r6, r5, r4		IF		ID		EX	ROB		WB		
6. add r5, r3, r4		IF		ID			EX	ROB	WB		

Traza de emisión desordenada, con las etapas de ROB (escritura en ROB en su correspondiente línea del resultado generado por la ejecución) y WB (retirada del resultado almacenado en ROB. Valor para escribirlo en el banco de registros).

Sw no necesita pasar por ROB porque escribe en memoria directamente. Se ha supuesto que se pueden escribir dos resultados en ROB por ciclo, y que se pueden retirar 2 resultados al banco de regs por ciclo. No cuentan las instr de almacenamiento y por eso en el ciclo 8 hay tres WB, porque una es de sw y va a memoria directo sin ayuda del ROB.

Podemos observar que la finalización es ordenada: las instrucciones se retiran WB en 5, 8, 8, 8, 9, 9; aunque hayan terminado de ejecutarse desordenadamente de la forma 3, 6, 7, 4, 5, 7

2.2. Procesamiento de las instrucciones de salto

Los procesadores superescalares son segmentados. Los saltos pueden ser muy dañinos a la ejecución, como cambian el orden de la secuencia de instrucciones puede ser que hayamos malgastado ciclos haciendo instr que no tocaban y encima tenemos que deshacer los efectos de esas instrucciones sobre el estado del procesador.

Además, el hecho de procesar múltiples instr por ciclo sólo hace que aumente este riesgo: más posibilidades de procesar un salto en un ciclo y, por tanto, más instrucciones que potencialmente tenemos que deshacer.

Predicción de saltos, BTB

Es muy fácil lidiar con saltos incondicionales, porque en la etapa de precaptación son marcados y en la etapa de captación IF se empiezan a procesar (calcular la dirección de destino del salto). El problema son los saltos condicionales: aunque se podría hacer lo mismo, hasta que no conozcamos el valor de la condición no sabremos si se va a producir el salto o no. Para

solucionarlo se tiende a la **predicción de saltos**, determinar la alternativa (saltar o no) más probable y continuar el procesamiento con la secuencia de instrucciones más probable. Cuando lleguemos a la condición sabremos si nuestra predicción es correcta o no: si no acertamos, el rendimiento se verá afectado porque tendremos que volver atrás además de deshacer las instrucciones previas. Es decir: la eficacia de este procedimiento dependerá de nuestra precisión a la hora de hacer predicciones.

- Existen procedimientos estáticos basados en la propia instrucción de salto. Se tiene en cuenta dirección del salto (palante o atrás), la condición ($<$, $>$, $\leq\dots$), si permite bit de predicción... Estas predicciones no tienen en cuenta la dinámica real de ejecución del código
- Los procedimientos dinámicos tienen en cuenta el historial de comportamiento de cada instrucción de salto condicional para determinar el comportamiento más probable. Para ello se emplea una estructura llamada **BTB (Branch Target Buffer)** en la que se introduce información sobre las instrucciones de salto de un programa que se haya ejecutado con anterioridad.

De esta forma, con la dirección de la instrucción de salto se almacena además una serie de bits que codifica la historia de ejecución del salto.

Cuando una instrucción de salto llega al cauce, se comprueba si tiene entrada en BTB. Si no la tiene, se hace una predicción estática y se le asigna una entrada, donde apuntaremos en el campo de bits si la predicción ha acertado o no. Si sí tiene entrada, se realiza la predicción según lo que indique la mayoría de bits y se actualiza con el nuevo resultado. Por ejemplo, una entrada que almacena los 3 últimos intentos que sea 011 quiere decir que ha fallado una vez y acertado dos, así que se hará el salto. Luego se actualizan los bits de la forma X01, donde X es si el salto ha sido correcto o no. Hay ríos de tinta en este tema sobre cuánta memoria conviene gastar en la historia.

La recuperación tras una predicción incorrecta implica volver a la primera instrucción de la secuencia alternativa de instrucción y deshacer las instrucciones ejecutadas. La dirección de la instrucción se puede determinar sin mayor problema si, previa predicción...

- ...en el caso de predecir saltar, guardamos la instrucción siguiente al salto
- ...en el caso de predecir NO saltar, calculamos la dirección de destino del salto

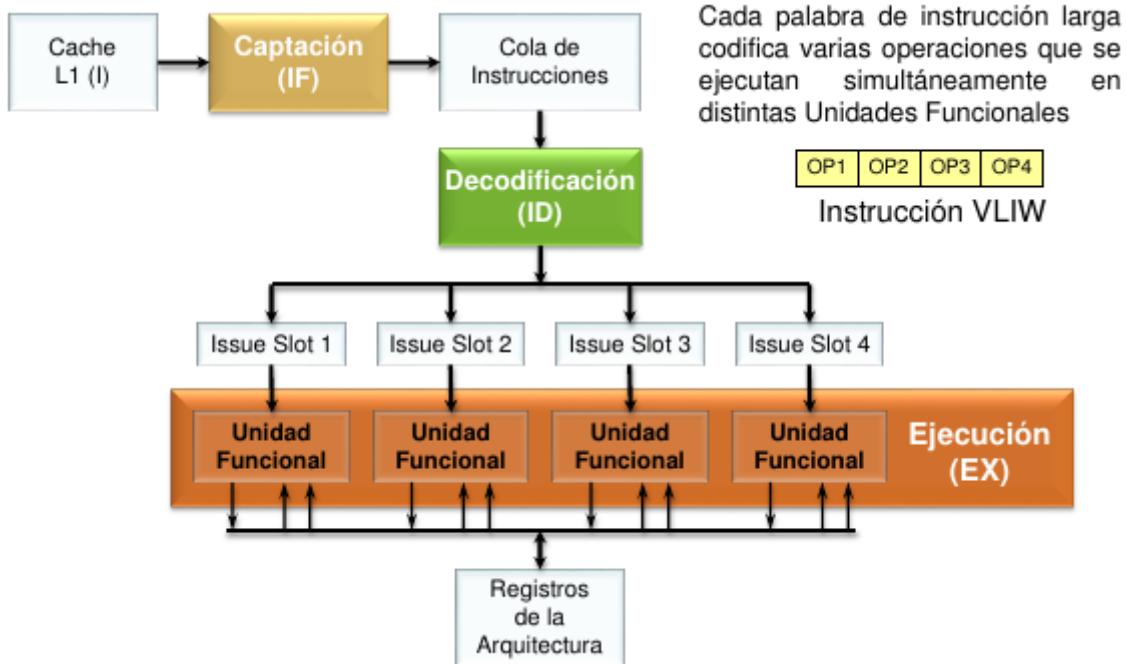
Respecto a deshacer las instrucciones ejecutadas, con el ROB se incluye el campo pred que determina las instrucciones introducidas en el ROB especulativamente, así como el campo flush que permite retirar sin ninguna modificación en el procesador las instrucciones introducidas marcadas con pred.

3. Procesadores VLIW

Un VLIW es un procesador segmentado que puede ejecutar más de una instrucción por ciclo mediante agrupaciones de instrucciones que se pueden ejecutar paralelamente generadas por el compilador. Los VLIW pueden ejecutar operaciones empaquetadas gracias al trabajo del compilador, que se ejecutan sin mayores comprobaciones.

Mientras que **los procesadores superescalares planifican dinámicamente, los VLIW lo hacen estáticamente**. Esto contribuye a que la microarquitectura de un VLIW sea **más simple** que la de un superescalar, porque no necesitan estructuras complejas como los buffers o estaciones de reserva. Es por ello que los VLIW se plantearon como una alternativa a los superescalares que consiguiera más IPC con menor consumo y mejor escalabilidad, empleando el aumento de transistores en más unidades funcionales para trabajar en paralelo -- Sin embargo, **para aprovechar realmente el VLIW se necesita encontrar un número suficientemente elevado de operaciones** paralelizables, además de aplicar ciertas estrategias que requieren de ciertos recursos hardware. Estas estrategias son necesarias para compensar el hecho de contar con menos información en el momento de la compilación sobre la ejecución.

También son menos portables, porque requieren de información específica sobre el cauce para el que está generando código (número de uds funcionales, latencia...).



Esquema de cauce de un VLIW:

- La captación IF toma una o varias instr VLIW y las pasa a la cola de instr. Una instr VLIW encompasa varias instr así que aunque "solo sea una" se están ejecutando varias inst por ciclo.
- Las instr VLIW se toman desde la cola y se pasan a la ud de decod ID, donde se determina las operaciones que hay que mandar a cada uno de los slots de emisión. Cada uno de los slots debe poder ejecutarse en alguna unidad.

Por lo tanto, para aprovechar un VLIW es necesario que el compilador encuentre tantas operaciones independientes como slots de emisión haya, y que cada uno de esos slots reciba operaciones que se puedan ejecutar en las unidades a las que puede acceder el slot. Si no se encuentra una operación independiente, se inserta una operación tipo nop (no operar), dando lugar a códigos poco densos que si no tienen alguna estrategia adecuada requerirán más espacio que un programa equivalente en un superescalar.

3.1. Técnicas software para ampliar los bloques básicos

Los riesgos de control asociados a los saltos condicionales dificultan la planificación del código al compilador porque, en tiempo de compilación, no se saben los comportamientos del salto en ejecución. Así que es difícil encontrar operaciones independientes, y cuantos menos saltos haya, mejor.

Se denomina bloque básico al cjto de operaciones situadas entre dos saltos, el compilador debe encontrar la mejor planificación para las operaciones de cada uno de los bloques básicos del programa. Ahora veremos unas técnicas para aumentar el número de instr indep en los bloques básicos: desenrollado de bucles y segmentación software. Para explicarlas usaremos el siguiente código:

```

inicio:    lw r1,n;           //carga en r1 el num de iteraciones
           ld f0,a;           //carga en f0 el valor de a
           add r3,r0,r0;        //hace r3 = 0 (índice del vector)

bucle:     ld f2,x(r3);      //f2 se carga con el elemento x[i]
           add f4,f2,f0;       //f4 = x[i] + s
           sd x(r3),f4;        //guardar en x[i] = f4
           subi r1,r1,#1;       //decrementar el número de elementos
           addi r3,r3,#8;        //apuntar al sig elemento del vector
           bnez r1,bucle;       //si quedan elementos, saltar de vuelta a bucle

```

Para construir las operaciones, el compilador debe:

1. Encontrar instr paralelizables
2. Cuántos slots hay
3. Qué tipo de operaciones puede manejar cada slot
4. Retardo de cada operación

Vamos a suponer un procesador con 3 slots, uno a una ALU de operaciones aritmético-lógicas con enteros, el siguiente a otra ALU pero para float, y uno para accesos a memoria.

Etiqueta	slot1: op mem	slot2: op ALU	slot3: op FP
inicio	ld f0,a	add r3,r0,r0	<i>nop</i>
	lw r1,n	<i>nop</i>	<i>nop</i>
bucle	ld f2,x(r3)	<i>nop</i>	<i>nop</i>
	<i>nop</i>	subi r1,r1,#1	add f4,f2,f0
	<i>nop</i>	bnez r1,bucle	<i>nop</i>
	sd x(r3),f4	addi r3,r3,#8	<i>nop</i>

HASTA AQUÍ HEMOS LLEGADO, SI PREGUNTAN MUCHO POR VLIW CAGUÉ 04:27 09-06-2022

3.2. Instrucciones con predicado

3.3. Procesamiento especulativo