

t4-AC.pdf



patrivc



Arquitectura de Computadores



2º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada



Descarga la APP de Wuolah.
Ya disponible para el móvil y la tablet.



Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



Tema 4. Arquitecturas con Paralelismo a nivel de Instrucción (ILP)

Lección 11. Microarquitecturas ILP. Cauces Superescalares

Mejora de las Prestaciones de los Procesadores

La velocidad de un procesador está relacionada con las instrucciones por ciclo y la frecuencia.
 $V_{cpu} = IPC \times F$

La mejora en la tecnología de fabricado de CI basada en el Silicio ha permitido que los circuitos integrados sean más grandes (se aumenta el tamaño) y la reducción del tamaño de los transistores.

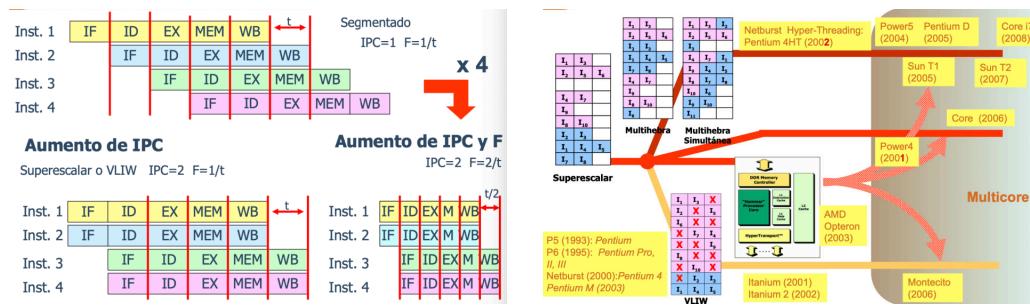
Se pueden incluir en un circuito integrado más transistores que se usan para diseños de microarquitecturas más complejas en un solo CI: Paralelismo entre instrucciones (Procesadores Superescalares).

Se reduce la longitud de puerta del transistor y con ello el tiempo de conmutación -> Mayores frecuencias de funcionamiento.

Los procesadores con paralelismo entre instrucciones más sencillos serían los **procesadores segmentados**. Una serie de etapas que funcionan independientemente y cada etapa trabaja con una instrucción, que se pueden ejecutar en paralelo. Entonces la instrucción por ciclo sería $IPC=1$ y la frecuencia $F=1/t$.

La mejora en la tecnología que permite aumentar el número de IPC se puede conseguir volviendo las etapas más complejas. Las etapas pueden trabajar con más instrucciones, esto son los **procesadores superescalares o VLIW**. Pueden captar dos instrucciones, ejecutar dos instrucciones, descodificar dos instrucciones... etc. El $IPC=2$ y la $F=1/t$

Otro ejemplo de mejora sería el aumento de IPC y F. Cada tiempo de la etapa sería la mitad. Podemos hacer que el tiempo de cada etapa sea la mitad, yendo así a una frecuencia del doble, mejorando la velocidad. Pasamos a un procesador con $IPC=2$ y $F=2/t$.



Los procesadores VLIW utilizan menos hardware. Otras alternativas de mejora son los procesadores multihebra y los multihrea simultánea.

Clasificación de cores multithread

-Temporal Multithreading (TMT): Ejecutan varios threads concurrentemente en el mismo core. La conmutación entre threads la decide y controla el hardware. Emite instrucciones de un único thread en un ciclo.



-Simultaneous MultiThreading (SMT) o multihilo simultáneo o horizontal multithread:
Ejecutan, en un core superescalar, varios threads en paralelo. Pueden emitir (para su ejecución) instrucciones de varios threads en un ciclo.

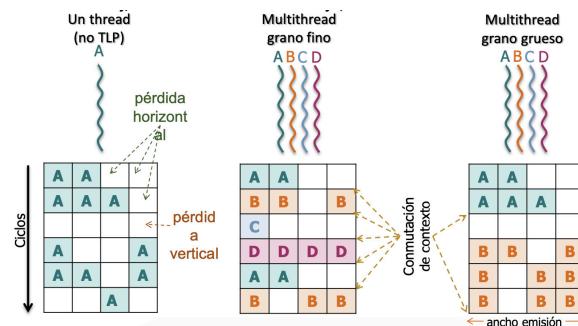
Clasificación de cores con TMT

-Fine-grain multithreading (FGMT) o interleaved multithreading (multihebra de grano fino):
La conmutación entre threads la decide el hardware cada ciclo (coste 0). Son procesadores de tipo temporal en la que cada ciclo pueden pasar de una hebra a otra. A la larga son los multihebra de grano grueso. Funciona por turno rotatorio (round-robin) o por eventos de cierta latencia combinado con alguna técnica de planificación (ej. thread menos recientemente ejecutado). Los eventos pueden ser: dependencia funcional, acceso a datos a cache L1, salto no predecible, una operación de cierta latencia (ej. div), ...

-Coarse-grain multithreading (CGMT) o blocked multithreading (multihebra de grano grueso):
La conmutación entre threads la decide el hardware (coste de 0 a varios ciclos) tras intervalos de tiempo prefijados (timeslice multithreading) o por eventos de cierta latencia (switch-on-event multithreading).

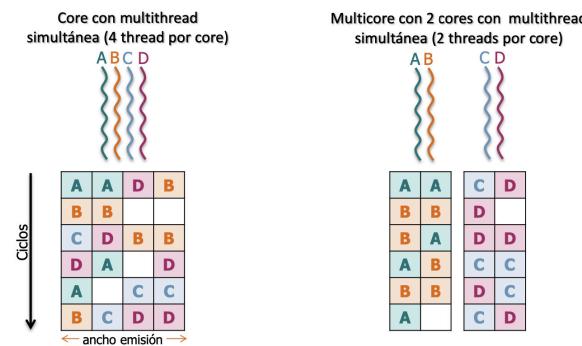
Alternativas en un core con emisión múltiple de instrucciones de un thread

Tenemos una única hebra en la cual se emiten instrucciones de esa hebra, entonces tenemos una perdida horizontal. En otros casos puede ocurrir que no se pueda ejecutar ninguna instrucción hasta que no se realicen las anteriores, entonces tenemos una perdida vertical. En un core superescalar o VLIW se emiten más de una instrucción cada ciclo de reloj; en las alternativas de abajo, de un único thread



Core multithread simultánea y multicores

Tenemos un núcleo multihebra que trabaja con 4 hebras simultáneamente, entonces, son dos núcleos y cada uno de ellos es multihebra simultánea. En un multicore y en un core superescalar con SMT (Simultaneous MultiThread) se pueden emitir instrucciones de distintos threads cada ciclo de reloj.



Paralelismo entre instrucciones (ILP). Orden en Emisión y Finalización

Ordenaciones en una secuencia de instrucciones

En una secuencia de instrucciones se pueden distinguir tres tipos de ordenaciones:

- El **orden en que se captan las instrucciones** (el orden de las instrucciones en el código): orden en el que aparecen en el programa y se codifican.
 - El **orden en que las instrucciones se ejecutan**: las instrucciones que se codifican pasan a ejecutarse según un determinado orden.
 - El **orden en que las instrucciones cambian los registros y la memoria**: modifican el estado interno de la máquina.

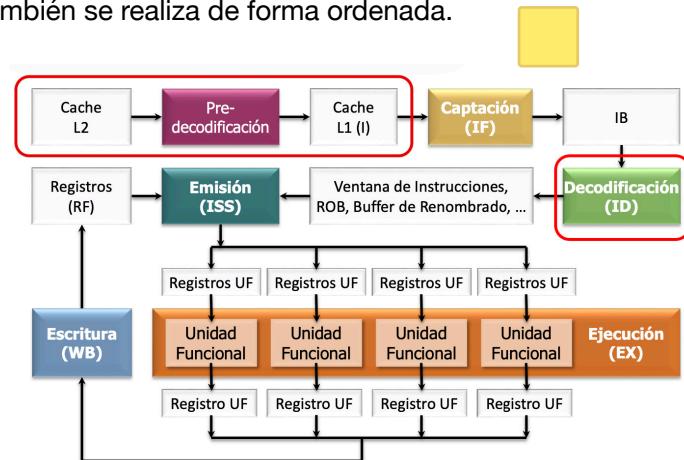
El procesador superescalar debe ser capaz de identificar el paralelismo entre instrucciones (ILP) que exista en el programa y organizar la captación, decodificación y ejecución de instrucciones en paralelo, utilizando eficazmente los recursos existentes (el paralelismo de la máquina).

Cuanto más sofisticado sea un procesador superescalar, menos tiene que ajustarse a la ordenación de las instrucciones según se captan, para la ejecución y modificación de los registros, de cara a mejorar los tiempos de ejecución. *La única restricción es que el resultado del programa sea correcto.*

Cauces superescalares

Etapas de un Procesador Superescalar: Predecodificación I

Tenemos las distintas etapas del cauce y en blanco los distintos elementos de almacenamiento. Hay una etapa de **pre-decodificación**. A estas instrucciones se le añade código adicional, que permite que luego cuando estemos en otra etapa diferencie mas rápidamente a las instrucciones (por ejemplo si es una instrucción de salto). La captación se realiza según el orden del programa y la descodificación también se realiza de forma ordenada.



Etapas de un Procesador Superescalar: Emisión Paralela de Instrucciones

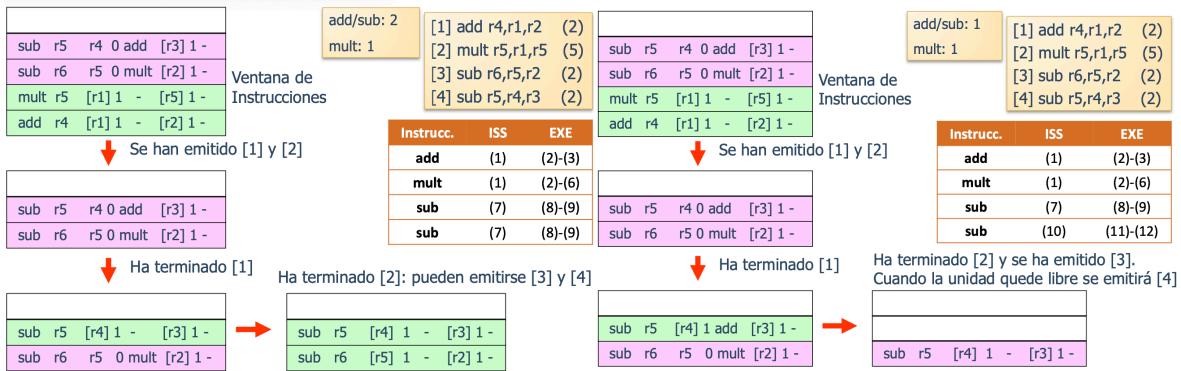
Una vez que las instrucciones de descodifican pasan a la ventana de instrucciones. La ventana de instrucciones almacena las instrucciones pendientes (todas, si la ventana es centralizada, o las de un tipo determinado, si es distribuida).

Las **instrucciones se cargan en la ventana una vez decodificadas**. Se utiliza un **bit** para indicar si **un operando está disponible** (se almacena el valor o se indica el registro desde donde se lee) o no (se almacena la unidad funcional desde donde llegará el operando).

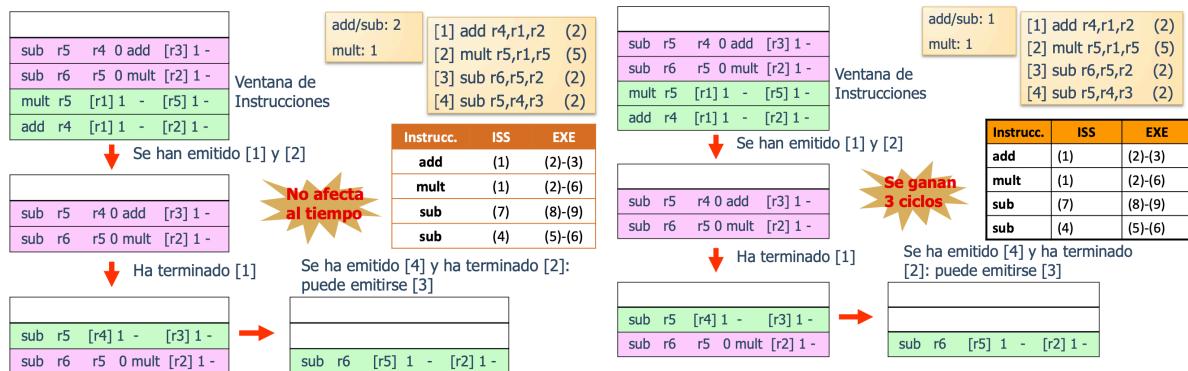
Una **instrucción puede ser emitida cuando tiene todos sus operandos disponibles y la unidad funcional donde se procesará**. Hay diversas posibilidades para el caso en el que varias instrucciones estén disponibles (características de los buses, etc.).



Emisión Paralela de Instrucciones: Ordenada



Emisión Paralela de Instrucciones: Desordenada



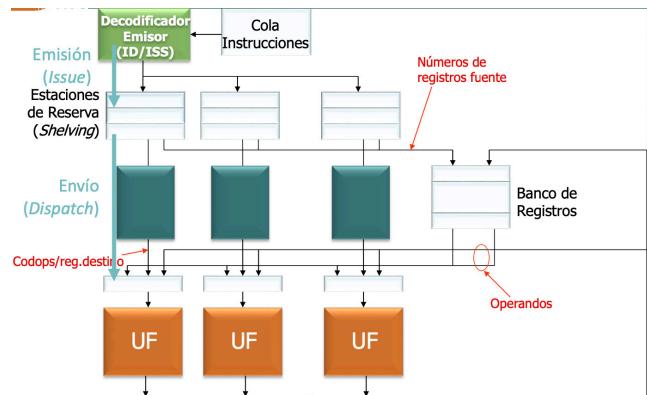
Estaciones de Reserva I (Ventana de Instrucciones Distribuida)

Estación de reserva: es un **rediseño de la ventana de instrucciones**. Solo se meten las instrucciones que vaya a usar la unidad funcional y espera a que estén disponibles los operandos para realizar las instrucciones.

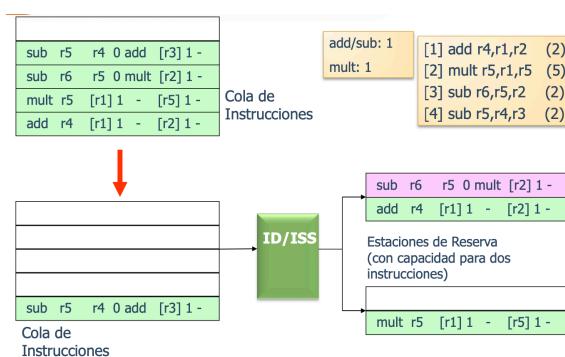
Cuando la estación de reserva está disponible (tiene espacio) se realiza el envío.

Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



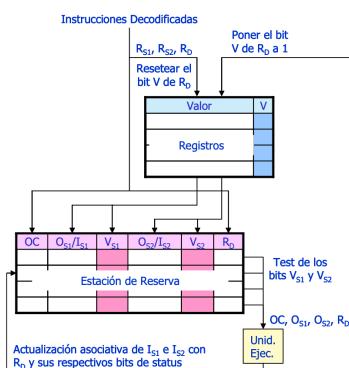
Ejemplo de Emisión con Estaciones de Reserva



Comprobación de los Operandos

En cada uno de los **registros** se añade un **bit de validez**: 1 si esta libre 0 si no. (Captación de los operandos en la emisión):

- **OC**: Código de operación
- **RS1, RS2**: Registros fuente
- **RD**: Registro de destino
- **OS1, OS2**: Operandos fuente
- **IS1, IS2**: Identificadores de los operandos fuente
- **VS1, VS2**: Bits válidos de los operandos fuente



Ejemplo de uso de Estaciones de Reserva I

En el ciclo i emitimos la instrucción de multiplicación de r1, r2 que se guarda en r3. En el ciclo i+1 emitimos dos instrucciones: add r5,r2,r3 y add r6,r3,r4.

Entonces, en el ciclo i se emite la instrucción de multiplicación, ya decodificada, a la estación de reserva, se anula el valor de r3 en el banco de registros y se copian los valores de r1 y r2 (disponibles) en la estación de reserva.

En el ciclo i+1 la operación de multiplicación tiene sus operadores preparados (VS1 =1 y VS2 =1) así que puede enviarse a la unidad de ejecución.

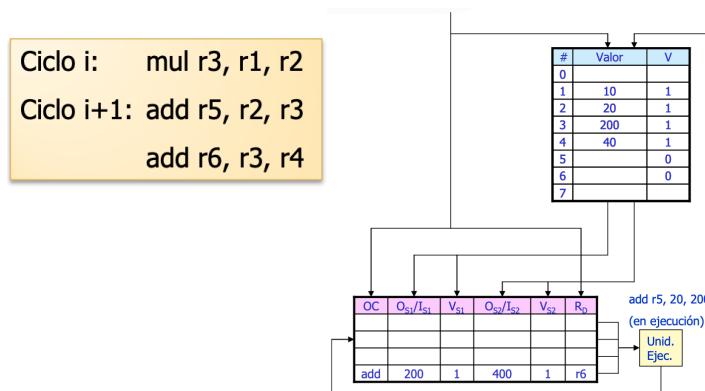


En el ciclo $i + 1$ (cont.): se emiten las dos instrucciones de suma a la estación de reserva, se anulan los valores de $r5$ y $r6$ en el banco de registros y se copian los valores de los operandos disponibles y los identificadores de los operandos no preparados.

Ciclos $i + 2 \dots i + 5$: la multiplicación sigue ejecutándose. No se puede ejecutar ninguna suma hasta que esté disponible el resultado de la multiplicación ($r3$).

Ciclo $i + 6$: se escribe el resultado de la multiplicación en el banco de registros y en las entradas de la estación de reserva. Se actualizan los bits de disponibilidad de $r3$ en el banco de registros y en la estación de reserva.

Ciclo $i + 6$ (cont.): las sumas tienen sus operadores preparados ($VS1 = 1$ y $VS2 = 1$), así que pueden enviarse a la unidad de ejecución. Como sólo hay una unidad de ejecución, se envía la instrucción más antigua de la estación de reserva, la primera suma.

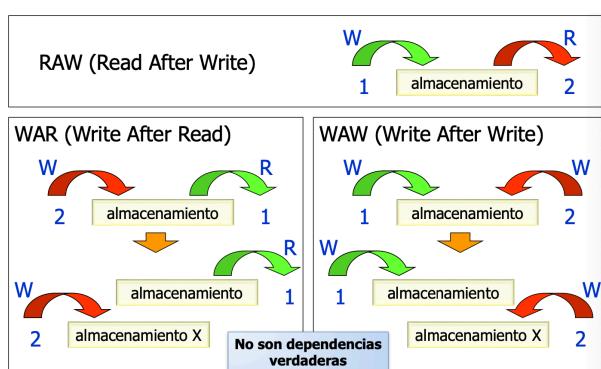


Renombramiento de registros

Riesgos de Datos

Sabemos que lo que determina que unas instrucciones se puedan ejecutar o no, es la dependencia de datos entre instrucciones. Los riesgos de datos que tenemos son: RAW (Read After Write), WAR (Write After Read) y WAW (Write After Write).

Las dependencias que son realmente esenciales son las de tipo RAW.



Renombramiento de Registros

Técnica para evitar el efecto de las dependencias WAR, o Antidependencias (en la emisión desordenada) y WAW, o Dependencias de Salida (en la ejecución desordenada)



Implementación estática: durante la compilación

Implementación dinámica: durante la ejecución (circuitería adicional y registros extra).

Características de los Buffers de Renombrado:

- Tipos de Buffers (separados o mezclados con los registros de la arquitectura)
- Número de Buffers de Renombrado
- Mecanismos para acceder a los Buffers (asociativos o indexados)

Velocidad del Renombrado

- Máximo número de nombres asignados por ciclo que admite el procesador

Buffers de Renombramiento

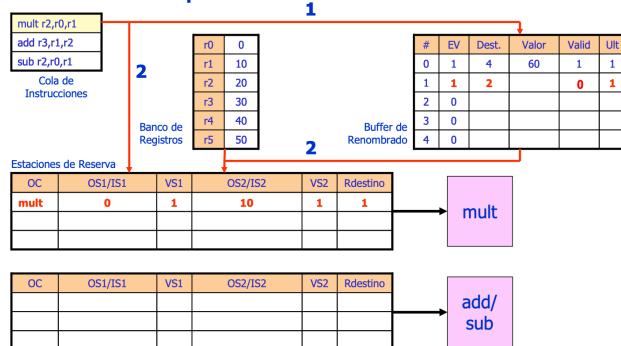
Entrada Válida	Registro Destino	Valor	Valor Válido	Último
1	5	50	1	1
1	12	1200	1	1
1	2	20	1	1
1	1	3	1	1
⋮	⋮	⋮	⋮	⋮

Permite varias escrituras pendientes a un mismo registro

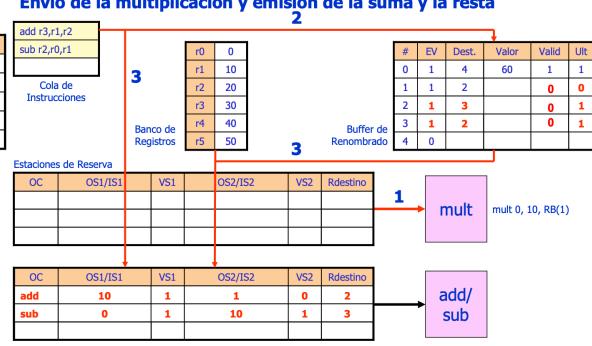
El bit de último se utiliza para marcar cual ha sido la más reciente.

Algoritmo de Tomasulo

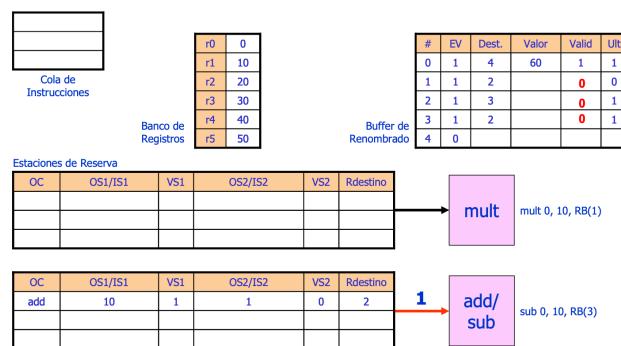
Emisión de la multiplicación



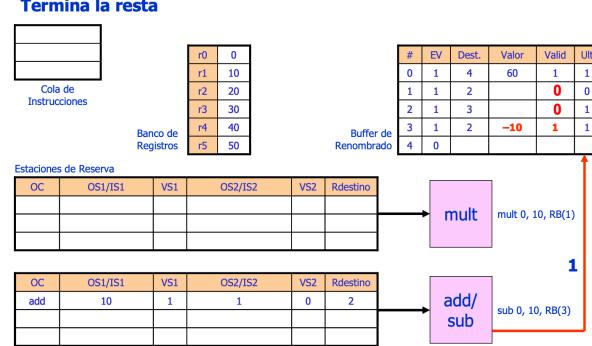
Envío de la multiplicación y emisión de la suma y la resta

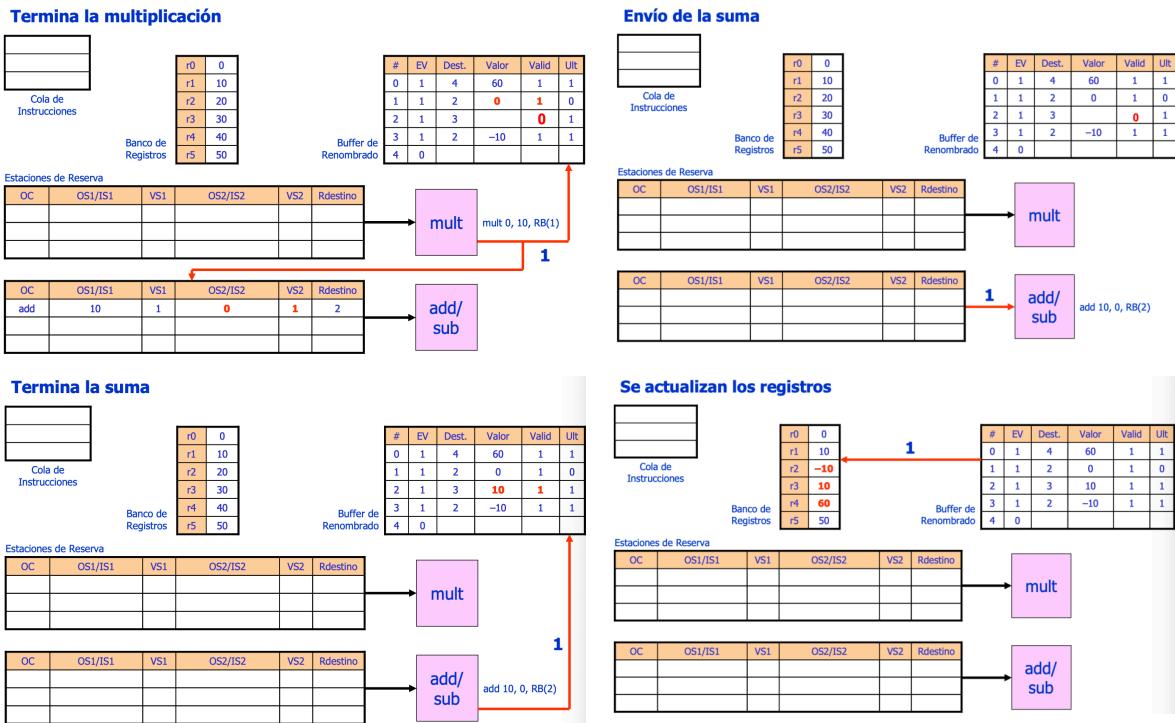


Envío de la resta



Termina la resta





Lección 12. Consistencia del procesador y procesamiento de saltos

Consistencia. Reordenamiento.

En el **Procesamiento de una Instrucción** se puede distinguir entre:

- El **Final de la Ejecución de la Operación** codificado en las instrucciones. (Se dispone de los resultados generados por las UF pero no se han modificado los registros de la arquitectura).
- El **Final del Procesamiento de la Instrucción** o momento en que se Completa la Instrucción (Complete o Commit). (Se escriben los resultados de la Operación en los Registros de la Arquitectura. Si se utiliza un Buffer de Reorden, ROB, se utiliza el término Retirar la Instrucción, Retire, en lugar de Completar)

La **Consistencia de un Programa** se refiere a:

- El orden en que las instrucciones se completan
- El orden en que se accede a memoria para leer (LOAD) o escribir (STORE)

Cuando se ejecutan instrucciones en paralelo, el orden en que termina (finish) esa ejecución puede variar según el orden que las correspondientes instrucciones tenían en el programa pero **debe existir consistencia entre el orden en que se completan las instrucciones y el orden secuencial que tienen en el código de programa.**

Podemos distinguir entre la **consistencia del procesador** y la **consistencia de memoria**.

En caso de la **consistencia de memoria** también tenemos una **consistencia débil**, en la que los **accesos a memoria** se realizan desordenadamente siempre que no afecten a las dependencias y es interesante porque en los accesos a memoria tenemos instrucciones de tipo LOAD (lectura) y tipo STORE (escritura), con load hay que calcular la dirección que se accede a memoria , intentamos ver si el dato esta en cache y el dato que leería lo escribiría en un registro y en las instrucciones de tipo store hay que calcular la dirección que se accede a memoria y como escribe un dato en memoria entonces hay que esperar que ese dato este disponible.

La razón por la cual se permite consistencia a memoria débil: Deben detectarse y resolverse las dependencias de acceso a memoria.

Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



Consistencia de Procesador: Consistencia en el orden en que se completan las instrucciones. Encontramos:

-**Débil:** Las instrucciones se pueden completar desordenadamente siempre que no se vean afectadas las dependencias. Deben detectarse y resolverse las dependencias.

-**Fuerte:** Las instrucciones deben completarse estrictamente en el orden en que están en el programa. Se consigue mediante el uso de ROB

Consistencia de memoria: Consistencia del orden de los accesos a memoria. Encontramos:

-**Débil:** Los accesos a memoria (Load/Stores) pueden realizarse desordenadamente siempre que no afecten a las dependencias. Deben detectarse y resolverse las dependencias de acceso a memoria.

-**Fuerte:** Los accesos a memoria deben realizarse estrictamente en el orden en que están en el programa. Se consigue mediante el uso de ROB.

				Tendencia
Consistencia de Procesador Consistencia en el orden en que se completan las instrucciones	Débil: Las instrucciones se pueden completar desordenadamente siempre que no se vean afectadas las dependencias	Deben detectarse y resolverse las dependencias	Power1 (90) PowerPC 601 (93) Alpha R8000 (94) MC88110 (93)	↓
	Fuerte: Las instrucciones deben completarse estrictamente en el orden en que están en el programa	Se consigue mediante el uso de ROB	PowerPC 620 PentiumPro (95) UltraSparc (95) K5 (95) R10000 (96)	
Consistencia de Memoria Consistencia del orden de los accesos a memoria	Débil: Los accesos a memoria (Load/Stores) pueden realizarse desordenadamente siempre que no afecten a las dependencias	Deben detectarse y resolverse las dependencias de acceso a memoria	MC88110 (93) PowerPC 620 UltraSparc (95) R10000 (96)	↑
	Fuerte: Los accesos a memoria deben realizarse estrictamente en el orden en que están en el programa	Se consigue mediante el uso del ROB	PowerPC 601 (93) E/S 9000 (92)	

Tendencia / Prestaciones

Reordenamiento Load/Store I

Las instrucciones LOAD y STORE implican cambios en el Procesador y en Memoria:

-**LOAD:** cálculo de Dirección en ALU o Unidad de Direcciones, acceso a Cache y escritura del Dato en Registro

-**STORE:** cálculo de Dirección en ALU o Unidad de Direcciones y esperar que esté disponible el dato a almacenar (en ese momento acaba).

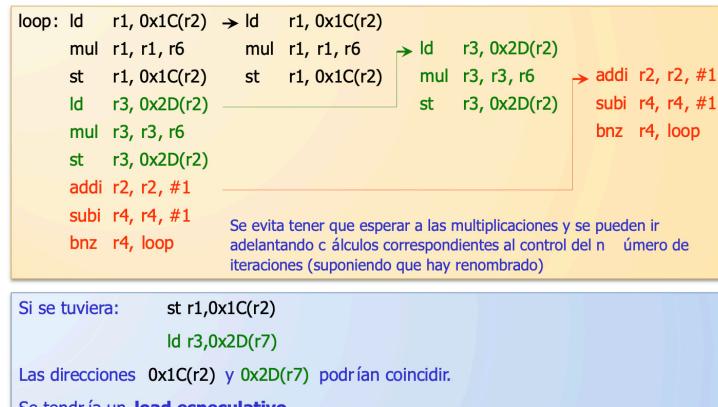
La Consistencia de Memoria Débil (reordenación de los accesos a memoria):

‘Bypass’ de Loads/Stores: Los Loads pueden adelantarse a los Stores pendientes y viceversa (siempre que no se violen dependencias)

Permite los Loads y Stores Especulativos: Cuando un Load se adelanta a un Store que le precede antes de que se haya determinado la dirección se habla de Load especulativo. Igual para un Store que se adelanta a un Load o a un Store.

Permite ocultar las Faltas de Cache: Si se adelanta un acceso a memoria a otro que dio lugar a una falta de cache y accede a Memoria Principal.



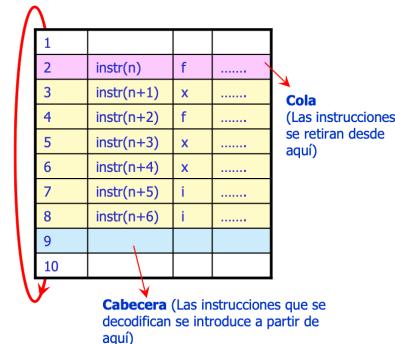


Hay una multiplicación que se va a almacenar en el registro, luego tenemos una instrucción load que se quiere almacenar en r3 y este dato es necesario para la multiplicación, entonces hay que esperar que terminen los cálculos, pero estos no están vinculados donde se quiere almacenar y escribir. Entonces no hace falta tener que esperar a la multiplicación y se pueden ir adelantando los cálculos correspondientes al control del numero de iteraciones (suponiendo que hay renombrado). Para que se produzca el reordenamiento hace falta que el procesador tenga consistencia de memoria débil.

Buffer de Reordenamiento (ROB)

Buffer de reordenamiento: El puntero de cabecera apunta a la siguiente posición libre y el puntero de cola a la siguiente instrucción a retirar. Las instrucciones se introducen en el ROB en orden de programa estricto y pueden estar marcadas como emitidas (issued, i), en ejecución (x), o finalizada su ejecución (f). Las instrucciones sólo se pueden retirar (se produce la finalización con la escritura en los registros de la arquitectura) si han finalizado, y todas las que les preceden también. La consistencia se mantiene porque sólo las instrucciones que se retiran del ROB se completan (escriben en los registros de la arquitectura) y se retiran en el orden estricto de programa.

La gestión de interrupciones y la ejecución especulativa se pueden implementar fácilmente mediante el ROB.



Ejemplo de uso del ROB

I1: mult r1, r2, r3
I2: st r1, 0x1ca
I3: add r1, r4, r3
I4: xor r1, r1, r3

Dependencias:
RAW: (I1,I2), (I3,I4)
WAR: (I2,I3), (I2,I4)
WAW: (I1,I3), (I1,I4), (I3,I4)

I1: Se puede empezar a ejecutar inmediatamente (se suponen disponibles r2 y r3)
I2: Se envía a la unidad de almacenamiento hasta que esté disponible r1
I3: Se puede empezar a ejecutar inmediatamente (se suponen disponibles r4 y r3)
I4: Se envía a la estación de reserva de la ALU para esperar a r1

Estación de Reserva (Unidad de Almacenamiento)

codop	dirección	op1	ok1
st	0x1ca	3	0

Estación de Reserva (ALU)

codop	dest	op1	ok1	op2	ok2
xor	6	5	0	[r3]	1

Ciclo 7							
#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	-	0	x
4	st	8	-	store	-	0	i
5	add	9	r1	int_add	-	0	x
6	xor	10	r1	int_alu	-	0	i

Ciclo 9 No se puede retirar add aunque haya finalizado su ejecución

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	-	0	x
4	st	8	-	store	-	0	i
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	-	0	x

Ciclo 10 Termina xor, pero todavía no se puede retirar

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	-	0	x
4	st	8	-	store	-	0	i
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	21	1	f

Ciclo 12

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	33	1	f
4	st	8	-	store	-	1	f
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	21	1	f

Ciclo 13

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	21	1	f

Se ha supuesto que se pueden retirar dos instrucciones por ciclo. Tras finalizar las instrucciones mult y st en el ciclo 12, se retirarán en el ciclo 13. Después, en el ciclo 14 se retirarán las instrucciones add y xor.

Procesamiento especulativo de saltos

Aspectos del Procesamiento de Saltos en un procesador Superescalar

Detección de la Instrucción de Salto: Cuanto antes se detecte que una instrucción es de salto menor será la posible penalización. Los saltos se detectan usualmente en la fase de decodificación e incluso en la captación (si hay predecodificación).

Gestión de los Saltos Condicionales no Resueltos: Si en el momento en que la instrucción de salto evalúa la condición de salto ésta no se haya disponible se dice que el salto o la condición no se ha resuelto. Para resolver este problema se suele utilizar el procesamiento especulativo del salto.

Acceso a las Instrucciones destino del Salto: Hay que determinar la forma de acceder a la secuencia a la que se produce el salto

El efecto de los saltos en los procesadores superescalares es más pernicioso ya que, al emitirse varias instrucciones por ciclo, prácticamente en cada ciclo puede haber una instrucción de salto.

Gestión de Saltos Condicionales no resueltos

Gestión de Saltos Condicionales no Resueltos (Una condición de salto no se puede comprobar si no se ha terminado de evaluar)	Bloqueo del Procesamiento del Salto	Se bloquea la instrucción de salto hasta que la condición esté disponible (68020, 68030, 80386)
	Procesamiento Especulativo de los Saltos	La ejecución prosigue por el camino más probable (se especula sobre las instrucciones que se ejecutarán). Si se ha errado en la predicción hay que recuperar el camino correcto. (Típica en los procesadores superescalares actuales)
	Múltiples Caminos	Se ejecutan los dos caminos posibles después de un salto hasta que la condición de salto se evalúa. En ese momento se cancela el camino incorrecto. (Máquinas VLIW experimentales: Trace/500 , URPR2)
Evitar saltos condicionales	Ejecución Vigilada (Guarded Exec.)	Se evitan los saltos condicionales incluyendo en la arquitectura instrucciones con operaciones condicionales (IBM VLIW, Cydra-5, Pentium, HP PA, Dec Alpha)

Esquemas de Predicción de Salto

Predicción Fija: Se toma siempre la misma decisión: el salto siempre se realiza, 'taken', o no, 'not taken'

Predicción Verdadera: La decisión de si se realiza o no se realiza el salto se toma mediante:

- Predicción Estática:** Según los **atributos** de la **instrucción de salto** (el código de operación, el desplazamiento, la decisión del compilador)
- Predicción Dinámica:** Según el **resultado** de **ejecuciones pasadas** de la instrucción (historia de la instrucción de salto)

Predicción Estática

Predicción basada en el Código de Operación: Para ciertos códigos de operación (ciertos saltos condicionales específicos) se predice que el salto se toma, y para otros que el salto no se toma. Ejemplo: MC88110 (93) PowerPC 603(93)

Predicción basada en el Desplazamiento del Salto: Si el desplazamiento es positivo (salto hacia delante) se predice que no se toma el salto y si el desplazamiento es negativo (salto hacia atrás) se predice que se toma. Ejemplo: Alpha 21064 (92) PowerPC 603 (93)

Predicción dirigida por el Compilador: El compilador es el que establece la predicción fijando, para cada instrucción, el valor de un bit específico que existe en la instrucción de salto (bit de predicción).

Ejemplo: Predicción Estática en el MC88110

Formato	Instrucción		Predicción
	Condición Especificada	Bit 21 de la Instr.	
bcnd (Branch Conditional)	$\neq 0$	1	Tomado
	$= 0$	0	No Tomado
	> 0	1	Tomado
	< 0	0	No Tomado
	≥ 0	1	Tomado
	≤ 0	0	No Tomado
	bb1 (Branch on Bit Set)		Tomado
	bb0 (Branch on Bit Clear)		No Tomado

Predicción Dinámica

La predicción para cada instrucción de salto puede cambiar cada vez que se va a ejecutar ésta según la historia previa de saltos tomados/no-tomados para dicha instrucción.

El presupuesto básico de la predicción dinámica es que **es más probable que el resultado de una instrucción de salto sea similar al que se tuvo en la última** (o en las n últimas ejecuciones)

Presenta **mejores prestaciones de predicción**, aunque su **implementación es más costosa**

Predicción Dinámica Implícita: No hay bits de historia propiamente dichos sino que se almacena la dirección de la instrucción que se ejecutó después de la instrucción de salto en cuestión.

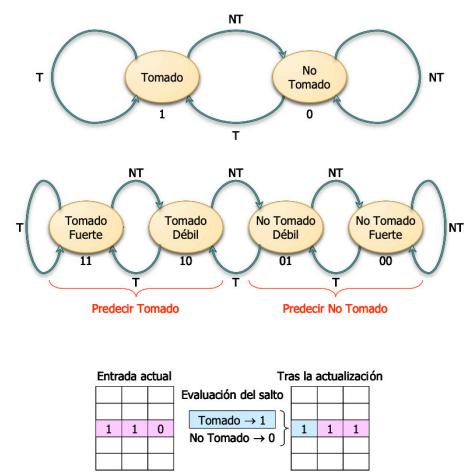
Predicción Dinámica Explícita: Para cada instrucción de salto existen unos bits específicos que codifican la información de historia de dicha instrucción de salto

Ejemplos de Procedimientos Explícitos de Predicción Dinámica de Saltos

Predicción con 1 bit de historia: La designación del estado, Tomado (1) o No Tomado T (0), indica lo que se predice, y las flechas indican las transiciones de estado según lo que se produce al ejecutarse la instrucción (T o NT)

Predicción con 2 bits de historia: Existen cuatro posibles estados. Dos para predecir Tomado y otros dos para No Tomado. La primera vez que se ejecuta un salto se inicializa el estado con predicción estática, por ejemplo 11. Las flechas indican las transiciones de estado según lo que se produce al ejecutarse la instrucción (T o NT)

Predicción con 3 bits de historia: Cada entrada guarda las últimas ejecuciones del salto. Se predice según el bit mayoritario (por ejemplo, si hay mayoría de unos en una entrada se predice salto). La actualización se realiza en modo FIFO, los bits se desplazan, introduciéndose un 0 o un 1 según el resultado final de la instrucción de salto



Instrucciones de Ejecución Condicional (Guarded Execution)

Se pretende reducir el número de instrucciones de salto incluyendo en el repertorio máquina instrucciones con operaciones condicionales ('conditional operate instructions' o 'guarded instructions').

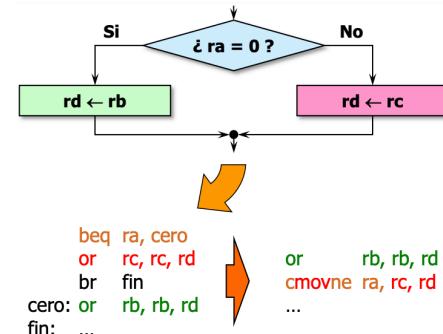
Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



Estas instrucciones tienen dos partes: la condición (denominada guardia) y la parte de operación. Ejemplo: cmovxx de Alpha.

cmovxx ra.rq, rb.rq, rc.wq
• xx es una condición
• ra.rq, rb.rq enteros de 64 bits en registros ra y rb
• rc.wq entero de 64 bits en rc para escritura
• El registro ra se comprueba en relación a la condición xx y si se verifica la condición rb se copia en rc.
Sparc V9, HP PA, y Pentium ofrecen también estas instrucciones.



Lección 13. Procesamiento VLIW

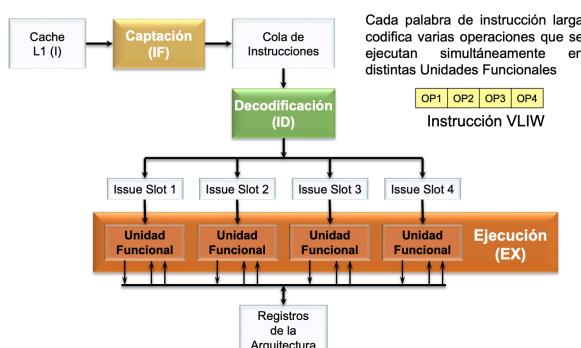
Características generales y motivación (ILP hardware vs. ILP software)

Las arquitecturas VLIW utilizan varias unidades funcionales independientes.

En lugar de tratar de enviar varias instrucciones independientes a las unidades funcionales, una arquitectura VLIW empaqueta varias operaciones en una única instrucción (muy larga, Very Long, por ejemplo, entre 112 y 128 bits) y ordena las instrucciones en el paquete de emisión con las mismas restricciones de independencia.

La decisión de qué instrucciones se deben emitir simultáneamente corresponde al compilador (hardware más sencillo que el de un superescalar).

Las ventajas de la aproximación VLIW crecen a medida que se pretende emitir más instrucciones por ciclo (el hardware adicional para un superescalar que emite dos instrucciones por ciclo es relativamente pequeño, pero crece a medida que se pretenden emitir más instrucciones por ciclo).



Planificación estática El papel del Compilador

Planificación estática
Necesita asistencia del compilador, que puede realizar renombrados, reorganizaciones de código, etc., para mejorar el uso de los recursos disponibles, el esquema de predicción de saltos,...

Planificación dinámica
Requiere menos asistencia del compilador pero más coste hardware. Facilita la portabilidad del código entre la misma familia de procesadores



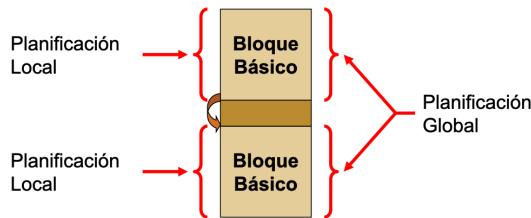
El compilador construye paquetes de instrucciones (ventanas de emisión) sin dependencias, de forma que el procesador no necesita comprobarlas explícitamente.

Existen dependencias RAW entre cada dos instrucciones consecutivas, y además existe una instrucción de salto que controla el final del bucle: Parece que no se puede aprovechar mucho ILP.

Planificación estática local y global

Planificación local: actúa sobre un bloque básico (mediante desenrollado de bucles y planificación de las instrucciones del cuerpo aumentado del bucle).

Planificación global: actúa considerando bloques de código entre instrucciones de salto.



Planificación Estática Local

Desenrollado de bucles: Al desenrollar un bucle se crean bloques básicos más largos, lo que facilita la planificación local de sus sentencias. Además de disponer de más sentencias, éstas suelen ser independientes, ya que operan sobre diferentes datos

Segmentación software (software pipelining): Se reorganizan los bucles de forma que cada iteración del código transformado contiene instrucciones tomadas de distintas iteraciones del bloque original. De esta forma se separan las instrucciones dependientes en el bucle original entre diferentes iteraciones del bucle nuevo.

Planificación Estática con Desenrollado de Bucles I

for (i = 1000 ; i > 0 ; i = i - 1)
x[i] = x[i] + s;

loop:
Id f0, 0(r1)
addd f4, f0, f2
sd f4, 0(r1)
subui r1, r1, #8
bne r1, loop

Existen dependencias RAW entre cada dos instrucciones consecutivas, y además existe una instrucción de salto que controla el final del bucle: Parece que no se puede aprovechar mucho ILP

loop:
Id f0, 0(r1)
Id f6, -8(r1)
Id f10, -16(r1)
Id f14, -24(r1)
Id f18, -32(r1)
addd f4, f0, f2
addd f8, f6, f2
addd f12, f10, f2
addd f16, f14, f2
addd f20, f18, f2
sd f4, 0(r1)
sd f8, -8(r1)
sd f12, -16(r1)
sd f16, -24(r1)
sd f20, -32(r1)
subui r1, r1, #40
bne r1, loop

El desenrollado pone de manifiesto un mayor paralelismo ILP

Instrucción Entera	Instrucción FP	Load/Store	Ciclo
loop		Id f0, 0(r1)	1
		Id f6, 0(r1)	2
	addd f4, f0, f2	Id f10, -16(r1)	3
	addd f8, f6, f2	Id f14, -24(r1)	4
	addd f12, f10, f2	Id f18, -32(r1)	5
	addd f16, f14, f2	sd f4, 0(r1)	6
	subui r1, r1, #40	addd f20, f18, f2	7
		sd f8, -8(r1)	
		sd f12, 24(r1)	8
		sd f16, 16(r1)	9
		sd f20, 8(r1)	10
		Slot 1	
		Slot 2	
		Slot 3	

Se tardarían $10 \times (1000/5) = 2000$ ciclos, frente a los $10 \times 1000 = 10000$ ó $6 \times 1000 = 6000$ ciclos del bucle sin desenrollar.

Planificación Estática con Segmentación Software I

for (i = 1000 ; i > 0 ; i = i - 1)
x[i] = x[i] + s;

loop:
Id f0, 0(r1)
addd f4, f0, f2
sd f4, 0(r1)
subui r1, r1, #8
bne r1, loop

Iter. i: Id f0, 0(r1)
Iter. i+1: addd f4, f0, f2
sd f4, 0(r1)
Iter. i+2: subui r1, r1, #8
bne r1, loop

loop:
sd f4, 16(r1)
addd f4, f0, f2
Id f0, 0(r1)
subui r1, r1, #8
bne r1, loop

Iter. i: Id f0, 16(r1)
addd f4, f0, f2
sd f4, 16(r1)

Iter. i+1: Id f0, 8(r1)
addd f4, f0, f2
sd f4, 8(r1)

Iter. i+2: Id f0, 0(r1)
addd f4, f0, f2
sd f4, 0(r1)

Planificación estática global

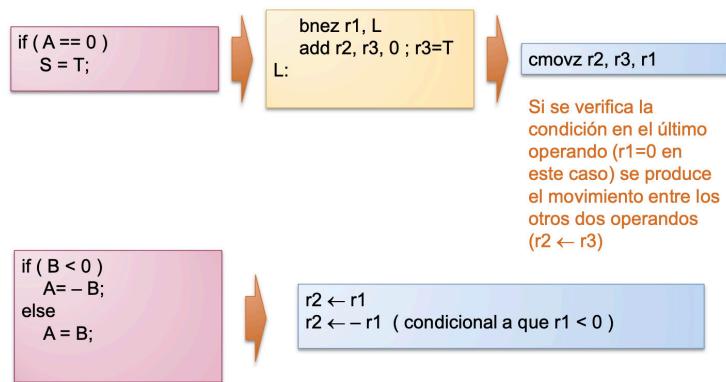
La planificación global mueve código a través de los saltos condicionales (que no correspondan al control del bucle)

Se parte de una estimación de las frecuencias de ejecución de las posibles alternativas tras una instrucción de salto condicional

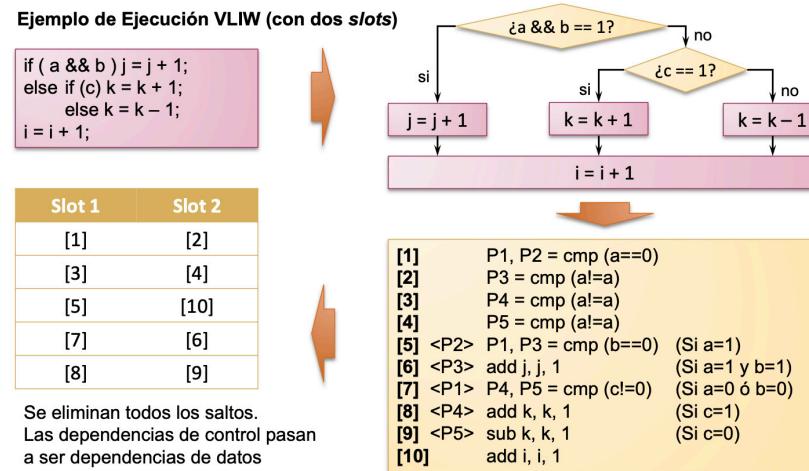
Apoyo para facilitar la planificación global:

- Instrucciones con predicado y
- Especulación

Instrucciones de Ejecución Condicional



Instrucciones con Predicado I



Procesamiento especulativo

El procesamiento especulativo se basa en la predicción de que determinada instrucción, condición, etc. será muy probable, para adelantar su procesamiento, mejorando las prestaciones del procesador.

El procesamiento especulativo tiene un coste si la predicción que se ha hecho no es correcta.

Este coste va desde el correspondiente a haber ejecutado una instrucción que no tendría que haberse ejecutado, hasta la necesidad de incluir código que deshaga el efecto de la operación implementada, vigilar el comportamiento frente a las excepciones, etc.

Slot 1	Slot 2
lw r1,40(r2)	add r3,r4,r5
	add r6,r3,r7
beqz r10,loop	
lw r8,0(r10)	
lw r9,0(r8)	

Slot 1	Slot 2
lw r1,40(r2)	add r3,r4,r5
lwc r8,0(r10),r10	add r6,r3,r7
beqz r10,loop	
lw r9,0(r8)	

Se produce el lw r8,0(r10) si r10!=0

Se aprovecha el slot de acceso a memoria. Si beqz da lugar al salto se ha ejecutado una instrucción de forma innecesaria. Otro problema son las excepciones.

Uso de Centinelas para Permitir la Especulación de las Referencias a Memoria

Cuando no existe ninguna ambigüedad, el compilador adelanta los LOADs con respecto a los STOREs para reducir la longitud del camino crítico en el código

Cuando existe ambigüedad:

- Se incluye en la arquitectura una instrucción para comprobar los conflictos de direcciones.
- La instrucción se sitúa en la posición original del LOAD (centinela)
- Cuando se ejecuta el LOAD especulativo, el hardware guarda la dirección a la que se ha realizado el acceso.
- Si los sucesivos STOREs no han accedido a esa dirección, la especulación es correcta. En caso contrario, la especulación ha fallado.
- Si la especulación ha fallado:
 - Si la especulación afecta al LOAD solamente, se vuelve a ejecutar cuando se llega al centinela.
 - Si se han ejecutado instrucciones que dependen del LOAD habrá que repetir todas esas instrucciones (se necesita mantener información de todas ellas en un trozo de código cuya dirección se incluye en la instrucción centinela)

