Tema 10. Introducción a la optimización de código

Organización de Computadores

LUIS ENRIQUE MORENO LORENTE
RAÚL PÉRULA MARTÍNEZ
ALBERTO BRUNETE GONZALEZ
DOMINGO MIGUEL GUINEA GARCIA ALEGRE
CESAR AUGUSTO ARISMENDI GUTIERREZ
JOSE CARLOS CASTILLO MONTOYA

Departamento de Ingeniería de Sistemas y Automática

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0
España.

R4000 PRESTACIONES

- El CPI ideal de 1 es difícil de conseguir debido a:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)

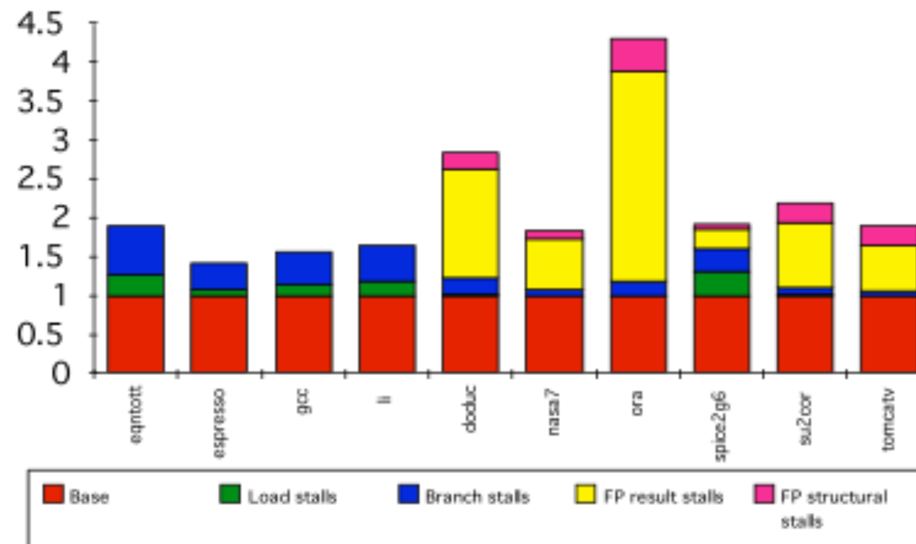

Imagen de Katz, 1996

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

SEGMENTACIÓN AVANZADA Y PARALELISMO AL NIVEL DE INSTRUCCIONES (ILP)

- ILP: Solapa la ejecución de instrucciones no relacionadas
- gcc -> 17% instr. de transferencia de control
  - Conjuntos de 5 instrucciones + 1 branch
  - Ir más allá de un bloque básico para conseguir un mayor paralelismo al nivel de instrucción
- Paralelismo a nivel de bucle es una oportunidad SW y HW

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

ALGUNOS TIPOS DE OPTIMIZACIÓN DE CÓDIGO

- Entre las posibles técnicas de optimización de código tenemos
  - Reordenación del código para eliminar burbujas y aprovechar la ventana de retardo.
  - Desenrollado de bucles.
  - Software pipelining.

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

EJEMPLO: BUCLES → DÓNDE ESTÁN LOS RIESGOS?

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

```
Loop:   LD      F0,0(R1)     ;F0=vector element
        ADDD    F4,F0,F2     ;add scalar from F2
        SD      0(R1),F4     ;store result
        SUBI    R1,R1,8      ;decrement pointer 8 Bytes (DW)
        BNEZ    R1,Loop      ;branch R1!=zero
        NOP                  ;delayed branch slot
```

¿Dónde están las detenciones?

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

EJEMPLO: BUCLES → DÓNDE ESTÁN LOS RIESGOS?

Loop:	1								...								17
LD F0, 0(R1)	IF	ID	Ex	M	Wb												
ADDD F4, F0,F2		IF	x	x	ID	Ex	M	Wb									
SD 0(R1) ,F4					IF	x	x	ID	Ex	M	--						
SUBI R1, R1,8								IF	ID	Ex	M	Wb					
BNEZ R1, Loop									IF	x	x	ID	Ex	M	--		
NOP												IF	ID	Ex	M	Wb	
NOP													IF	ID	Ex	M	Wb
														IF			

Ventana de retardo

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

BUCLE: REORDENACIÓN DEL CÓDIGO

```
1 Loop:   LD      F0,0(R1)
2         ADDD    F4,F0,F2
3         SUBI    R1,R1,8
4         BNEZ    R1,Loop     ; salto retardado (delayed branch)
5         SD      8(R1),F4    ; movida cuando se movió la anterior SUBI
```

Se intercambia BNEZ y SD cambiando la dirección de SD

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

BUCLE: REORDENACIÓN DEL CÓDIGO

Instrucción	1					...								14
Loop: LD F0,0(R1)	IF	ID	Ex	M	Wb									
ADDD F4,F0,F2		IF	x	x	ID	Ex	M	Wb						
SUBI R1,R1,8					IF	ID	Ex	M	Wb					
BNEZ R1,Loop						IF	x	x	ID	Ex	M	--		
SD 8(R1),F4									IF	ID	Ex	M	--	
NOP										IF	ID	Ex	M	Wb
											IF			

Se suma 8 para evitar
el decremento en el subi.

Ventana de retardo

Ahorra tres ciclos

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

DESENRROLLADO DEL BUCLE (4 VECES) (FORMA DIRECTA)

```
1 Loop:   LD      F0 , 0 (R1)
2         ADDD    F4, F0, F2
3         SD      0 (R1), F4         ;eliminado SUBI & BNEZ
4         LD      F6, -8 (R1)
5         ADDD    F8,F6,F2
6         SD      -8 (R1),F8         ;eliminado SUBI & BNEZ
7         LD      F10, -16 (R1)
8         ADDD    F12, F10, F2
9         SD      -16 (R1), F12      ;eliminado SUBI & BNEZ
10        LD      F14, -24 (R1)
11        ADDD    F16, F14, F2
12        SD      -24 (R1), F16
13        SUBI    R1, R1, #32        ;modificado a 4*8
14        BNEZ    R1, LOOP
15        NOP
```

Re-escribimos el bucle para minimizar las detenciones?

15 + 4 x (2+2) = 31 ciclos, o 7.8 por iteración. Asumimos que R1 es múltiplo de 4

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

DESENRROLLADO DEL BUCLE QUE MINIMIZA LAS DETENCIONES

```
1 Loop:  LD     F0,0(R1)
2        LD     F6,-8(R1)
3        LD     F10,-16(R1)
4        LD     F14,-24(R1)
5        ADDD   F4,F0,F2
6        ADDD   F8,F6,F2
7        ADDD   F12,F10,F2
8        ADDD   F16,F14,F2
9        SD     0(R1),F4
10       SD     -8(R1),F8
11       SD     -16(R1),F12
12       SUBI   R1,R1,#32
13       BNEZ   R1,LOOP
14       SD     8 (R1),F16        ; 8-32 = -24
```

16 ciclos de reloj, o 4 por iteración

Cúando es seguro mover instrucciones?

Qué suposiciones se han hecho cuando se ha movido el código?

- OK a mover stores después de SUBI incluso si el registro cambia
- OK a mover loads antes de stores: obtienen los datos adecuados?
- Cúando es seguro para el compilador hacer estos cambios?

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

DESENRROLLADO DEL BUCLE QUE MINIMIZA LAS DETENCIONES

1 Loop: LD F0,0(R1)	IF	ID	Ex	M	Wb														
2 LD F6,-8(R1)		IF	ID	Ex	M	Wb													
3 LD F10,-16(R1)			IF	ID	Ex	M	Wb												
4 LD F14,-24(R1)				IF	ID	Ex	M	Wb											
5 ADDD F4,F0,F2					IF	ID	Ex	M	Wb										
6 ADDD F8,F6,F2						IF	ID	Ex	M	Wb									
7 ADDD F12,F10,F2							IF	ID	Ex	M	Wb								
8 ADDD F16,F14,F2								IF	ID	Ex	M	Wb							
9 SD 0(R1),F4									IF	ID	Ex	M	Wb-						
10 SD -8(R1),F8										IF	ID	Ex	M	Wb					
11 SD -16(R1),F12											IF	ID	Ex	M	Wb				
12 SUBI R1,R1,#32												IF	x	x	ID	Ex	M	Wb	
13 BNEZ R1,LOOP															IF	ID	Ex	M	Wb
14 SD 8 (R1),F16 ; 8-32 = -24																			

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

Universidad Carlos III de Madrid
www.uc3m.es

PUNTO DE VISTA DEL COMPILADOR RESPECTO A LA REORDENACIÓN DEL CÓDIGO

- El compilador se ve afectado por las dependencias en el programa, incluso si un riesgo HW no depende de un cauce dado
- Planificar el código evita riesgos
- Dependencias de datos (verdadera) (RAW, si hay un riesgo por recursos HW)
  - Instrucción i produce un resultado usado por la instrucción j, o
  - Instrucción j depende de datos de la instrucción k, y la instrucción k depende de un dato de la instrucción i.
- Si son dependientes, no pueden ejecutarse en paralelo
- Fácil de determinar para los registros (nombres fijos)
- Difícil en el caso de la memoria:
  - Son iguales 100(R4) = 20(R6)?
  - En iteraciones diferentes de un bucle, son iguales 20(R6) = 20(R6)?
- Se requiere que el compilador conozca que si R1 no cambia entonces: `0(R1)` ≠ `-8(R1)` ≠ `-16(R1)` ≠ `-24(R1)`

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

PUNTO DE VISTA DEL COMPILADOR RESPECTO A LA REORDENACIÓN DEL CÓDIGO

- Otro tipo de dependencia denominado dependencia de nombre:
  dos instrucciones que usan el mismo nombre (registro o dirección de memoria) pero que no intercambian datos
- Antidependencia (WAR, si hay un riesgo por HW)
  - La instrucción j escribe en un registro o posición de memoria de la que otra instrucción i lee y la instrucción i se ejecuta primero
- Dependencia de Salida (WAW, si hay un riesgo por HW)
  - La instrucción i y la instrucción j escriben en el mismo registro o posición de memoria; el orden de las instrucciones debe de ser conservado.

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

DÓNDE ESTÁN LAS DEPENDENCIAS DE NOMBRE?

```
1 Loop:   LD      F0, 0 (R1)
2         ADDD    F4, F0, F2
3         SD      0 (R1), F4        ;eliminadas SUBI & BNEZ
4         LD      F0, -8 (R1)
2         ADDD    F4, F0, F2
3         SD      -8 (R1) ,F4       ;eliminadas SUBI & BNEZ
7         LD      F0, -16 (R1)
8         ADDD    F4, F0, F2
9         SD      -16 (R1), F4      ;eliminadas SUBI & BNEZ
10        LD      F0, -24 (R1)
11        ADDD    F4, F0, F2
12        SD      -24 (R1), F4
13        SUBI    R1, R1, #32       ;cambiado a  4*8
14        BNEZ    R1, LOOP
15        NOP
```

Cómo eliminarlas?

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0
España.

DÓNDE ESTÁN LAS DEPENDENCIAS DE NOMBRE?

1 Loop:	LD	F0, 0 (R1)	
2	ADDD	F4, F0, F2	
3	SD	0 (R1), F4	;eliminadas SUBI & BNEZ
4	LD	F6, -8 (R1)	
5	ADDD	F8, F6, F2	
6	SD	-8 (R1), F8	; eliminadas SUBI & BNEZ
7	LD	F10, -16 (R1)	
8	ADDD	F12, F10, F2	
9	SD	-16 (R1), F12	; eliminadas SUBI & BNEZ
10	LD	F14, -24 (R1)	
11	ADDD	F16, F14, F2	
12	SD	-24 (R1), F16	
13	SUBI	R1, R1, #32	;cambiado a 4*8
14	BNEZ	R1, LOOP	
15	NOP		

Haciendo un “renombrado de registros” (“register renaming”)

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

DEPENDENCIAS ACARREADAS

- Ejemplo: Dónde están las dependencias de datos? (A,B,C distintas y no se solapan)

```
for (i=1; i<=100; i=i+1) {
        A[i] = A[i-1] + C[i-1];    /* S1 */
        B[i] = B[i-1] + A[i];} /* S2 */
```

- S1 utiliza un valor calculado por S1 en una iteración anterior, dado que la iteración i calcula A[i] el cual es leído en la iteración i+1(A[i-1]). Lo mismo le ocurre a S2 para B[i] y B[i-1].
  - Esto es una dependencia acarreada (loop-carried dependence): entre iteraciones
  - Esto implica que las iteraciones son dependientes, y no pueden ser ejecutadas en paralelo
  - No es el caso del ejemplo anterior: cada iteración era independiente
- Por otro lado S2 utiliza el valor A[i] calculado por S1 en la misma iteración.
  - No es acarreada
  - Se pueden ejecutar en paralelo varias iteraciones del bucle siempre que las dos instrucciones se ejecuten juntas

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0
España.

PASOS DEL COMPILADOR PARA DESENRROLLAR

- Verificar si se puede mover S.D después de DSUBI y BNEZ, y encontrar la cantidad para ajustar el offset del S.D
- Determinar si el desenrollado del bucle será útil: se determina qué iteraciones del bucle son independientes
- Renombrar los registros para evitar dependencias de nombre
- Eliminar los tests extras e instrucciones de salto, y ajustar la terminación del bucle y el código que se itera
- Determinar los loads y stores en el bucle desenrollado que pueden ser intercambiados, para lo que se observa qué loads y stores de diferentes iteraciones son independientes
  - requiere analizar las direcciones de memoria y buscar para que no haya referencias a la misma dirección de memoria.
- Planificar el código, preservando cualquier dependencia que sea necesaria para obtener el mismo resultado que el código original

Este obra se publica bajo una
[licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.]

DEPENDENCIAS DE CONTROL

- Ejemplo

```
if p1 {S1;};
if p2 {S2;};
```

- S1 es dependiente de control respecto a p1
- S2 es dependiente de control respecto a p2 pero no respecto a p1.

- Dos restricciones (obvias) en las dependencias de control:
- Una instrucción que es dependiente de control respecto a un branch no puede moverse antes (fuera) del branch para que su ejecución no este controlada por el branch.

- Una instrucción que no es dependiente de control respecto a un branch no puede ser movida después (dentro) del branch ya que su ejecución entonces es controlada por el branch.

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

OTRA POSIBILIDAD: SOFTWARE PIPELINING

- Observación: si las iteraciones de los bucles son independientes, entonces es posible conseguir un mayor ILP tomando instrucciones de diferentes iteraciones
- Software pipelining: reorganiza los bucles de forma que cada iteración está formada por instrucciones elegidas de diferentes iteraciones del bucle original (~ Tomasulo in SW)

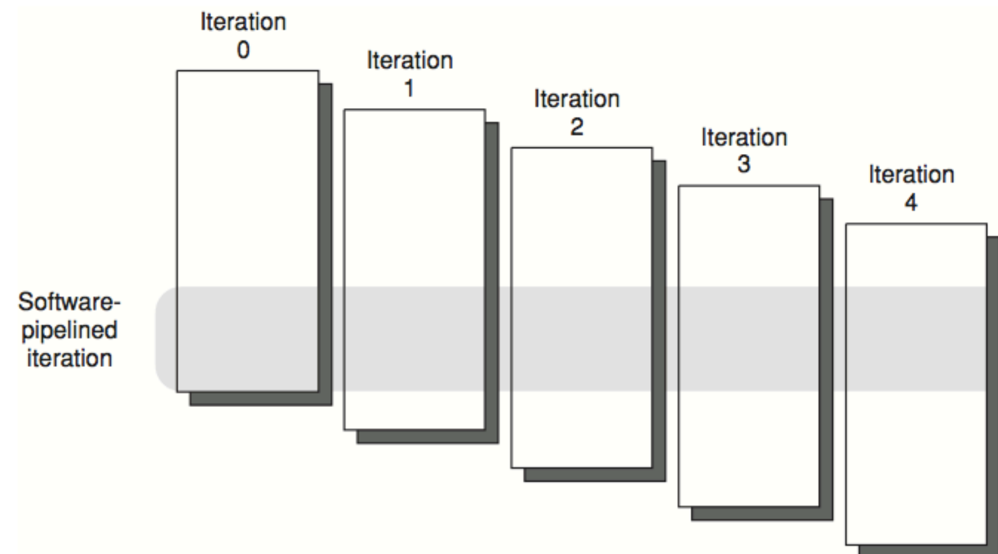

Imagen de Hennessy-Patterson,1997

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

SOFTWARE PIPELINING

- Desenrollado simbólico del bucle
  - Maximiza resultado
  - Código más pequeño que con desenrollado convencional
  - Llena y vacía el cauce sólo una vez por bucle, en vez de una vez por iteración desenrollada en el caso de desenrollado del bucle

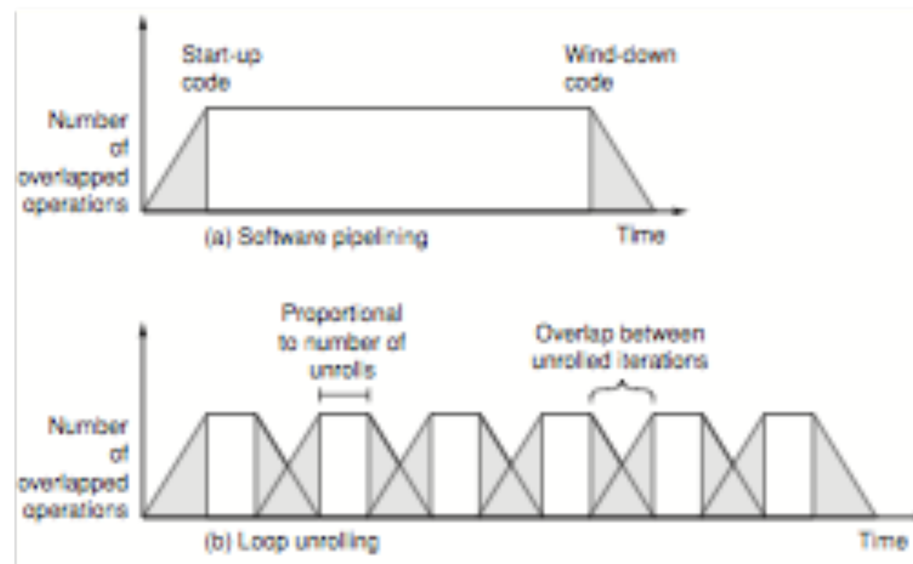

Imagen de Hennessy-Patterson,1997

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

SOFTWARE PIPELINING: EJEMPLO 1

- Partimos de un código ensamblador con dependencias de flujo

```
        ADDI R6, R0, #40
Loop:
        ADD R1, R2, R3
        SUBI R4, R1, R0
        SD R4, 0(R6)
        SUBI R6, R6, #8
        BNEZ R6, loop
```

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

SOFTWARE PIPELINING: EJEMPLO 1

- Primero identificamos qué instrucciones se van a repetir (como al desenrollar)
- Estas instrucciones se tienen que ejecutar secuencialmente por las dependencias

```
        ADDI R6, R0, #40
Loop:
        ADD R1, R2, R3
        SUBI R4, R1, R0
        SD R4, 0(R6)
        SUBI R6, R6, #8
        BNEZ R6, loop
```

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

SOFTWARE PIPELINING: EJEMPLO 1

- Es posible simular una segmentación a nivel software, dado que no existe dependencia entre instrucciones de distintas iteraciones

ADD R1, R2, R3		
SUBI R4, R1, R0	ADD R1, R2, R3	
SD R4, 0(R6)	SUBI R4, R1, R0	ADD R1, R2, R3
	SD R4, -8(R6)	SUBI R4, R1, R0
		SD R4, -16(R6)

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

SOFTWARE PIPELINING: EJEMPLO 1

- De aquí podemos sacar un nuevo patrón de instrucciones que se repiten, aparte de un prólogo y epílogo.

ADD R1, R2, R3		
SUBI R4, R1, R0	ADD R1, R2, R3	
SD R4, 0(R6)	SUBI R4, R1, R0	ADD R1, R2, R3
	SD R4, -8(R6)	SUBI R4, R1, R0
		SD R4, -16(R6)

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

SOFTWARE PIPELINING: EJEMPLO 1

- Nuevamente podemos reordenar el código, aún sin ser un bucle

```
ADD R1, R2, R3
SUBI R4, R1, R0
ADD R1, R2, R3

SD R4, 0(R6)
SUBI R4, R1, R0
ADD R1, R2, R3

SD R4, -8(R6)
SUBI R4, R1, R0
SD R4, -16(R6)
```

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0
España.

SOFTWARE PIPELINING: EJEMPLO 1

- Si añadimos el bucle, quedaría así (cuidado con los índices en los accesos a memoria)

```
        ADDI R6, R0, #40
        ADD R1, R2, R3
        SUBI R4, R1, R0
        ADD R1, R2, R3
loop:   SD R4, 0(R6)
        SUBI R4, R1, R0
        ADD R1, R2, R3
        SUBI R6, R6, #8
        BNEZ R6, loop
        SD R4, -8(R6)
        SUBI R4, R1, R0
        SD R4, -16(R6)
```

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

SOFTWARE PIPELINING : EJEMPLO 2

Antes: Desenrollando 3 veces

```
1    L.D     F0,0(R1)
2    ADD.D   F4,F0,F2
3    S.D     0(R1),F4
4    L.D     F6,-8(R1)
5    ADD.D   F8,F6,F2
6    S.D     -8(R1),F8
7    L.D     F10,-16(R1)
8    ADD.D   F12,F10,F2
9    S.D     -16(R1),F12
10   DSUBUI  R1,R1,#24
11   BNEZ    R1,LOOP
```

Después: Software Pipelined

```
1    S.D     0(R1),F4 ;     Stores X[i]
2    ADD.D   F4,F0,F2 ;     Adds to X[i-1]
3    L.D     F0,-16(R1);    Loads X[i-2]
4    DSUBUI  R1,R1,#8
5    BNEZ    R1,LOOP
```

5 cycles per iteration

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

SOFTWARE PIPELINING : EJEMPLO

- Falta el código de comienzo y finalización (start-up & finish-up code)
  - Comienzo: instrucciones que corresponden a las iteraciones 1 y 2 que no se han ejecutado
    - L.D iteraciones 1 y 2
    - ADD.D iteración 1
  - Finalización: instrucciones que corresponden a las dos últimas iteraciones que no se han ejecutado
    - ADD.D última iteración
    - S.D dos últimas iteraciones

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

ESQUEMAS HW: PARALELISMO DE INSTRUCCIÓN

- Por qué en HW y en run time?
  - Porque funciona bien cuando no se conocen las dependencias reales en tiempo de compilación
  - El compilador es más simple
  - El código para una máquina funciona bien en otra
- Idea clave: permitir que instrucciones situadas después de una detención prosigan su ejecución

```
DIVD  F0,F2,F4
ADDD  F10,F0,F8
SUBD  F12,F8,F14
```

  - Permitir la ejecución out-of-order execution => finalización out-of-order
  - Aparece en el Scoreboard de la máquina CDC 6600 en 1963

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 España.

REFERENCIAS

[Katz, 19996] R. H. Katz (1996) Computer Science 252 course. http://bnrg.cs.berkeley.edu/~randy/Courses/CS252.S96/Lecture09.pdf

[Hennessy-Patterson, 1997]David A. Patterson, John L. Hennessy (1997) Computer Organization & Design: The Hardware/Software Interface, Second Edition. Morgan Kaufmann.

Este obra se publica bajo una
licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0
España.