UT 2. Computadores segmentados

Tema 2.2 Unidades multiciclo y gestión estática de instrucciones

J. Flich, P. López, V. Lorente,
A. Pérez, S. Petit, J.C. Ruiz, S. Sáez, J. Sahuquillo

Departamento de Informática de Sistemas y Computadores
Universitat Politècnica de València

UNIVERSIDAD POLITECNICA DE VALENCIA

DISCA

Índice

1 Operaciones multiciclo

2 Tipos de dependencias

3 Técnicas para aumentar ILP

4 Loop Unrolling

5 Software Pipelining

Bibliografía

John L. Hennessy and David A. Patterson.
Computer Architecture, Fifth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2012.

Índice

1 Operaciones multiciclo

2 Tipos de dependencias

3 Técnicas para aumentar ILP

4 Loop Unrolling

5 Software Pipelining

Problema

- Instrucciones enteras que realizan operaciones más complicadas (`mult, div`).
- Instrucciones de coma flotante (`add.s, add.d, mult.s, div.s, ...`)

$\rightarrow$ necesitan mayor tiempo en la fase EX $\Rightarrow$ ¿Cómo segmentar la unidad de instrucción con estas operaciones?

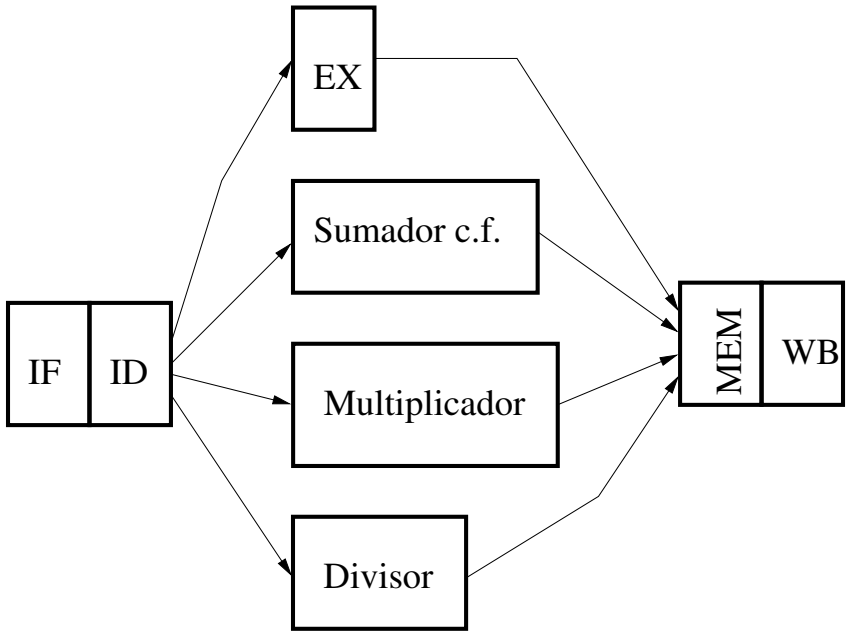
Soluciones

- Aumentar el período de reloj de la unidad de instrucción $\rightarrow$ se ralentiza toda la máquina.
- Permitir que la fase EX de estas operaciones se prolongue durante varios ciclos de reloj (**operaciones multiciclo**) $\rightarrow$ el período de reloj no cambia.
  Ejemplo: `mult` con un operador de 40ns

  Fases: IF ID EX EX EX EX MEM WB

Múltiples operadores

- Como las nuevas instrucciones requieren un *hardware* específico, se prefiere añadir nuevos operadores *especializados* en lugar de un único operador multifunción.
- En la fase ID se envía la instrucción al operador adecuado.
- Cuando se finaliza la operación, se envía a la fase MEM.



Tipos de operadores multiciclo

Los operadores multiciclo añadidos pueden ser convencionales o segmentados.
$\rightarrow$ Parámetros característicos:

- latencia o tiempo de evaluación, (tiempo en obtener el primer resultado), y
- tasa de iniciación, *IR* (inversa del tiempo entre resultados). Si está segmentado, $IR = 1$

Ejemplo de operadores añadidos al MIPS

- Sum/rest coma flotante segmentado. T_{ev}: 4. IR: 1 cada ciclo.
- Mult entero/coma flotante segmentado. T_{ev}:7. IR: 1 cada ciclo.
- Div entero/coma flotante convencional. T_{ev}: 24. IR: 1 cada 24 ciclos.

Ejemplo de ejecución de código con operaciones multiciclo

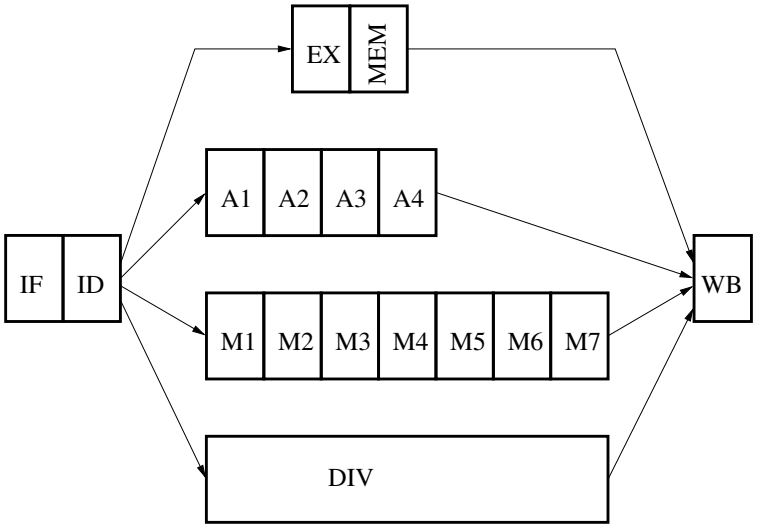
```
DADD R1,R2,R3   IF ID EX ME WB
ADD.D F0,F2,F4     IF ID A1 A2 A3 A4  ME  WB
MULT.D F6,F8,F10      IF ID M1 M2 M3  M4  M5  M6  M7  ME  WB
MULT.D F12,F14,F16       IF ID M1 M2  M3  M4  M5  M6  M7  ME  WB
DIV.D F18,F20,F22           IF ID DIV DIV DIV DIV DIV DIV DIV DIV
```

Revisión de la unidad de instrucción segmentada con op. multiciclo:

- Todas las instrucciones ya no duran el mismo número de ciclos de reloj.
- La fase MEM está vacía en las operaciones multiciclo.

$\Rightarrow$ Eliminar la fase MEM en las operaciones multiciclo.

MIPS segmentado incluyendo los nuevos operadores



- Se necesitan nuevos registros inter-etapas:
  1. ID/EX $\rightarrow$ ID/EX, ID/A1, ID/M1, ID/DIV
  2. A1/A2, A2/A3, . . . , A3/A4
  3. M1/M2, M2/M3, . . . , M6/M7
- Se necesita un multiplexor a la entrada de la etapa WB.

Riesgos estructurales por uso de unidades con IR $<$ 1 op/ciclo

Ejemplo:

i1 DIV.D ... (Tev = 6 ciclos, IR = 1 op/6 ciclos)
i2 DIV.D ...
i3 Instrucción entera
i4 Instrucción entera

Problema: ¡Dos instrucciones en el mismo operador!

i1	IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	WB	
i2		IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	WB
i3			IF	ID	EX	ME	WB			
i4				IF	ID	EX	ME	WB		

Riesgos estructurales por uso de unidades con IR $<$ 1 op/ciclo (cont.)

Solución:
La 2ª instrucción debe esperar en ID hasta que el operador se libere, insertando ciclos de parada.

i1	IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	WB				
i2		IF	id	id	id	id	id	ID	DIV	DIV	DIV	DIV	...
i3			if	if	if	if	if	IF	ID	EX	M	WB	
i4									IF	ID	EX	M	

$\Rightarrow$ Lógica de detección concentrada en ID.
$\Rightarrow$ En cuanto una instrucción se detiene en ID, cesa la búsqueda de instrucciones (IF).

Riesgos estructurales por escritura simultánea en el banco de registros

Ejemplo:

i1 MUL.D ... (Tev = 4 ciclos, IR = 1 op/ciclo)
i2 ADD.D ... (Tev = 3 ciclos, IR = 1 op/ciclo)
i3 ADD.D ...
i4 Instrucción entera

Problema: ¡Varias instrucciones en WB!

i1	IF	ID	M1	M2	M3	M4	**WB**	
i2		IF	ID	A1	A2	A3	**WB**	
i3			IF	ID	A1	A2	A3	**WB**
i4				IF	ID	EX	ME	**WB**

$\rightarrow$ reducción del número de instrucciones que acceden simultáneamente a WB: bancos de registros independientes para enteros y coma flotante.

Bancos independientes de registros enteros y de coma flotante

Objetivo: que las instrucciones enteras y las instrucciones de coma flotante utilicen bancos de registros *distintos.*

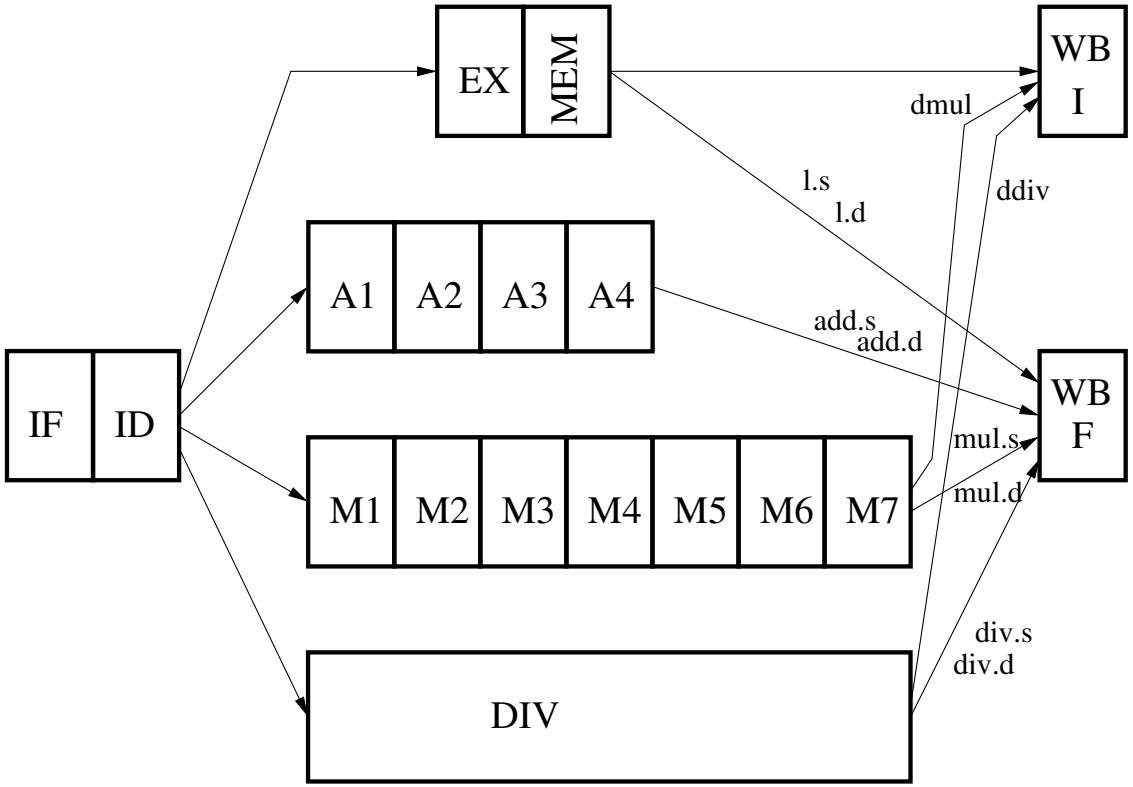
Ventajas

- Se reduce el número de riesgos estructurales.
- Se duplica el número total de registros, sin complicar la lógica de decodificación, sin aumentar el tiempo de acceso y sin añadir más bits en el formato.
- Se duplica el ancho de banda del banco de registros, sin añadir más puertos ni aumentar el tiempo de acceso

Inconvenientes

- A veces, es necesario transferir información entre los dos bancos de registros (Instrucciones `MFC0, MTC0` y `MFC1, MTC1`).
- Se limita a priori el número de registros de cada tipo.

MIPS segmentado con dos bancos de registros independientes



- Para eliminar riesgos estructurales innecesarios, las stores de coma flotante terminan en la etapa MEM.

Riesgos estr. por escritura simultánea en el banco de reg. (2)

Problema:

i1 MUL.D ... (Tev = 4 ciclos, IR = 1 op/ciclo)
i2 ADD.D ... (Tev = 3 ciclos, IR = 1 op/ciclo)
i3 ADD.D ...
i4 Instrucción entera

i1	IF	ID	M1	M2	M3	M4	**WB**	
i2		IF	ID	A1	A2	A3	**WB**	
i3			IF	ID	A1	A2	A3	WB
i4				IF	ID	EX	ME	WB

Riesgos estr. por escritura simultánea en el banco de reg. (2) (cont.)

Soluciones:

- Aumentar el número de puertos.
  Por término medio se realiza sólo una escritura por ciclo (ya que se lanza una instrucción/ciclo, y sólo se escribe una vez por instrucción) $\rightarrow$ no es eficiente.
- Insertar ciclos de parada.
  En la fase ID se comprueba si la instrucción escribirá en el banco de registros en el mismo ciclo que otras ya lanzadas en ejecución.

i1	IF	ID	M1	M2	M3	M4	WB		
i2		IF	id	ID	A1	A2	A3	WB	
i3			if	IF	ID	A1	A2	A3	WB
i4					IF	ID	EX	ME	WB

$\Rightarrow$ Lógica de detección concentrada en ID.
$\Rightarrow$ La detención en ID impide buscar nuevas instrucciones (IF).

Riesgos de datos

Se producen riesgos RAW (Read After Write) cuando una instrucción produce un resultado que es consumido por otra instrucción posterior cercana.

Ejemplo:

i1	ADD.D **F0**, F2, F4
i2	MUL.D F6, **F0**, F8
i3	Instrucción entera
i4	Instrucción entera
i5	Instrucción entera

Riesgos de datos (cont.)

Solución: Insertar de ciclos de parada hasta poder aplicar un cortocircuito.

ADD.D **F0**,. . .	IF	ID	A1	A2	A3	A4	WB			
MUL.D . . . ,**F0**		IF	id	id	id	ID	M1	M2	M3	. . .
i3			if	if	if	IF	ID	EX	ME	WB
i4							IF	ID	EX	ME
i5								IF	ID	EX

$\Rightarrow$ Lógica de detección concentrada en ID.
$\Rightarrow$ La detención en ID impide buscar nuevas instrucciones (IF).

Con operaciones multiciclo, la fase de ejecución puede durar varios ciclos de reloj $\rightarrow$ la penalización (número de *stalls* ó ciclos de parada) puede llegar a ser elevada.

Nuevos riesgos de datos: WAW

Se producen riesgos WAW (Write After Write) cuando dos instrucciones cercanas escriben en el mismo registro.

1. Además del riesgo WAW, hay un riesgo RAW.

```
i1   MUL.D F0,F2,F4
i2   DIV.D F6,F0,F8
i3   ADD.D F0,F10,F12
```

   La resolución del riesgo RAW, insertando ciclos de parada en ID, soluciona también el WAW:

i1	MUL.D **F0**,F2,F4	IF	ID	M1	M2	M3	M4	M5	M6	M7	**WB**	
i2	DIV.D F6,F0,F8		IF	id	id	id	id	id	id	ID	DIV	...
i3	ADD.D **F0**,F10,F12			if	if	if	if	if	if	IF	ID	A1

Nuevos riesgos de datos: WAW (cont.)

2 Ejemplo 2: Sólo hay riesgo WAW.

i1 MUL.D **F0**,F2,F4
i2 ADD.D **F0**,F10,F12

Problema: El orden de las escrituras no es el correcto. El registro F0 se queda con el valor intermedio y no con el final.

i1	MUL.D **F0**,F2,F4	IF	ID	M1	M2	M3	M4	M5	M6	M7	**WB**
i3	ADD.D **F0**,F10,F12		IF	ID	A1	A2	A3	A4	**WB**		

Solución: Detección en ID e inserción de ciclos de parada.

i1	MUL.D **F0**,F2,F4	IF	ID	M1	M2	M3	M4	M5	M6	M7	**WB**	
i3	ADD.D **F0**,F10,F12		IF	id	id	id	ID	A1	A2	A3	A4	**WB**

¿Pueden ocurrir situaciones donde se lanzan dos escrituras sobre el mismo registro sin lecturas intermedias?

Nuevos riesgos de datos: WAW (cont.)

Considerar el siguiente código y la optimización del compilador por detección del camino más frecuente:

```
if (flag == 0)
   /* camino más frecuente */
   x = x*2;
else
   x = a;
cont: ...
```

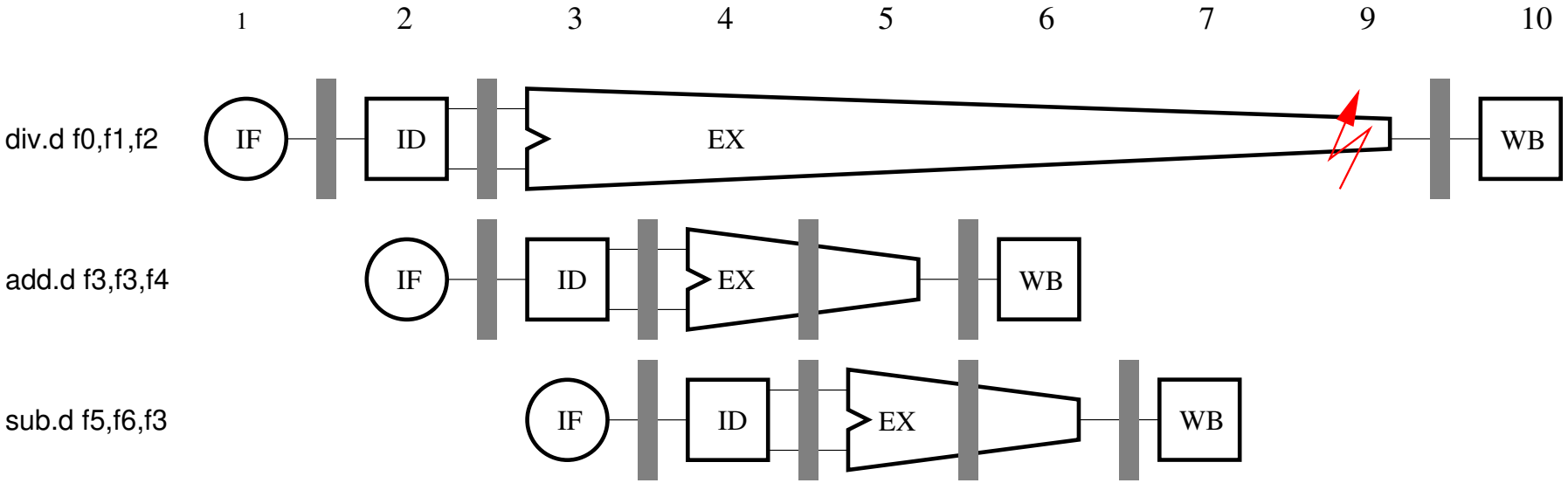
$r10 \equiv flag, f0 \equiv x, f2 \equiv 2, r11 \equiv @a$

```
mul.d f0,f0,f2
beqz r10,cont
l.d f0,0(r11)
cont: ...
```

Si el salto no es efectivo, hay un riesgo WAW entre `MUL.D` y `L.D`.

Tratamiento de las excepciones

Problema:
Con operaciones multiciclo altera el orden de finalización de la ejecución de las instrucciones → en el momento en que se produce la excepción, algunas instrucciones *posteriores* han terminado ya:



Índice

1 Operaciones multiciclo

2 Tipos de dependencias

3 Técnicas para aumentar ILP

4 Loop Unrolling

5 Software Pipelining

Paralelismo a nivel de instrucciones

El problema:

Unidad segmentada sin operaciones multiciclo $\longrightarrow$ pocos ciclos de parada $\longrightarrow$ CPI ≈ 1

Unidad segmentada con operaciones multiciclo $\longrightarrow$ muchos ciclos de parada $\longrightarrow$ CPI $>> 1$

Instruction Level Paralelism o **ILP**:

- Posibilidad de solapamiento en las secuencias de instrucciones.
- Depende de si las instrucciones son independientes.

ILP $\uparrow \longrightarrow$ pocos conflictos $\longrightarrow$ pocos ciclos de parada $\longrightarrow$ **CPI** $\downarrow$

Dependencia entre instrucciones del programa

Dos instrucciones son independientes si pueden ejecutarse simultáneamente sin ningún problema $\Leftrightarrow$ Se pueden reordenar.

Tipos de dependencias:
- De datos
- De nombre
- De control

Dependencias de datos

Dadas dos instrucciones *i* y *j*, *j* lógicamente después de *i*: $\left\{ \begin{array}{l} \text{instr } i \\ \ldots \\ \text{instr } j \end{array} \right.$

existe una dependencia de datos si:

- La instrucción *i* produce un resultado utilizado por la *j*
- Hay una dependencia de datos entre la instrucción *i* y otra *k*, y la instrucción *k* produce un resultado utilizado por la *j*. Esta cadena puede ser tan larga como el propio programa.

Ejemplo:

```
loop:  L.D F0,0(R1)
       ADD.D F4,F0,F2
       S.D F4,0(R1)
       ...
```

El dato compartido que causa la dependencia puede estar almacenado en los registros o en la memoria.

Dependencias de nombre

Se originan cuando dos instrucciones utilizan el mismo registro o posición de memoria, pero no hay flujo de datos entre las instrucciones asociadas con este nombre.

Dadas dos instrucciones *i* y *j*, *j* lógicamente después de *i*: $\left\{ \begin{array}{l} \text{instr } i \\ \ldots \\ \text{instr } j \end{array} \right.$

se pueden producir las siguientes dependencias de nombre:

Antidependencia

La instrucción *j* escribe sobre un registro o una posición de memoria que la instrucción *i* lee.

Dependencia de salida

Las dos instrucciones *i* y *j* escriben sobre el mismo registro o posición de memoria.

Dependencias de nombre (cont.)

Ejemplo:

```
loop:  L.D ***F0***,0(R1)
       ADD.D F4,*F0*,F2
       S.D F4,0(R1)
       L.D ***F0***,-8(R1)
       ADD.D F4,F0,F2
       S.D F4,-8(R1)
       ...
```

Dependencias de control

Determina la ordenación de algunas instrucciones respecto a un salto
$\rightarrow$ hay que ejecutar antes la instrucción de salto.

Toda instrucción (excepto las primeras del programa) tiene una dependencia de control con algún salto.

Ejemplo:

```
            ...
            BEQZ R1, exit
            L.D F10,0(R1)
            ADD.D F14,F10,F2
            S.D F14,0(R1)
            DSUB R1,R1,#8
            ...
      exit:
```

Dependencias de control (cont.)

El problema

ILP $\downarrow$ $\longrightarrow$ Presencia simultánea de instrucciones dependientes en la unidad segmentada $\longrightarrow$ (posibles) riesgos $\longrightarrow$ (posibles) ciclos de parada $\longrightarrow$ CPI $\uparrow$

$\Downarrow$

Dependencia:	*Riesgo:*
De datos	RAW
Antidependencia	WAR
De salida	WAW
De control	De control

Índice

1 Operaciones multiciclo

2 Tipos de dependencias

3 Técnicas para aumentar ILP

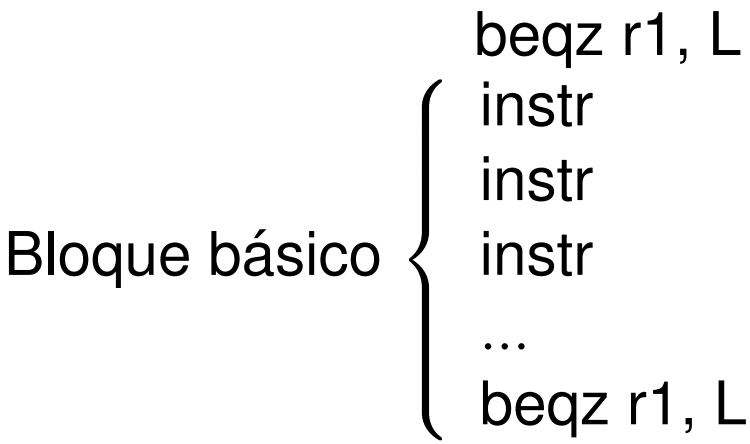
4 Loop Unrolling

5 Software Pipelining

Concepto de bloque básico

El objetivo es el de aumentar el ILP de las instrucciones que están simultáneamente en ejecución en la unidad segmentada.

Se consideran secuencias de instrucciones comprendidas entre instrucciones de salto (denominadas **bloques básicos**) y se estudia el grado de paralelismo extraíble de la ejecución de éstas.



¿Hay suficiente ILP en un solo bloque básico?

1. Estadísticas: 15 % son saltos $\rightarrow$ 6 ó 7 instrucciones por bloque básico.
2. Las instrucciones de un bloque básico suelen tener dependencias.

$\rightarrow$ Necesidad de explotar ILP a lo largo de múltiples bloques básicos: hay que ejecutar en paralelo instrucciones procedentes de distintos bloques básicos.

Un caso particular es el *Loop-level parallelism*, que explota ILP en las iteraciones de un bucle, solapando iteraciones:

```
for i := 1 to 1000 do
    x[i] := x[i] + s;
```

¿Cómo aumentar ILP solapando varios bloques básicos?

Gestión estática de instrucciones

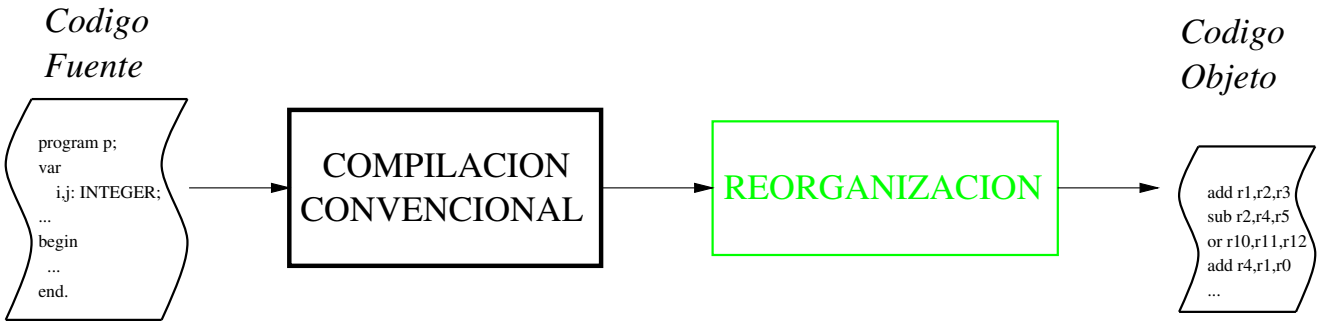
El compilador reordena/modifica el código.

Gestión dinámica de instrucciones

El *hardware* reordena las instrucciones en tiempo de ejecución.

Gestión estática de instrucciones

El compilador reordena/modifica el código para aumentar ILP reduciendo/eliminando las dependencias o sus efectos (riesgos y ciclos de parada):



Algunas técnicas de gestión estática de instrucciones:

Loop-Unrolling Separa las instrucciones dependientes y explota ILP a lo largo de varios bloques básicos.

Software Pipelining Reordena el código para separar las instrucciones con dependencias.

Índice

1 Operaciones multiciclo

2 Tipos de dependencias

3 Técnicas para aumentar ILP

4 Loop Unrolling

5 Software Pipelining

Ejemplo: $\vec{Z} = a + \vec{Y}$

```
        i = 0;
        while (i<n) {
          z[i] = a + y[i];
          i = i+1;
        }
```

↓

```
start:
        daddi r1, r0, y      ; r1 = dirección de y
        daddi r2, r0, z      ; r2 = dirección de z
        l.d f0, a(r0)        ; f0 = a
        daddi r3, r1, #512   ; 64 elementos son 512 bytes
loop:
        l.d f2, 0(r1)        ; L
        add.d f4, f0, f2     ; A
        s.d f4,0(r2)         ; S
        daddi r1, r1, #8
        daddi r2, r2, #8
        dsub r4, r3, r1
        bnez r4, loop
```

Ejemplo: $\vec{Z} = a + \vec{Y}$ (cont.)

Datos:

1 Sumador c.f. (Tev = 5 ciclos, IR = 1 op/ciclos)

$\rightarrow$ En cada iteración del bucle se insertan 4 ciclos de parada:

`l.d f2,0(r1)`	IF	ID	EX	M	WB						
`add.d f4,f0,f2`		IF	id	ID	A1	A2	A3	A4	A5	WB	
`s.d f4,0(r2)`			if	IF	id	id	id	ID	EX	M	
`dadd r1,r1,#8`					if	if	if	IF	ID	EX	M
`dadd r2,r2,#8`									IF	ID	EX
`dsub r4,r3,r1`										IF	ID
`bnez r4,loop`											IF

máx + 1

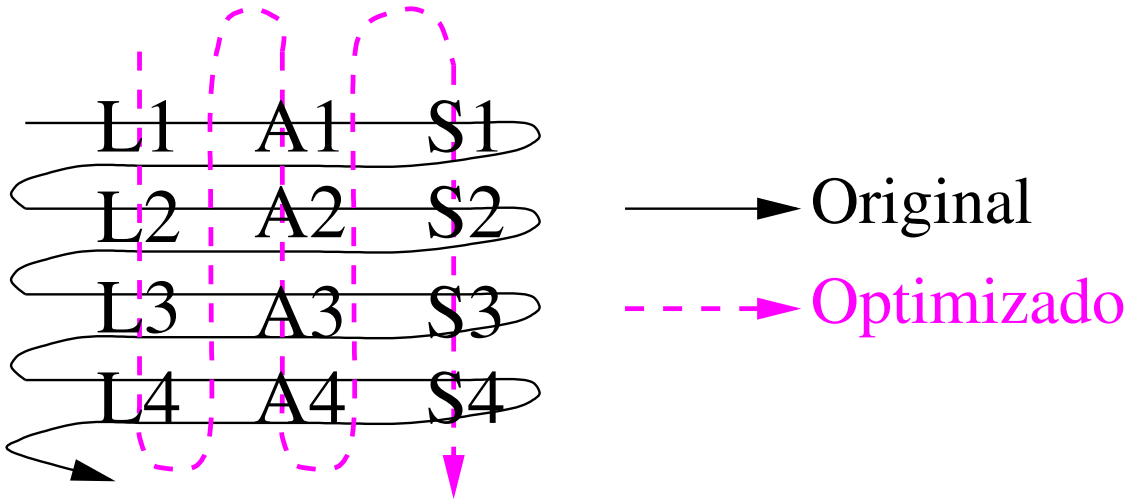
Idea básica

Se replica el código base del bucle varias veces, disminuyendo así el número de iteraciones realizadas.

```
i = 0;
while (i<n) {
   z[i] = a + y[i];
   z[i+1] = a + y[i+1];
   z[i+2] = a + y[i+2];
   z[i+3] = a + y[i+3];
   i = i+4;
} /* endwhile */
```

Idea básica (cont.)

- Reduce la sobrecarga originada por las instrucciones de control de los bucles.
- Al aumentar el tamaño del bloque básico aumenta la flexibilidad que el compilador tiene para separar las instrucciones con dependencia de datos:



Código del bucle replicado 4 veces:

```
start:
        daddi r1, r0, y       ; r1 = dirección de y
        daddi r2, r0, z       ; r2 = dirección de z
        l.d f0, a(r0)         ; f0 = a
        daddi r3, r1, #512    ; 64 elem. son 512 bytes
loop:
        l.d f2, 0(r1)         ; (1.1)
        add.d f4, f0, f2      ; (1.2)
        s.d f4,0(r2)          ; (1.3)
        l.d f2, 8(r1)         ; (2.1)
        add.d f4, f0, f2      ; (2.2)
        s.d f4,8(r2)          ; (2.3)
        l.d f2, 16(r1)        ; (3.1)
        add.d f4, f0, f2      ; (3.2)
        s.d f4,16(r2)         ; (3.3)
        l.d f2, 24(r1)        ; (4.1)
        add.d f4, f0, f2      ; (4.2)
        s.d f4,24(r2)         ; (4.3)
        daddi r1, r1, #32     ; 4 veces 8 = 32
        daddi r2, r2, #32
        dsub r4, r3, r1
        bnez r4, loop
```

Optimización:

Es necesario realizar un *renombrado de registros* para eliminar las dependencias de nombre:

```
loop:   l.d f2, 0(r1)
        add.d f4, f0, f2
        s.d f4,0(r2)
        l.d f2, 8(r1)
        add.d f4, f0, f2
        s.d f4,8(r2)
        ...
```

→

```
loop:   l.d f2, 0(r1)
        add.d f4, f0, f2
        s.d f4,0(r2)
        l.d f12, 8(r1)
        add.d f14, f0, f12
        s.d f14,8(r2)
        ...
```

Código del bucle replicado 4 veces y *optimizado:*

```
start:
        daddi r1, r0, y       ; r1 = dirección de y
        daddi r2, r0, z       ; r2 = dirección de z
        l.d f0, a(r0)         ; f0 = a
        daddi r3, r1, #512    ; 64 elem. son 512 bytes
loop:
        l.d f2, 0(r1)         ; (1.1)
        l.d f6, 8(r1)         ; (2.1)
        l.d f10, 16(r1)       ; (3.1)
        l.d f14, 24(r1)       ; (4.1)
        add.d f4, f0, f2      ; (1.2)
        add.d f8, f0, f6      ; (2.2)
        add.d f12, f0, f10    ; (3.2)
        add.d f16, f0, f14    ; (4.2)
        s.d f4,0(r2)          ; (1.3)
        s.d f8,8(r2)          ; (2.3)
        s.d f12,16(r2)        ; (3.3)
        s.d f16,24(r2)        ; (4.3)
        daddi r1, r1, #32
        daddi r2, r2, #32
        dsub r4, r3, r1
        bnez r4, loop
```

Código del bucle replicado 4 veces y *optimizado:* (cont.)

⇒ Se han separado las instrucciones dependientes, insertando tantas instrucciones entre ellas como número de ciclos de parada había que insertar en el peor caso (3 ciclos entre add.d y s.d).

`l.d f2,0(r1)`	IF	ID	EX	M	WB										
`l.d f6,8(r1)`		IF	ID	EX	M	WB									
`l.d f10,16(r1)`			IF	ID	EX	M	WB								
`l.d f14,24(r1)`				IF	ID	EX	M	WB							
`add.d f4,f0,f2`					IF	ID	A1	A2	A3	A4	A5	WB			
`add.d f8,f0,f6`						IF	ID	A1	A2	A3	A4	A5	WB		
`add.d f12,f0,f10`							IF	ID	A1	A2	A3	A4	A5	WB	
`add.d f16,f0,f14`								IF	ID	A1	A2	A3	A4	A5	WB
`s.d f4,0(r2)`									IF	ID	EX	M			
`s.d f8,8(r2)`										IF	ID	EX	M		
`s.d f12,16(r2)`											IF	ID	EX	M	
`s.d f16,24(r2)`												IF	ID	EX	M

⇒ Ya no es necesario insertar ciclos de parada.

Índice

1 Operaciones multiciclo

2 Tipos de dependencias

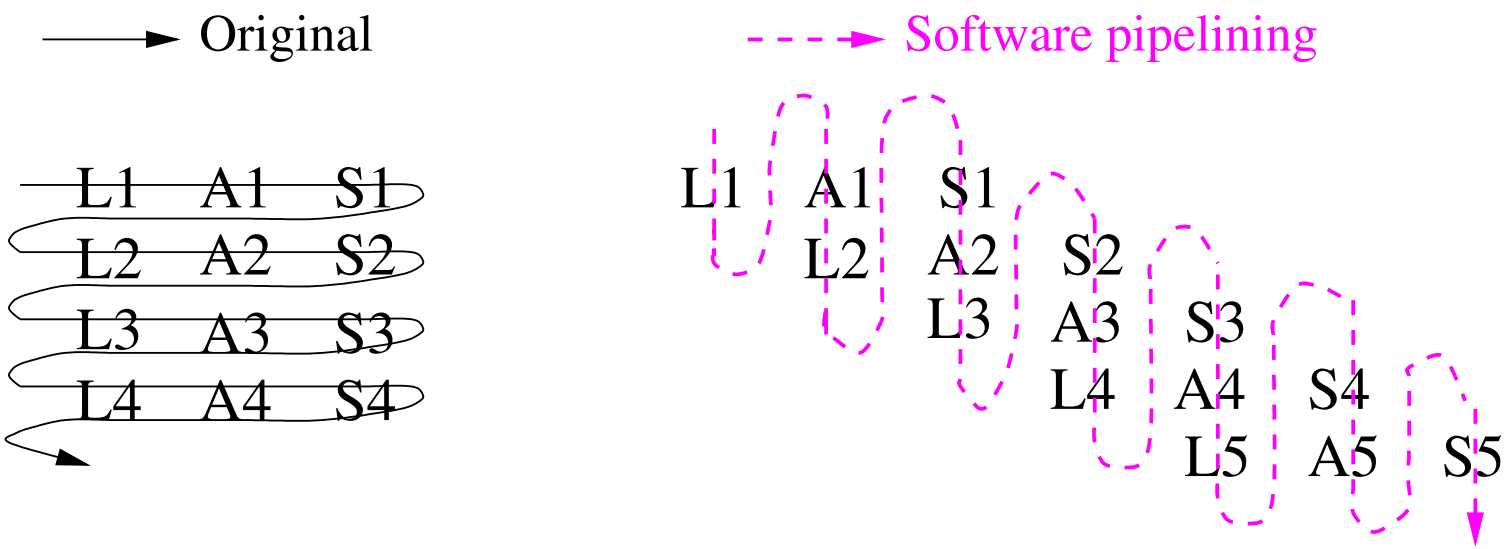
3 Técnicas para aumentar ILP

4 Loop Unrolling

5 Software Pipelining

Idea básica

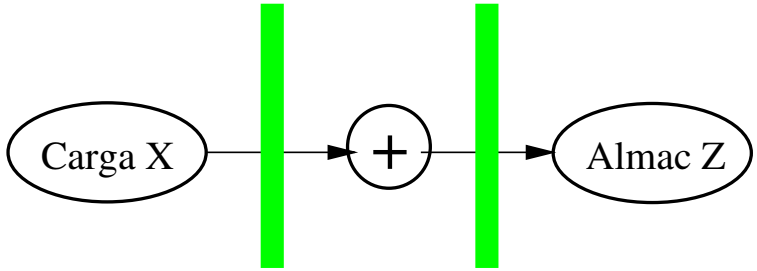
Transformar un bucle con instrucciones dependientes e iteraciones independientes en otro bucle con instrucciones independientes e iteraciones dependientes:



El nombre le viene por que realiza el procesamiento del bucle original simulando el comportamiento de una unidad segmentada:

Idea básica (cont.)

- Los “datos” de la unidad segmentada son las iteraciones.
- Para no sobreescribir los resultados intermedios, el procesamiento se realiza desde la última etapa hacia la primera.




```
...
lt. i      l.d f2,0(r1)   add.d f4,f0,f2   s.d f4,0(r2)
lt. i+1                   l.d f2,8(r1)     add.d f4,f0,f2   s.d f4,8(r2)
lt. i+2                                    l.d f2,16(r1)    add.d f4,f0,f2   s.d f4,16(r2)
...
```

Código con software pipelining

```
start:
    daddi r1, r0, y      ; r1 = dirección de y
    daddi r2, r0, z      ; r2 = dirección de z
    l.d f0, a(r0)       ; f0 = a
    daddi r3, r1, #512 ; 64 elementos son 512 bytes

prepara:
    l.d f2, 0(r1)      ; Lee it. 0
    add.d f4, f0, f2 ; Calcula it. 0
    l.d f2, 8(r1)      ; Lee it. 1
    daddi r1 ,r1, #16
```

Código con software pipelining (cont.)

```
loop:
     s.d f4,0(r2)         ; Escribe it. i
     add.d f4, f0, f2     ; Calcula it. i+1
     l.d f2, 0(r1)        ; Lee it. i+2
     daddi r1, r1, #8
     daddi r2, r2, #8
     dsub r4, r3, r1
     bnez r4, loop

resto:
     s.d f4,0(r2)       ; Escribe it. n-2
     add.d f4, f0, f2   ; Calcula it. n-1
     s.d f4,8(r2)       ; Escribe it. n-1
```