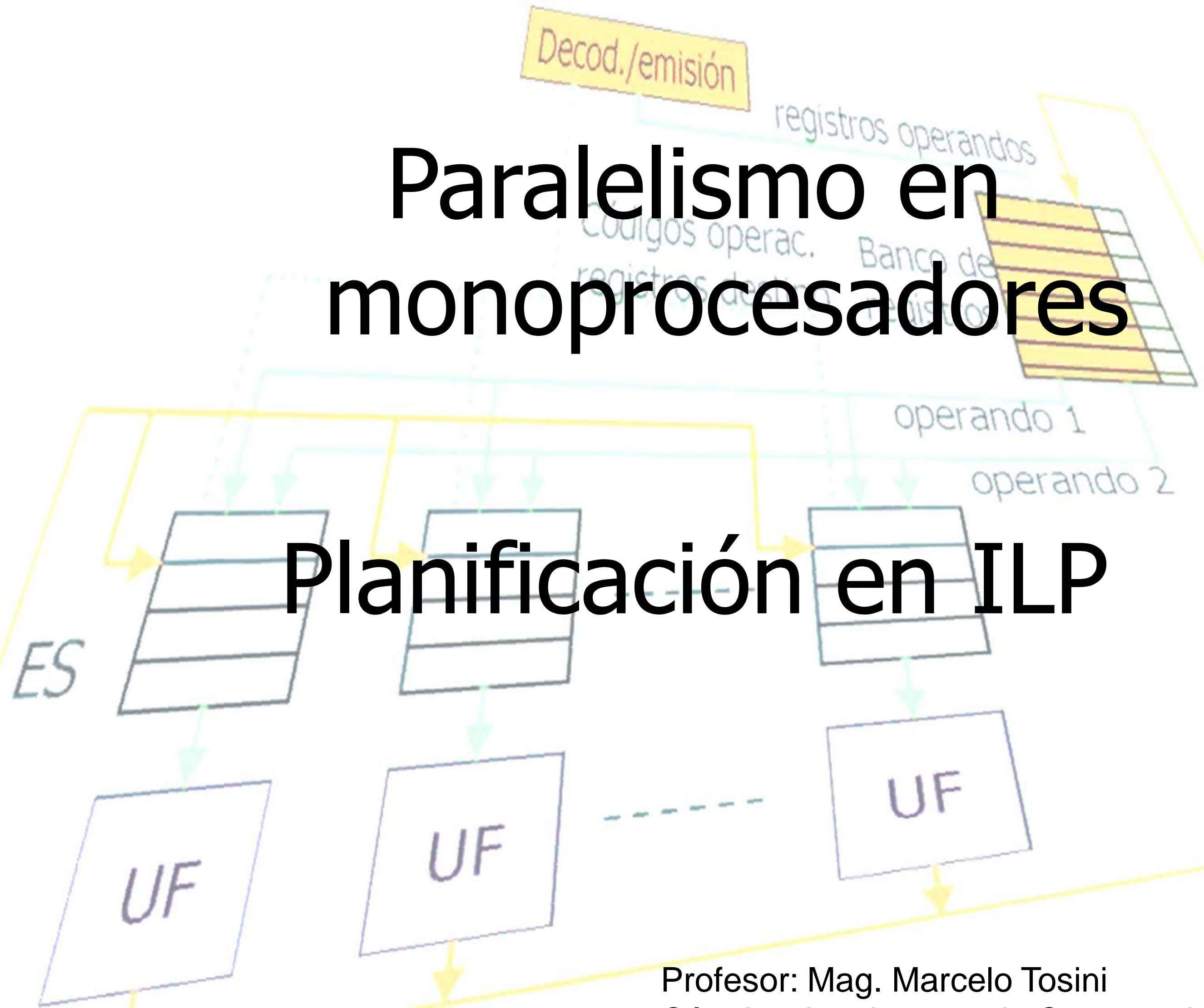
Paralelismo en monoprocesadores

Planificación en ILP



Profesor: Mag. Marcelo Tosini
Cátedra: Arquitectura de Computadoras y técnicas Digitales
Carrera: Ingeniería de Sistemas
Ciclo: 4º año

Conceptos básicos

- **Orden secuencial**: orden de las instrucciones tal como surge de la compilación tradicional. Las instrucciones de máquina respetan el orden computacional impuesto por el código de alto nivel.

- **Consistencia secuencial**: orden de ejecución de las instrucciones que no viola ninguna dependencia entre instrucciones (control, recursos o datos) y, por lo tanto, mantiene la consistencia computacional del algoritmo.

- **Secuencia** (de ejecución): orden en que se ejecutan las instrucciones en un procesador. La secuencia de ejecución puede diferir del orden secuencial siempre y cuando no se pierda la consistencia secuencial.

Planificación

Reordenamiento de las instrucciones de un programa a fin de optimizar el uso del procesador o aumentar la posibilidad de paralelismo.

Quien planifica:

- Compilador (Planificación estática): Reordenamiento de instrucciones en tiempo de compilación para aumentar las posibilidades de ejecución paralela.
- Procesador (Planificación dinámica): Selección de instrucciones para ejecución a fin de ocupar plenamente todas las unidades funcionales disponibles dentro del procesador.

Papel del compilador en la planificación

- **Planificación estática:**

  Proporcionar código paralelo optimizado y libre de dependencias

  Usada en procesadores antiguos sin capacidad de planificar (primeros Superescalares y VLIW)

- **Planificación dinámica:**

  Disparador de rendimiento

  La detección y resolución de dependencias se realiza por hardware

  Procesadores modernos (Escalares y Superescalares)

Tipos de Planificación de instrucciones en ILP

- **Planificación estática con optimización paralela**
  - Realizada por el compilador
  - El procesador recibe código sin dependencias y optimizado para ejecución paralela
  - Procesadores VLIW
- **Planificación dinámica sin optimización paralela estática**
  - Realizada por el procesador
  - El procesador recibe código no optimizado para ejecución paralela
  - Detecta y resuelve las dependencias
  - Primeros procesadores ILP
- **Planificación dinámica con optimización paralela estática**
  - Realizada por el procesador y el compilador
  - El procesador recibe código optimizado para ejecución paralela
  - Detecta y resuelve las dependencias
  - Mayor parte de los procesadores superescalares

Diferencias entre compiladores tradicionales e ILP

- **Ejecución secuencial: no hay dependencias WAW ni WAR**

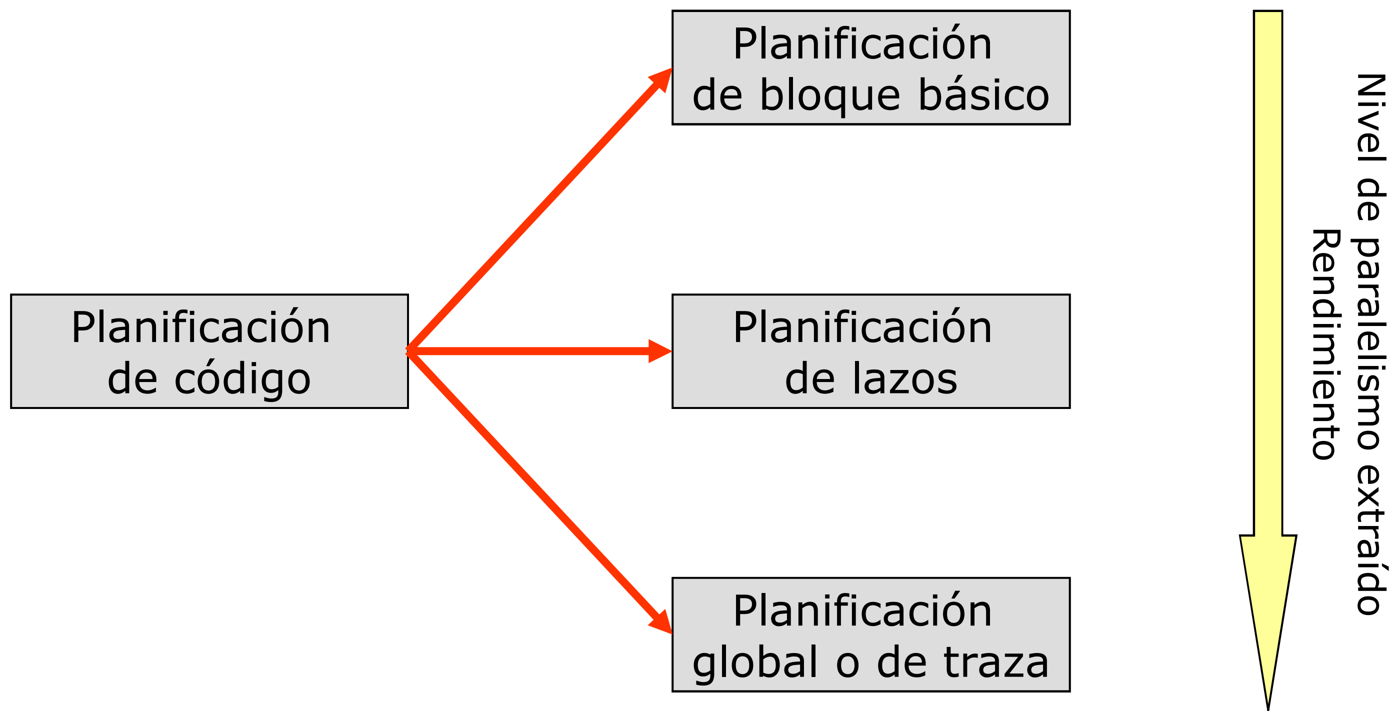
- **Criterios de asignación de registros:**

  - Los compiladores tradicionales maximizan la reutilización de registros
  - Los compiladores ILP evitan la reutilización de registros para disminuir las dependencias WAR y WAW
  - Los procesadores ILP necesitan más registros

- **Dependencias de datos asociadas a referencias a memoria**

  - Sólo se tienen en cuenta para compiladores ILP (adelantamiento de *Loads*)

Niveles de planificación estática



Planificación de bloque básico

- Es la técnica de planificación más simple y menos efectiva
- Sólo las instrucciones dentro de un bloque básico son elegibles para su reordenamiento
- Un bloque básico (BB) es una secuencia de código que no contiene saltos
- Trabaja con listas de items (tareas, **instrucciones**) planificables de las cuales se selecciona el item más adecuado en cada instante (ciclo de ejecución)
- Se debe definir:
    - **Regla de selección:** Permite seleccionar el conjunto de ítems planificables
    - **Regla de elección de la mejor planificación:** Selecciona del conjunto el ítem más apto en un instante dado

Reglas de selección y elección de la mejor planificación

- **regla de selección:** Son instrucciones elegibles
  - No tienen dependencias con instrucciones anteriores
  - Los recursos hardware necesarios están disponibles

- **Elección de la mejor planificación:** Buscar las instrucciones que puedan causar paradas de otras instrucciones

  - Basados en:

    - **Prioridad:** pe.: distancia al final del bloque

    - **Criterios:** Aplicación de un conjunto de criterios en un orden determinado

      - Número de sucesores
      - Longitud de la vía más lenta

Ejemplo: DLX

```
for (I = 1 ; I <= 100 ; I++)
          x[I] = x[I] + s;
```

```
lazo:   LD      f0, 0(r1)         ; carga x[I]
        ADDD    f4, f0, f2        ; suma s
        SD      0(r1), f4         ; guarda x[I]
        SUBI    r1, r1, #8        ; decrementa I
        BNEZ    r1, lazo          ; salta a inicio
```

Latencias:

ALU Pto Flotante	: 3 ciclos
ALU Pto Fijo	: 1 ciclo
Store	: 1 ciclo
Load	: 2 ciclos
Saltos	: 2 ciclos

Planificación en el DLX

Sin planificación

```
lazo:   LD      f0, 0(r1)
        detención
        ADDD    f4, f0, f2
        detención
        detención
        SD      0(r1), f4
        SUBI    r1, r1, #8
        BNEZ    r1, lazo
        detención
```

Tiempo : 9 ciclos

Con salto retardado

```
lazo:   LD      f0, 0(r1)
        detención
        ADDD    f4, f0, f2
        SUBI    r1, r1, #8
        BNEZ    r1, lazo
        SD      0(r1), f4
```

- SUBI y BNEZ suben a los slots de detención del ADDD
- SD baja al slot de detención de BNEZ

Tiempo : 6 ciclos

Planificación arriba-abajo

Las operaciones se planifican en cuanto los operandos están disponibles

La mayoría de las operaciones comienzan tan pronto como es posible

La evaluación del salto se realiza en las primeras etapas del algoritmo

Pasos:

1. Selección del conjunto de instrucciones planificables (CP)
2. Elección de una instrucción del CP (preferentemente vía crítica)
3. Se planifica la instrucción.
4. Se rearma CP
5. Se repite desde 2. hasta que CP se vacíe

Planificación arriba-abajo

Armar un grafo de precedencia de instrucciones

- Cada nodo es una instrucción del bloque básico
- Cada arco indica la dependencia RAW entre los nodos que conecta
- Cada arco se marca con 2 valores:
  - Latencia de la instrucción precedente (Li)
  - Latencia total acumulada (La)
  - Notación : (Li , La)

Parámetro:

- Earliest-Time (ET): para el sucesor de una instrucción planificada.

ET = tiempo actual + latencia de la instrucción planificada

Ejemplo

A = B + C – D
if E-D > 0 then
.....

```
Ld    r1, b(r0)
Ld    r2, c(r0)
Add   r3, r1, r2
Ld    r4, d(r0)
Sub   r5, r3, r4
Sd    r5, a(r0)
Ld    r6, e(r0)
Sub   r8, r6, r4
Cmp   r9, r8, r0
Bc    r9, _then
```

Bloque básico

Instrucciones del bloque básico

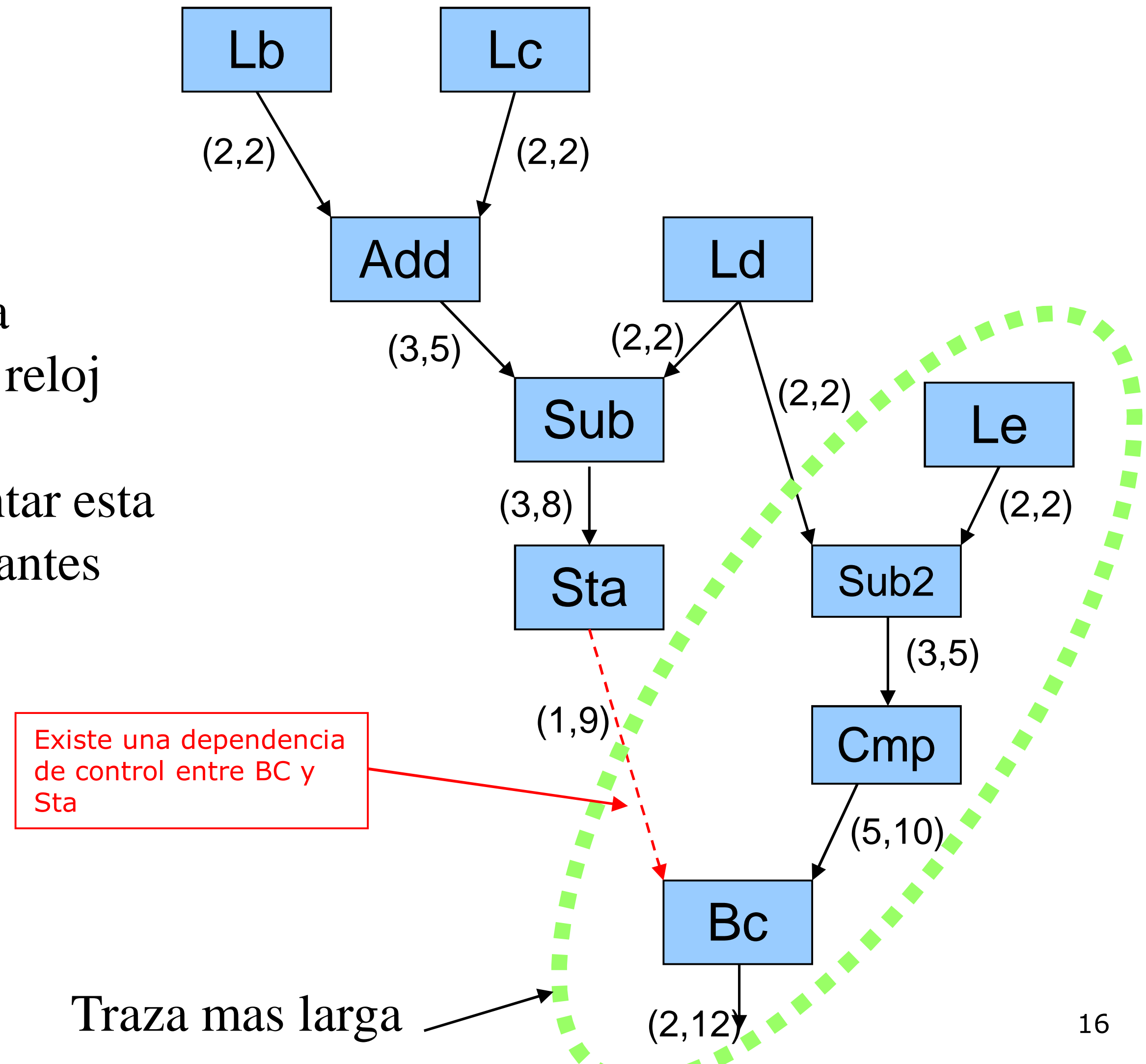
Ejemplo…

Orden	Instrucción		Nemotecnico	Latencia (ns)
1:	Ld	r1, b(r0)	Lb	2
2:	Ld	r2, c(r0)	Lc	2
3:	Add	r3, r1, r2	Add	3
4:	Ld	r4, d(r0)	Ld	2
5:	Sub	r5, r3, r4	Sub	3
6:	Sd	r5, a(r0)	Sta	1
7:	Ld	r6, e(r0)	Le	2
8:	Sub	r8, r6, r4	Sub2	3
9:	Cmp	r9, r8, r0	Cmp	5
10:	Bc	r9, _then	Bc	2

Ejemplo

La traza de mayor latencia es la del salto que tarda 12 ciclos de reloj

La planificación debiera adelantar esta traza para que empiece cuanto antes



Ejemplo

Planificación para tiempo 0

Ciclo	Conjunto de planificables	Elegida
0	Lb , Lc, Ld , Le	Le

Ejemplo

Planificación para tiempo 1

Ciclo	Conjunto de planificables	Elegida
0	Lb , Lc, Ld , Le	Le
1	Lb , Lc, Ld	Ld

Lb
Lc
(2,2)
(2,2)
Add
Ld
(3,5)
(2,2)
(2,2)
Sub
(3,8)
Sta
Sub2
ET = 2
(3,5)
(1,9)
Cmp
(5,10)
Bc
(2,12)

Ejemplo

Planificación para tiempo 2

Ciclo	Conjunto de planificables	Elegida
0	Lb , Lc, Ld , Le	Le
1	Lb , Lc, Ld	Ld
2	Lb , Lc	Lb

Ejemplo

Planificación para tiempo 3

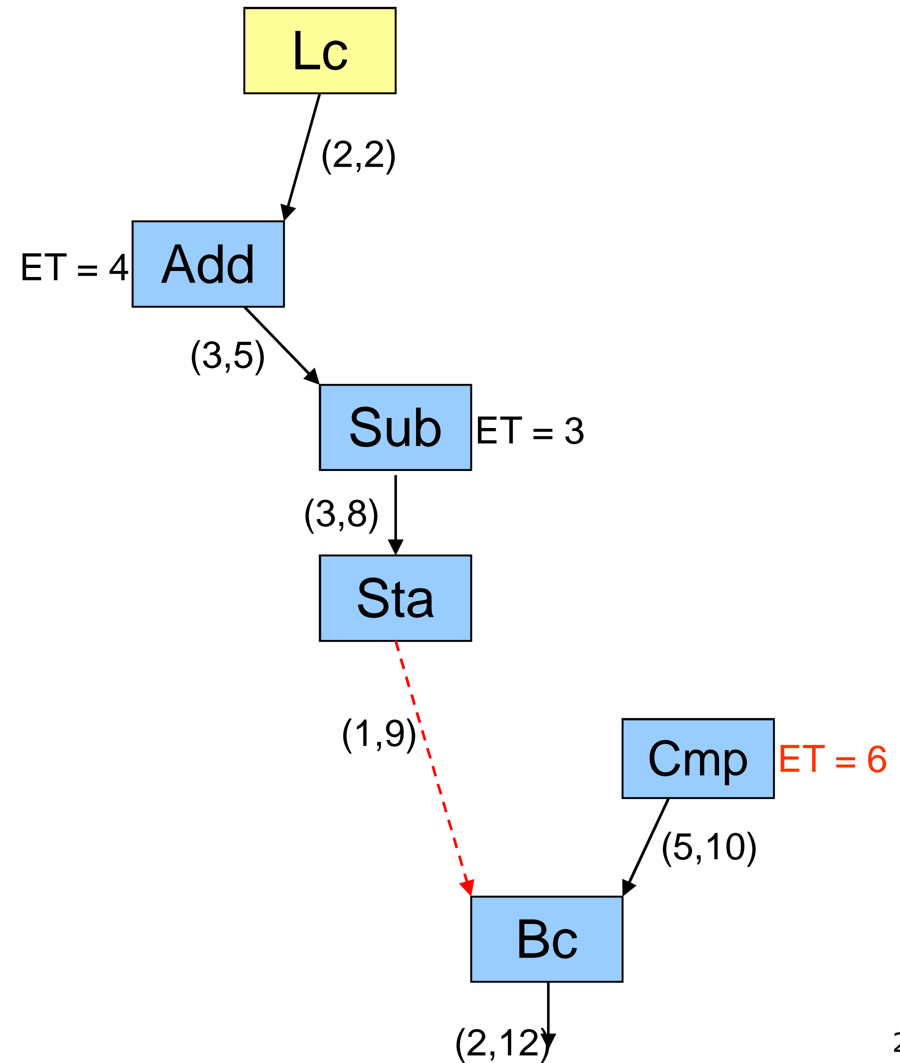
Ciclo	Conjunto de planificables	Elegida
0	Lb , Lc, Ld , Le	Le
1	Lb , Lc, Ld	Ld
2	Lb , Lc	Lb
3	Lc , Sub2	Sub2

Lc
(2,2)
ET = 4 Add
(3,5)
Sub ET = 3
(3,8)
Sta
Sub2 ET = 3
(3,5)
(1,9)
Cmp
(5,10)
Bc
(2,12)

Ejemplo

Planificación para tiempo 4

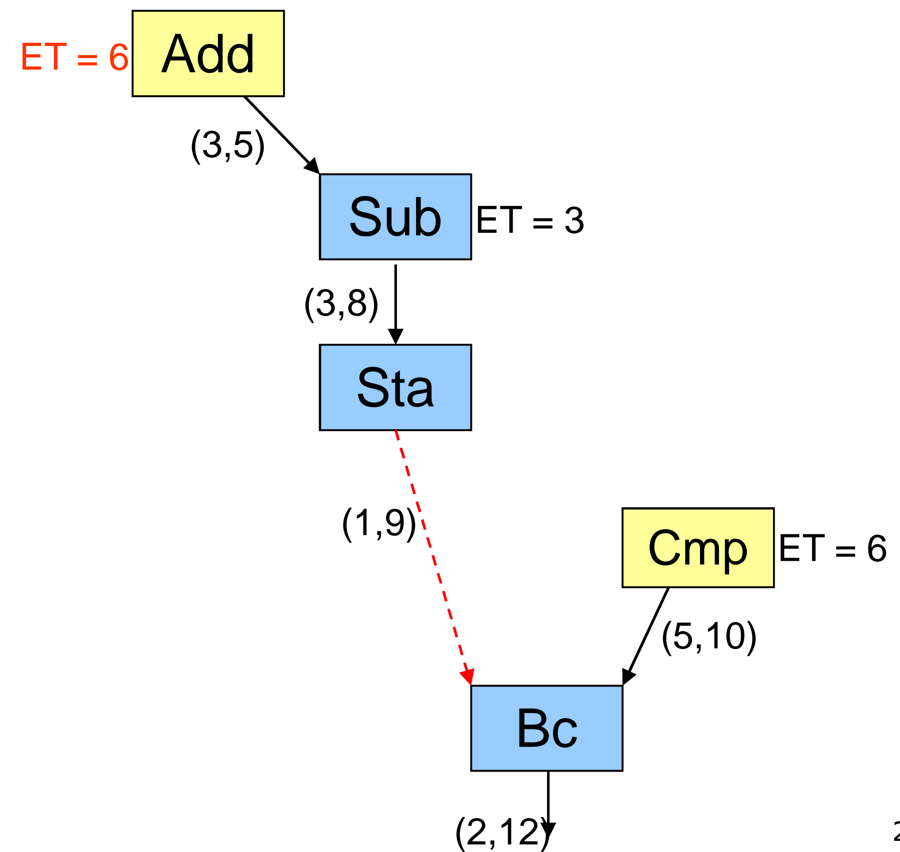
Ciclo	Conjunto de planificables	Elegida
0	Lb , Lc, Ld , Le	Le
1	Lb , Lc, Ld	Ld
2	Lb , Lc	Lb
3	Lc , Sub2	Sub2
4	Lc	Lc



Ejemplo

Planificación para tiempo 6

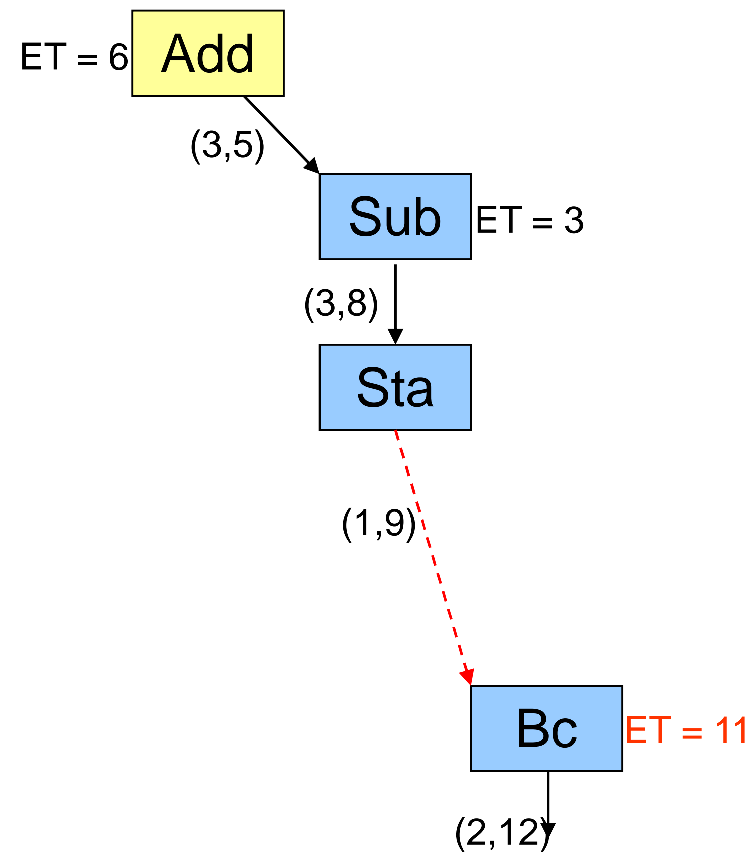
Ciclo	Conjunto de planificables	Elegida
0	Lb , Lc, Ld , Le	Le
1	Lb , Lc, Ld	Ld
2	Lb , Lc	Lb
3	Lc , Sub2	Sub2
4	Lc	Lc
6	Add , Cmp	Cmp



Ejemplo

Planificación para tiempo 7

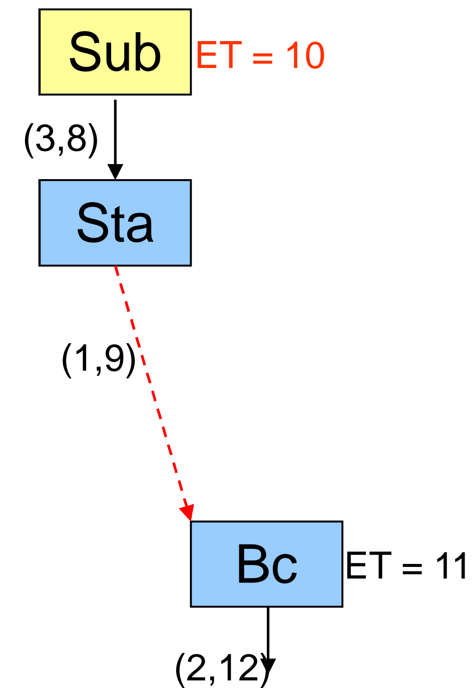
Ciclo	Conjunto de planificables	Elegida
0	Lb , Lc, Ld , Le	Le
1	Lb , Lc, Ld	Ld
2	Lb , Lc	Lb
3	Lc , Sub2	Sub2
4	Lc	Lc
6	Add , Cmp	Cmp
7	Add	Add



Ejemplo

Planificación para tiempo 10

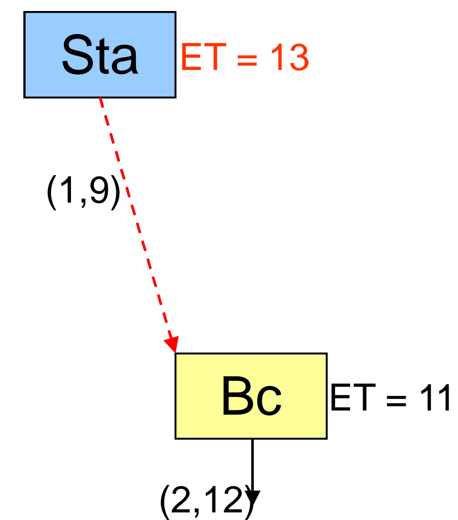
Ciclo	Conjunto de planificables	Elegida
0	Lb , Lc, Ld , Le	Le
1	Lb , Lc, Ld	Ld
2	Lb , Lc	Lb
3	Lc , Sub2	Sub2
4	Lc	Lc
6	Add , Cmp	Cmp
7	Add	Add
10	Sub	Sub



Ejemplo

Planificación para tiempo 11

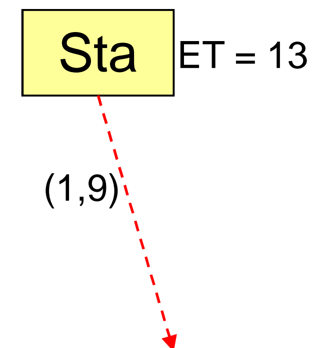
Ciclo	Conjunto de planificables	Elegida
0	Lb , Lc, Ld , Le	Le
1	Lb , Lc, Ld	Ld
2	Lb , Lc	Lb
3	Lc , Sub2	Sub2
4	Lc	Lc
6	Add , Cmp	Cmp
7	Add	Add
10	Sub	Sub
11	Bc	Bc



Ejemplo

Planificación para tiempo 13

Ciclo	Conjunto de planificables	Elegida
0	Lb , Lc, Ld , Le	Le
1	Lb , Lc, Ld	Ld
2	Lb , Lc	Lb
3	Lc , Sub2	Sub2
4	Lc	Lc
6	Add , Cmp	Cmp
7	Add	Add
10	Sub	Sub
11	Bc	Bc
13	Sta	Sta



ET = 13

Ejemplo

Planificación para tiempo 14

Ciclo	Conjunto de planificables	Elegida
0	Lb , Lc, Ld , Le	Le
1	Lb , Lc, Ld	Ld
2	Lb , Lc	Lb
3	Lc , Sub2	Sub2
4	Lc	Lc
6	Add , Cmp	Cmp
7	Add	Add
10	Sub	Sub
11	Bc	Bc
13	Sta	Sta
14	Next Instr.	

ET = 14

Ejemplo

1 :	Ld	r1, b(r0)
2 :	Ld	r2, c(r0)
3 :	Add	r3, r1, r2
4 :	Ld	r4, d(r0)
5 :	Sub	r5, r3, r4
6 :	Sd	r5, a(r0)
7 :	Ld	r6, e(r0)
8 :	Sub	r8, r6, r4
9 :	Cmp	r9, r8, r0
10 :	Bc	r9, _then

Orden original

7 :	Ld	r6, e(r0)
4 :	Ld	r4, d(r0)
1 :	Ld	r1, b(r0)
8 :	Sub	r8, r6, r4
2 :	Ld	r2, c(r0)
9 :	Cmp	r9, r8, r0
3 :	Add	r3, r1, r2
5 :	Sub	r5, r3, r4
10 :	Bc	r9, _then
6 :	Sd	r5, a(r0)

Orden planificado

Ejemplo

Ruta superescalar de 2 vías genéricas (acceso a memoria y ALU)

Código SIN planificar = 19 ciclos de reloj y 60% de utilización

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
ALU 1	Lc		Add			Sub				Sub2			Cmp					Bc	
ALU 2	Lb		Ld		Le				Sta										

Ejemplo

Ruta superescalar de 2 vías genéricas (acceso a memoria y ALU)

Código SIN planificar = 16 ciclos de reloj y 78% de utilización

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ALU 1	Lc		Add			Sub			Sta							
ALU 2	Lb		Ld		Le		Sub2			Cmp					Bc	

Ejemplo

Ruta superescalar de 2 vías genéricas (acceso a memoria y ALU)

Código planificado = 14 ciclos de reloj y 89% de utilización

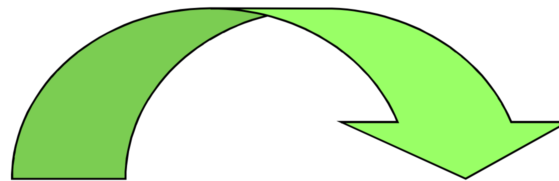
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
ALU 1	Ld		Lb		Lc		Add			Sub			Bc	
ALU 2	Le		Sub2			Cmp					Sta			

Planificación de lazos

- **Los lazos son la principal fuente de paralelismo para procesadores ILP**
    - Se aprovecha la estructura de control regular
    - Utilizado en todos los planificadores de procesadores VLIW y Superescalares
    - Incluido en los planificadores globales
- **Esquemas básicos**
    - Desenrollamiento de lazos (loop unrolling)
    - Segmentación de software (software pipelining)

Desenrollamiento de lazos

- **Objetivo:** Reducir las detenciones por riesgos de control
- Construir un nuevo lazo juntando varias iteraciones del lazo original



```
for (I = 1 ; I <= 1000 ; I++)
        x[I] = x[I] + s;
```

```
lazo:   LD      f0, 0(r1)
        ADDD    f4, f0, f2
        SD      0(r1), f4
        SUBI    r1, r1, #8
        BNEZ    r1, lazo
```

Desenrollamiento de lazos

Latencias:

ALU Pto Flotante	:	3 ciclos
ALU Pto Fijo	:	1 ciclo
Store	:	1 ciclo
Load	:	2 ciclos
Saltos	:	2 ciclos

1000 iteraciones

```
lazo:   LD      f0, 0(r1)
        detención
        ADDD    f4, f0, f2
        detención
        detención
        SD      0(r1), f4
        SUBI    r1, r1, #8
        BNEZ    r1, lazo
        detención
```

T = 1000 * 9 ciclos = **9000 ciclos**

```
lazo:   LD      f0, 0(r1)
        detención
        ADDD    f4, f0, f2
        SUBI    r1, r1, #8
        BNEZ    r1, lazo
        SD      0(r1), f4
```

T = 1000 * 6 ciclos = **6000 ciclos**

Desenrollamiento de lazos

250 iteraciones

```
lazo:   LD      f0, 0(r1)
        ADDD    f4, f0, f2
        SD      0(r1), f4
        LD      f0, -8(r1)
        ADDD    f4, f0, f2
        SD      -8(r1), f4
        LD      f0, -16(r1)
        ADDD    f4, f0, f2
        SD      -16(r1), f4
        LD      f0, -24(r1)
        ADDD    f4, f0, f2
        SD      -24(r1), f4
        SUBI    r1, r1, #32
        BNEZ    r1, lazo
```

- Desenrollamiento 4 veces
- Se eliminan 3 saltos y 3 SUBI
- Hay detenciones debido a LD, ADDD y salto
- 6,75 ciclos por elemento

T = **250** * 27 ciclos = 6750 ciclos

Ciclos por iteración = 7500 / 1000 = 6,75

```
lazo:   LD      f0, 0(r1)
        detención
        ADDD    f4, f0, f2
        detención
        detención
        SD      0(r1), f4
        LD      f0, -8(r1)
        detención
        ADDD    f4, f0, f2
        detención
        detención
        SD      -8(r1), f4
        LD      f0, -16(r1)
        detención
        ADDD    f4, f0, f2
        detención
        detención
        SD      -16(r1), f4
        LD      f0, -24(r1)
        detención
        ADDD    f4, f0, f2
        detención
        detención
        SD      -24(r1), f4
        SUBI    r1, r1, #32
        BNEZ    r1, lazo
        detención
```

Desenrollamiento de lazos

250 iteraciones

```
lazo:   LD      f10, 0(r1)
        LD      f11, -8(r1)
        LD      f12, -16(r1)
        LD      f13, -24(r1)
        ADDD    f20, f10, f2
        ADDD    f21, f11, f2
        ADDD    f22, f12, f2
        ADDD    f23, f13, f2
        SD      0(r1), f20
        SD      -8(r1), f21
        SD      -16(r1), f22
        SUBI    r1, r1, #32
        BNEZ    r1, lazo
        SD      -24(r1), f23
```

Planificación

- Mayor número de registros (registros de mismo nombre se renombrarán)
- Se planifican juntas instrucciones de distintas iteraciones
- Se eliminan todas las detenciones
- 3,5 ciclos por elemento

T = **250** * 14 ciclos = **3500 ciclos**

Ciclos por iteración = 3500 / 1000 = **3,5**

Aceleración = 9000 / 3500 = **2,57**

Segmentación Software

Segmentación de software (Software pipelining).

- La idea básica de la segmentación de software es superponer la ejecución de varias iteraciones diferentes de un loop.

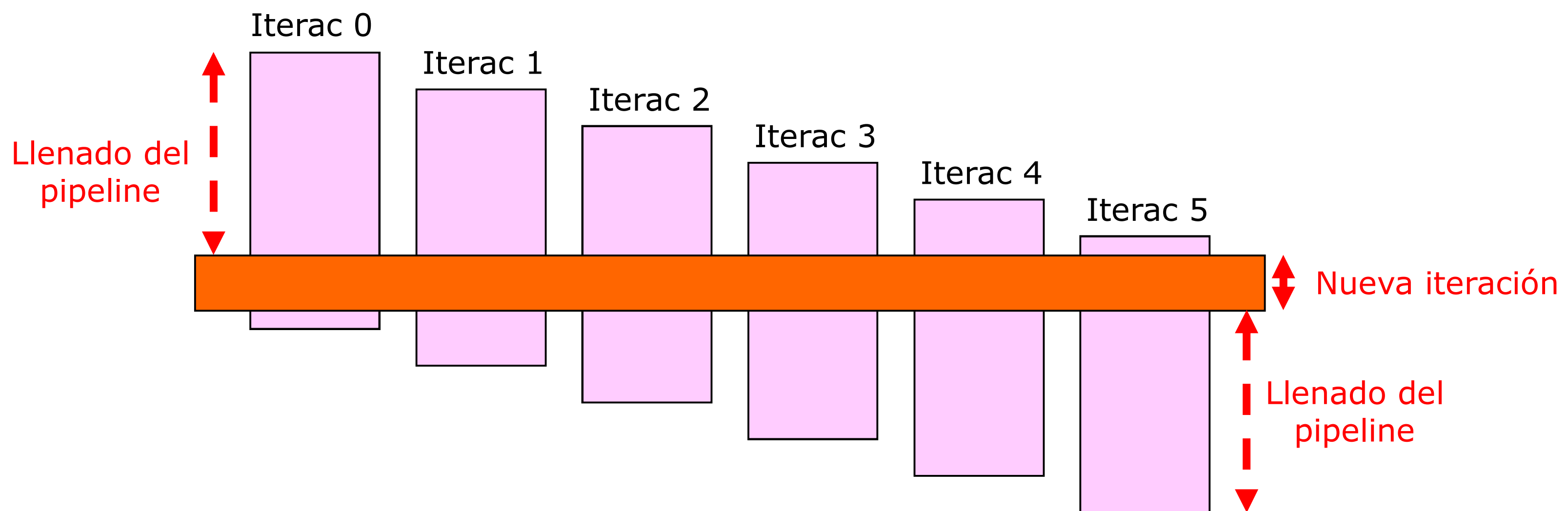
- La traza para el cuerpo de la nueva iteración (*kernel*) es fijada y generada de manera que el número de ciclos entre el comienzo de una iteración y la siguiente sea constante y el mínimo posible.

- El valor mínimo de este intervalo de iniciación de iteraciones, ***MII***, es elegido de manera que una nueva copia del *kernel* se superponga con la anterior cada ***MII***.

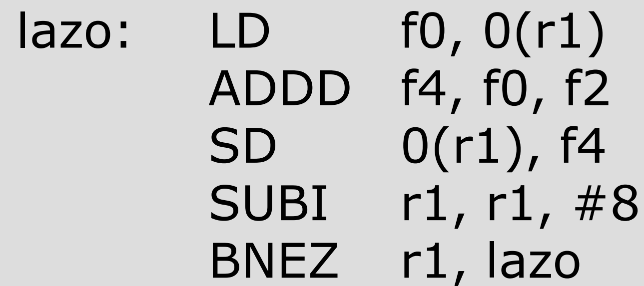
Segmentación Software (idea básica)

Reorganización de los lazos

- Cada iteración del código software segmentado está compuesto de instrucciones de diferentes iteraciones del lazo original
- No se desenrolla el lazo



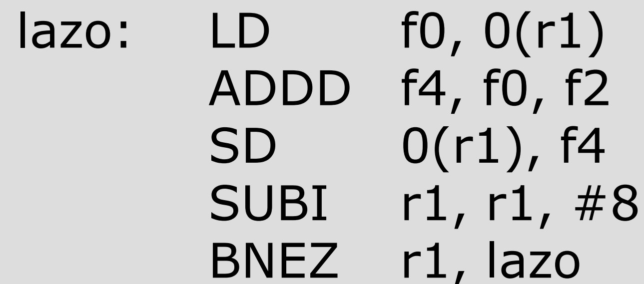
Segmentación Software: Ejemplo DLX



```
lazo:   LD      f0, 0(r1)
        ADDD    f4, f0, f2
        SD      0(r1), f4
        SUBI    r1, r1, #8
        BNEZ    r1, lazo
```

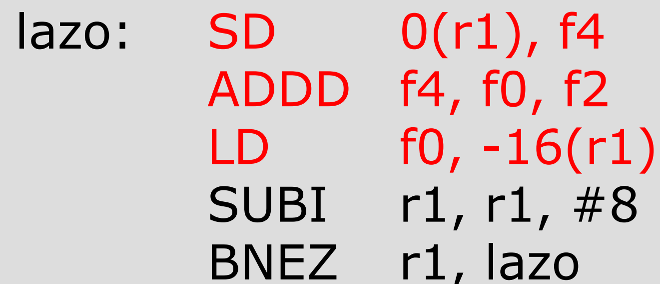
```
iter. i:     LD      f0, 0(r1)
             ADDD    f4, f0, f2
             SD      0(r1), f4
iter i+1:    LD      f0, -8(r1)
             ADDD    f4, f0, f2
             SD      -8(r1), f4
iter. i+2    LD      f0, -16(r1)
             ADDD    f4, f0, f2
             SD      -16(r1), f4
```

Segmentación Software: Ejemplo DLX



```
lazo:   LD      f0, 0(r1)
        ADDD    f4, f0, f2
        SD      0(r1), f4
        SUBI    r1, r1, #8
        BNEZ    r1, lazo
```

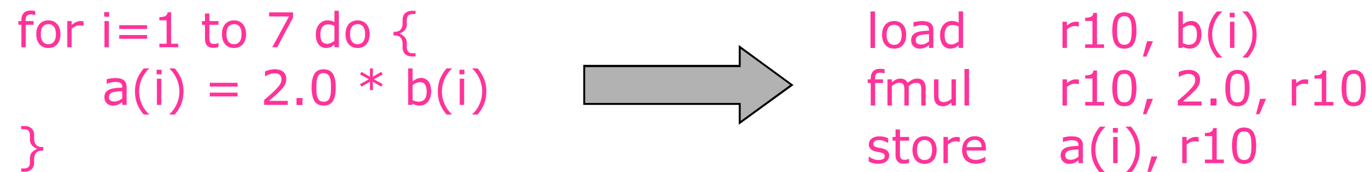
iter. i:	LD	f0, 0(r1)
	ADDD	f4, f0, f2
	SD	**0(r1), f4**
iter i+1:	LD	f0, -8(r1)
	ADDD	**f4, f0, f2**
	SD	-8(r1), f4
iter. i+2	**LD**	**f0, -16(r1)**
	ADDD	f4, f0, f2
	SD	-16(r1), f4



```
lazo:   SD      0(r1), f4
        ADDD    f4, f0, f2
        LD      f0, -16(r1)
        SUBI    r1, r1, #8
        BNEZ    r1, lazo
```

Segmentación Software en procesadores ILP

- Ejecución paralela de instrucciones
- Múltiples unidades funcionales en paralelo



- Suponemos unidades funcionales de FX, FP, y memoria
  - multiplicación: 3 ciclos de latencia
  - load y store: 1 ciclo de latencia

Segmentación software en procesadores ILP

```
load    r100, b(i);
fmul    r100, 2.0, r100;
store   a(i), r100;
```

ciclo	instrucción
c	load r101, b(1);
c+1	fmul r101, 2.0, r101;
c+2	detención
c+3	detención
c+4	store a(i), r100;

Segmentación software en procesadores ILP

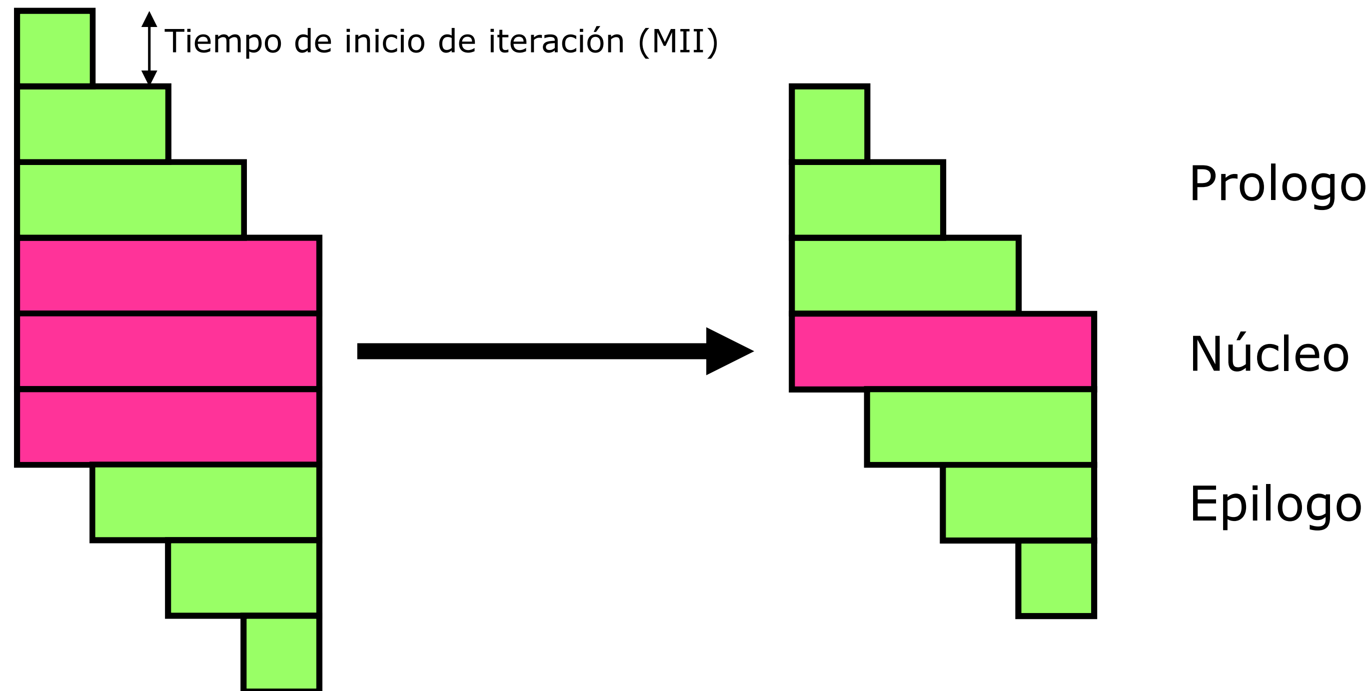
```
load     r100, b(i);
fmul     r100, 2.0, r100;
store    a(i), r100;
```

ciclo	iteración 1	iteración 2
c	load r101, b(1);	
c+1	fmul r101, 2.0, r101;	load r101, b(1);
c+2	detención	fmul r101, 2.0, r101;
c+3	detención	detención
c+4	store a(i), r100;	detención
c+5		store a(i), r100;

Segmentación software en procesadores ILP

Ciclo	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7
c	load						
c+1	fmul	load					
c+2	dec	fmul	load				
c+3	detención	dec	fmul	load			
c+4	store	detención	dec	fmul	load		
c+5		store	detención	dec	fmul	load	
c+6			store	detención	dec	fmul	load
c+7				store	detención	dec	fmul
c+8					store	detención	dec
c+9						store	detención
c+10							store

Segmentación software en procesadores ILP



Segmentación Software
Límites del II

Restricciones de ***MII*** debidas a las dependencias.

Una dependencia de la última instrucción de una iteración a la primera instrucción de la siguiente iteración podría limitar la ejecución a una iteración a la vez.

$$MII(max) <= T_L$$

El máximo paralelismo en un loop con un ciclo en su grafo de dependencias está limitado por el factor de dependencia crítico, T_{crit}, entonces el menor valor de II debido a dependencias de datos es

$$MIIdatos\ (DDG) = MAX_{\forall\ ciclo\ \in\ DDG} \frac{l(ciclo)}{d(ciclo)} = \frac{l(crit)}{d(crit)}$$

Siendo l(p) la suma de todas las latencias de una trayectoria y
d(p) la suma de todas las distancias de una trayectoria

Segmentación Software
Límites del II

Las restricciones de recursos limitan el número de operaciones de un mismo tipo que pueden ejecutarse simultáneamente

En general, si l(v) es el número de ciclos de reloj que un recurso de tipo v es usado en una iteración, y uf el número de unidades del recurso de tipo v disponibles en un procesador, entonces

$$MIIrecursos\ (DDG) = MAX_{\forall\ UF}\left(\left\lceil\frac{\sum_{\forall\ v\ \in\ UF} l(v)}{\#uf}\right\rceil\right)$$

Considerando todas las limitaciones:

$$\mathbf{1 <= max[MIIrecursos\ ,\ MIIdatos] <= MII <= T_L}$$

Segmentación Software. Ejemplo introductorio

```
Doall i=1,100
        a(i) = e(i) + 6
        c(i) = b(i) * d(i)
Enddo
```

Código en alto nivel del loop

Sentencia	Operación	Latencia
ld1:	load e(i)	1
a1:	add e(i) + 6	2
st1	store a(i)	1
ld2:	load b(i)	1
ld3:	load d(i)	1
m1:	mul b(i) * d(i)	3
st2:	store c(i)	1

Código assembler del loop

Segmentación Software.
Ejemplo introductorio

El loop se ejecutará en un procesador con dos puertos de memoria, un sumador y un multiplicador

Tiempo	Port 1 memoria	Port 2 memoria	Sumador	Multiplicador
0	ld1	ld2		
1	ld3		a1	
2				m1
3				
4		st1		
5		st2		

Organigrama para una iteración simple del loop de la figura anterior.

Segmentación Software. Ejemplo introductorio

Determinación de MII

- Dependencias de datos: no hay
- Dependencias de recursos:

$l_{mem} = 5$	$uf_{mem} = 2$	MII(mem) = 3
$l_{add} = 2$	$uf_{add} = 1$	MII(add) = 2
$l_{mul} = 3$	$uf_{mul} = 1$	MII(mul) = 3

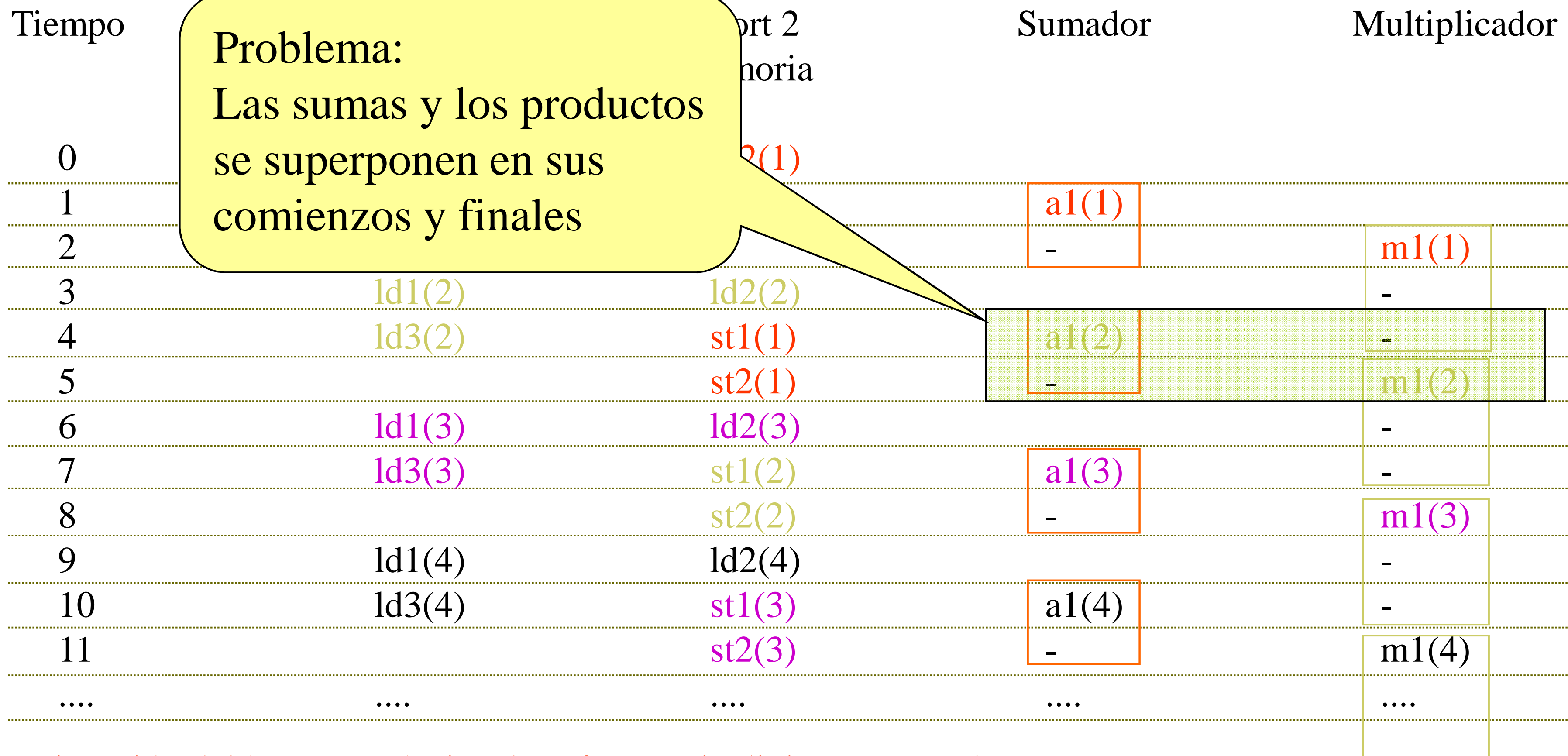
Por lo tanto: **MIIrecursos = 3.**

Segmentación Software. Ejemplo introductorio

Tiempo	Port 1 memoria	Port 2 memoria	Sumador	Multiplicador
0	ld1(1)	ld2(1)		
1	ld3(1)		a1(1)	
2			-	m1(1)
3	ld1(2)	ld2(2)		-
4	ld3(2)	st1(1)	a1(2)	-
5		st2(1)	-	m1(2)
6	ld1(3)	ld2(3)		-
7	ld3(3)	st1(2)	a1(3)	-
8		st2(2)	-	m1(3)
9	ld1(4)	ld2(4)		-
10	ld3(4)	st1(3)	a1(4)	-
11		st2(3)	-	m1(4)
....	....	....	....	....

Ejecución del loop con técnica de software pipelining y MII = 3.

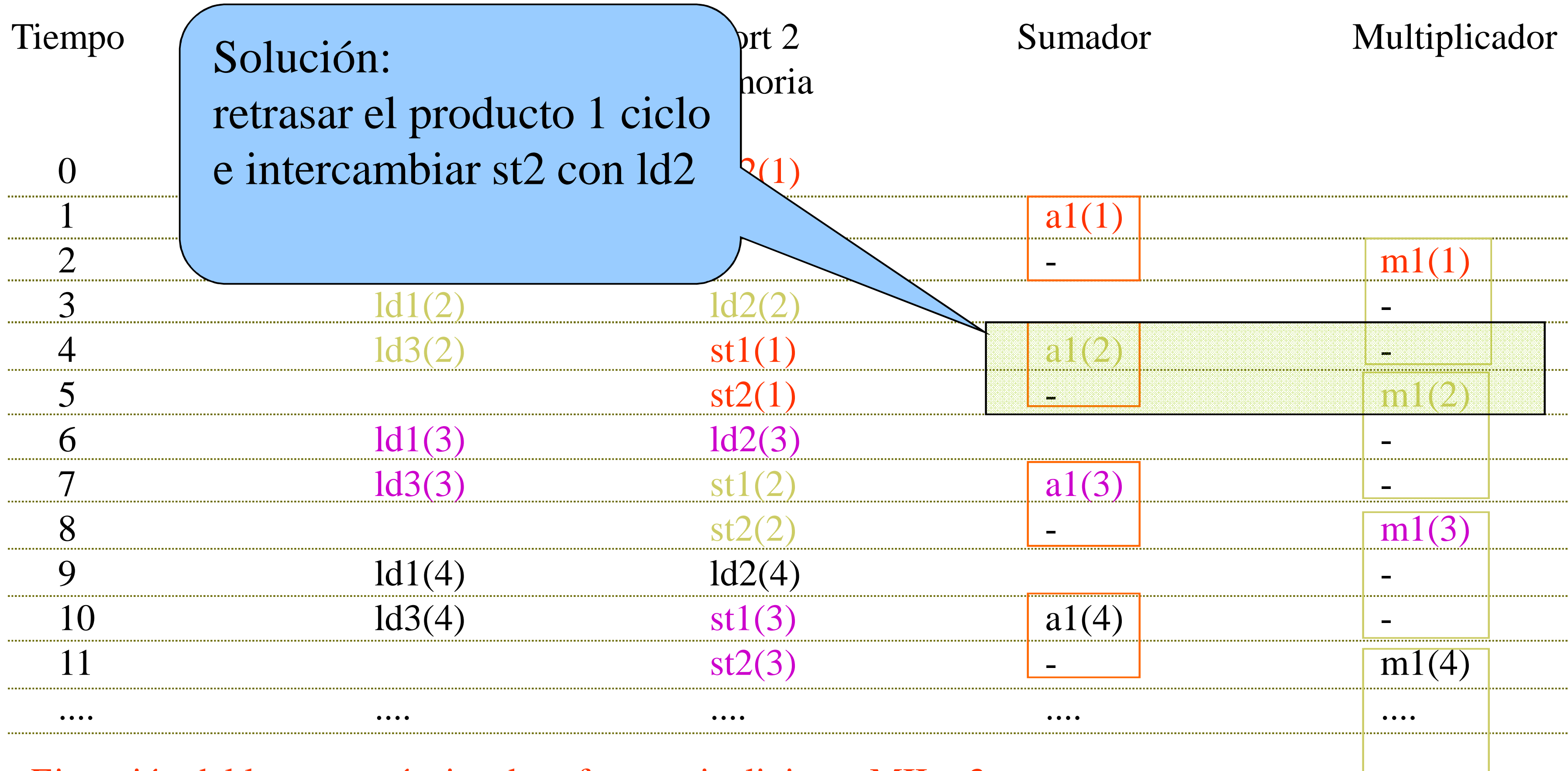
Segmentación Software. Ejemplo introductorio



Tiempo		rt 2 noria	Sumador	Multiplicador
0		2(1)		
1			a1(1)	
2			-	m1(1)
3	ld1(2)	ld2(2)		-
4	ld3(2)	st1(1)	a1(2)	-
5		st2(1)	-	m1(2)
6	ld1(3)	ld2(3)		-
7	ld3(3)	st1(2)	a1(3)	-
8		st2(2)	-	m1(3)
9	ld1(4)	ld2(4)		-
10	ld3(4)	st1(3)	a1(4)	-
11		st2(3)	-	m1(4)
....	....	....	....	....

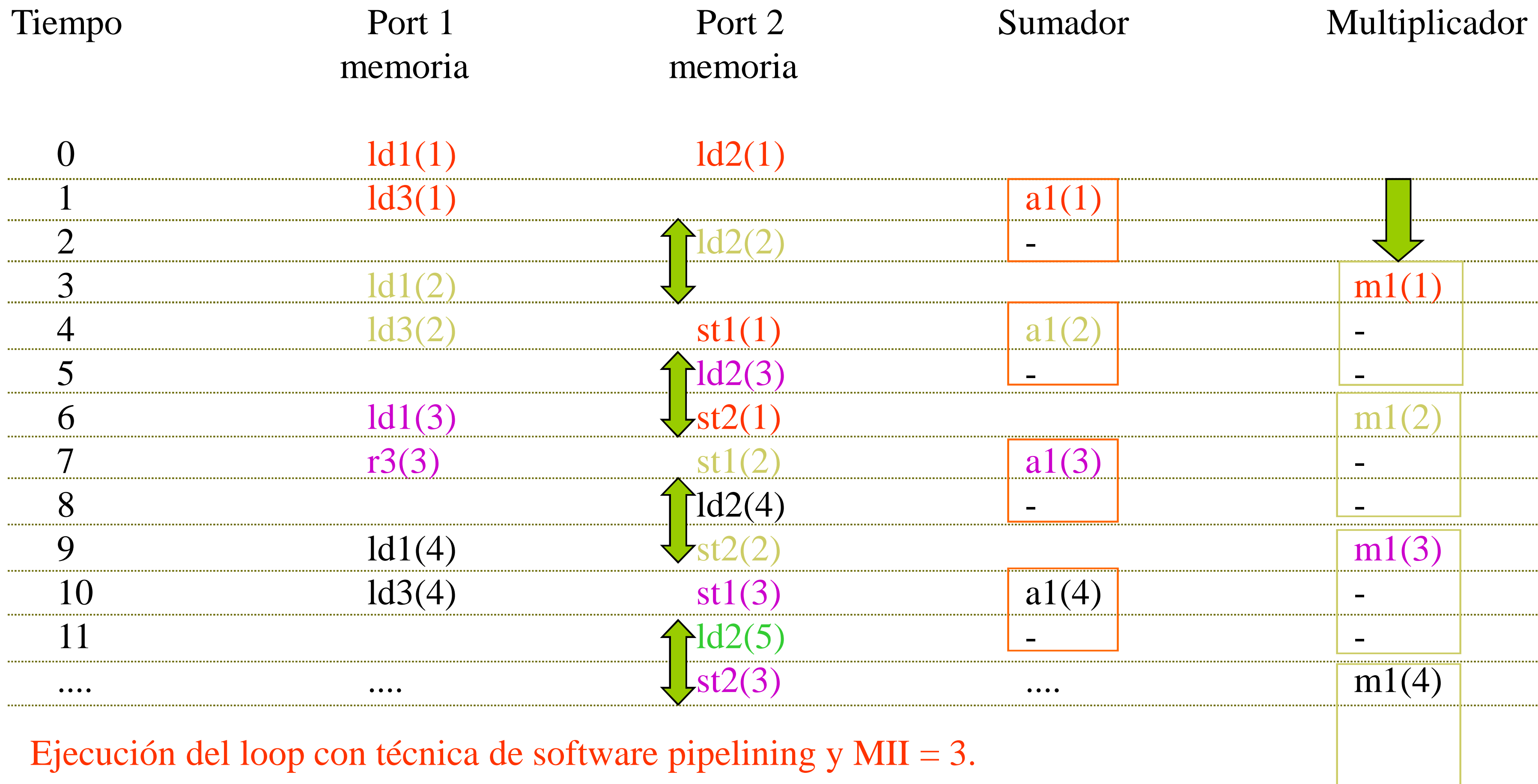
Ejecución del loop con técnica de software pipelining y MII = 3.

Segmentación Software. Ejemplo introductorio



Ejecución del loop con técnica de software pipelining y MII = 3.

Segmentación Software. Ejemplo introductorio



Tiempo	Port 1 memoria	Port 2 memoria	Sumador	Multiplicador
0	ld1(1)	ld2(1)		
1	ld3(1)		a1(1)	
2		ld2(2)	-	
3	ld1(2)			m1(1)
4	ld3(2)	st1(1)	a1(2)	-
5		ld2(3)	-	-
6	ld1(3)	st2(1)		m1(2)
7	r3(3)	st1(2)	a1(3)	-
8		ld2(4)	-	-
9	ld1(4)	st2(2)		m1(3)
10	ld3(4)	st1(3)	a1(4)	-
11		ld2(5)	-	-
....	....	st2(3)	....	m1(4)

Ejecución del loop con técnica de software pipelining y MII = 3.

Segmentación Software. Ejemplo introductorio

Tiempo	Port 1 memoria	Port 2 memoria	Sumador	Multiplicador
0	ld1(1)	ld2(1)		
1	ld3(1)		a1(1)	
2		ld2(2)	-	
3	ld1(2)			m1(1)
4	ld3(2)	st1(1)	a1(2)	-
5		ld2(3)	-	-
6	ld1(3)	st2(1)		m1(2)
7	ld3(3)	st1(2)	a1(3)	-
8		ld2(4)	-	-
9	ld1(4)	st2(2)		m1(3)
10	ld3(4)	st1(3)	a1(4)	-
11		ld2(5)	-	-
....	....	st2(3)	....	m1(4)

nuevo cuerpo del lazo

Ejecución del loop con técnica de software pipelining y MII = 3.

Segmentación Software. Ejemplo introductorio

```
        r1 = 1
lazo:   ld1
        a1
        st1
        ld2
        ld3
        m1
        st2
        inc r1
        r1<101?; lazo
```

Original

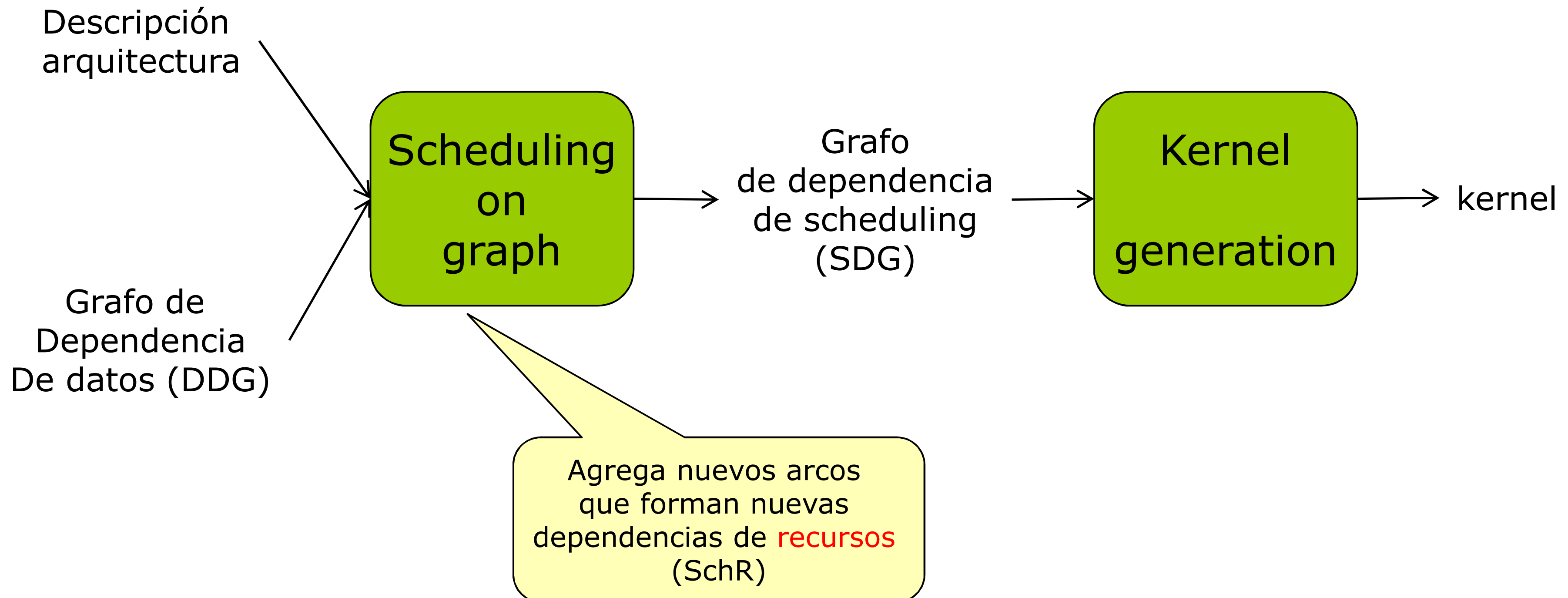
```
        ld1(1)     ; ld2(1)
        ld3(1)     ; a1(1)
        ld2(2)
        ld1(2)     ; m1(1)
        ld3(2)     ; st1(1)    ; a1(2)
        ld2(3)

        r1 = 0
lazo:   ld1(r1+3)  ; st2(r1+1)  ; m1(r1+2)
        ld3(r1+3)  ; st1(r1+2)  ; a1(r1+3)
        ld2(r1+4)
        inc r1
        r1<97?; lazo

        ld1(99)    ; st2(97)
        ld3(99)    ; st1(98)    ; a1(99)
        ld2(100)
        lt2(98)    ; m1(99)
```

Segmentado

Segmentación Software. Método GTSP (Graph Traverse Software Pipelining)

2 pasos:

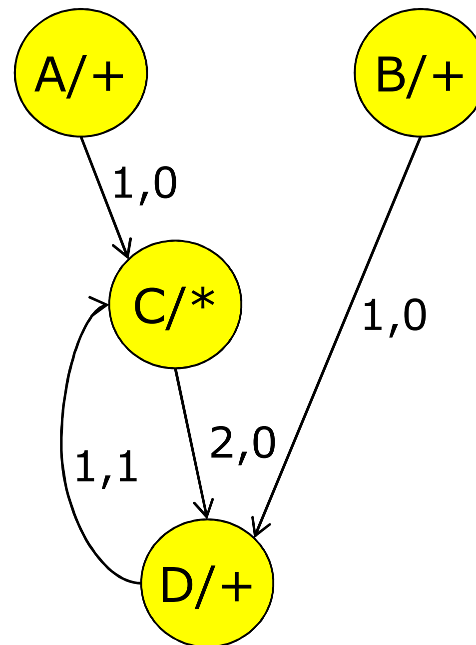
- Generación de scheduling de recursos
- Generación del kernel



Segmentación Software. Ejemplo método GTSP (Graph Traverse Software Pipelining)

```
Do I =
   a = r1 + r2
   b = r3 + r4
   c = a * b
   d = c + b
enddo
```

Código



DDG

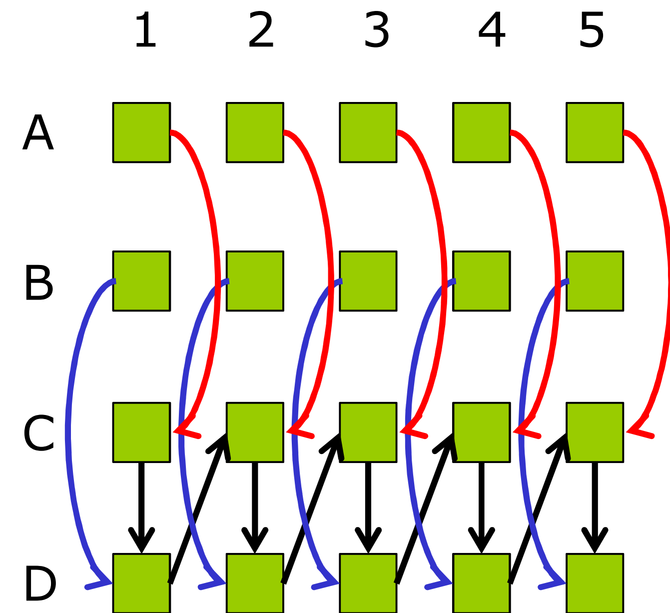
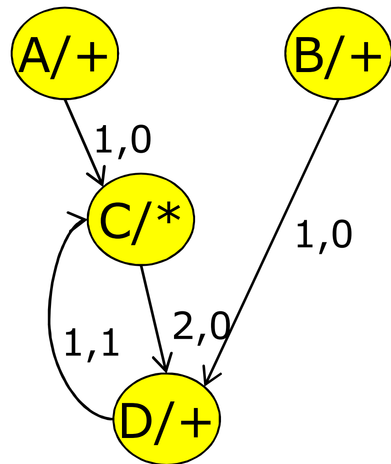


Gráfico de iteración espacial

Segmentación Software. Ejemplo método GTSP (Graph Traverse Software Pipelining)

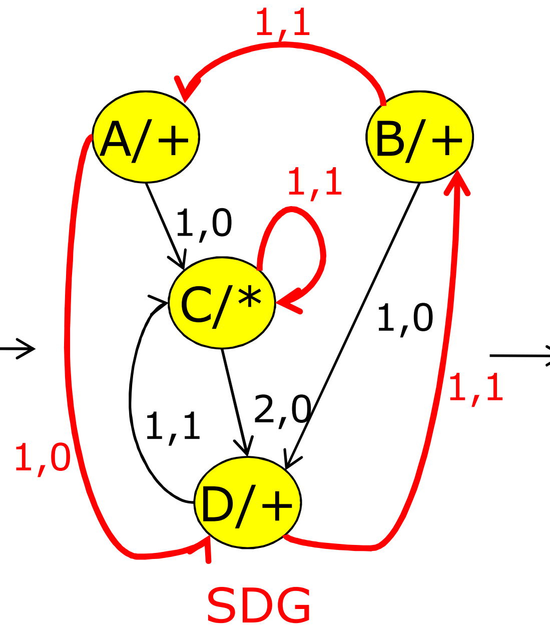
FU	#units
ADD	2
MUL	1 (piped)



DDG

MIIdatos = 3/1
MIIrecursos = 3/2

Schedule on graph



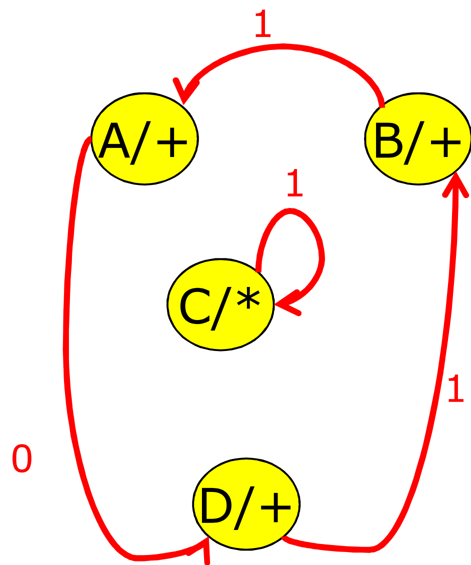
MIIdatos = 3/1

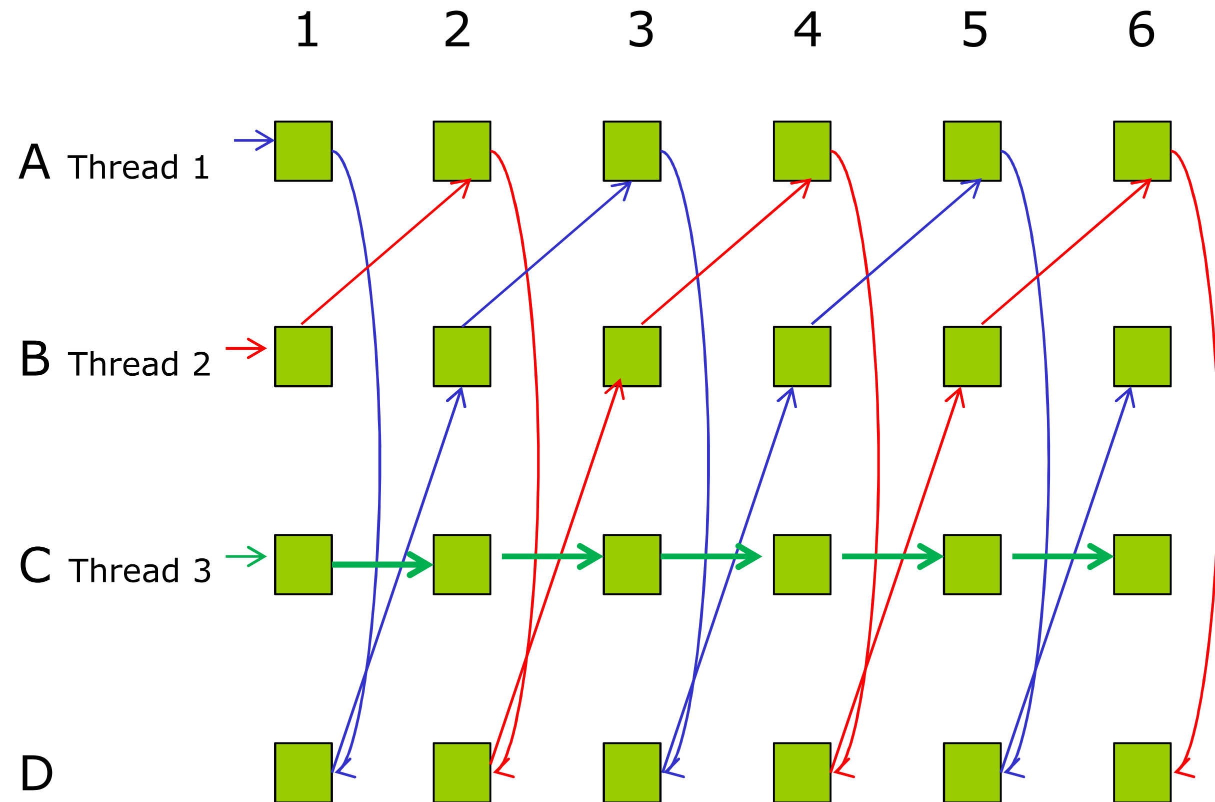
Schedule on graph

ADD	ADD	MUL
A_i	D_{i-1}	
	B_i	C_i

II = 3

Segmentación Software. Ejemplo método GTSP (Graph Traverse Software Pipelining)





2 recurrencias de recursos distintas para el bucle $A_i - D_i - B_{i+1} - A_{i+2}$

Segmentación Software.
Ejemplo método GTSP (Graph Traverse Software Pipelining)

- **Generación de código:** Generación de threads (verticales) a medida que se atraviesan las recurrencias de scheduling del SDG y asignación de las operaciones de esos threads en sus slots correspondientes en el **kernel.**
- Todas las recurrencias de recursos (SchR) deben correr a la misma velocidad.
- El cociente latencia/distancia de un SchR representa la máxima velocidad a la que puede correr.
- Para disminuir la velocidad de un SchR: **+**distancia ó **−**latencia.
- **Aumento de distancia:** agragado de nodos artificiales o **empty nodes**

$$MII(SDG) = MIIrecursos(SchR) = \frac{l(SchR) + ne}{d(SchR)}$$

=>

$$ne = d(SchR) * MII(SDG) - l(SchR)$$

Con ne = número de empty nodes

Segmentación Software.
Ejemplo método GTSP (Graph Traverse Software Pipelining)

Generación del kernel (II entero)

- Para cada SchR se debe calcular:
  - Nodo de cabecera
  - Tiempo de inicio (begin_time)
  - Iteración de inicio (begin_instance)
  - Número de empty_nodes a agragar

- Se define un nodo de referencia (sin dependencia de datos) y se setea su tiempo = 0 y su instancia = 0

PE: A0 en el ciclo 0

- Para los SchR (que no contienen al nodo de referencia) se establece
  - Nodo de cabecera
  - Trayectoria mas corta desde el nodo de referencia al nodo de cabecera

Entonces, begin-time = l(trayectoria) y instance_time = d(trayectoria)

- Los threads sucesivos de cada tipo de recurso (SchR) se inician con II ciclos y 1 iteración de diferencia respecto al previo

A_0 arranca en el ciclo 0 en el ADDER 1 y
A_1 arranca en el ciclo 3 en el ADDER 2

Segmentación Software.
Ejemplo método GTSP (Graph Traverse Software Pipelining)

Generación del kernel (II entero)

EJEMPLO: **MII = 3**

Para el SCHR de suma:

Nodo de referencia A_0
Begin_instance = 0
Begin_time = 0
ne = 2 * 3 – 3 = 3

- Como hay 2 ADDERS puede haber 2 instancias corriendo en paralelo, entonces

A_1 empieza en el segundo sumador
Begin_time = 3
Begin_instance = 1

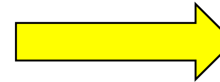
Para el SchR de multiplicación:

Nodo de cabecera = C
Begin_instance = d(path:A->C) = 0
Begin_time = l(path:A->C) = 1
ne = 1 * 3 – 1 = 2

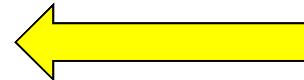
Segmentación Software. Ejemplo método GTSP (Graph Traverse Software Pipelining)

Generación del kernel (II entero)

SchR	MIIdatos	ne	SchR + ne
ADD	3/2	3	A e e D B e A
MUL	3/1	2	C e e C



Thread 1 ADD	Thread 2 ADD	Thread 3 MUL
A_0		
e		C_0
e		e
D_0	A_1	e
B_1	e	C_1
e	e	e
A_2	D_1	e
e	B_2	C_2
e	e	e
D_2	A_3	e
B_3	e	C_3



A_2	D_1	e
e	B_2	C_2
e	e	e

KERNEL
1 iteración!!

Segmentación Software.
Ejemplo método GTSP (Graph Traverse Software Pipelining)

Generación del kernel (II fraccionario)

- En caso de II = N/D
- El kernel final tiene N líneas de instrucciones abarcando D iteraciones* del bucle original.
- Ne puede ser racional de la forma X/Y
  - X = número total de empty_nodes a insertar en el SchR.
  - Y = número de iteraciones que atraviesa el SchR
- Delay puede ser racional de la forma M/P, entonces
  - delay = $\lceil M/P \rceil$

* Instrucciones que completarian D iteraciones

Segmentación Software.
Ejemplo método GTSP (Graph Traverse Software Pipelining)

Generación del kernel (II fraccionario)

EJEMPLO: **MII = 3/2**

Para el SCHR de suma:

Nodo de referencia A_0
Begin_instance = 0
Begin_time = 0
ne = 2 * 3/2 – 3 = 0

- Como hay 2 ADDERS puede haber 2 instancias corriendo en paralelo, entonces

A_1 empieza en el segundo sumador
Begin_time = 3
Begin_instance = 1

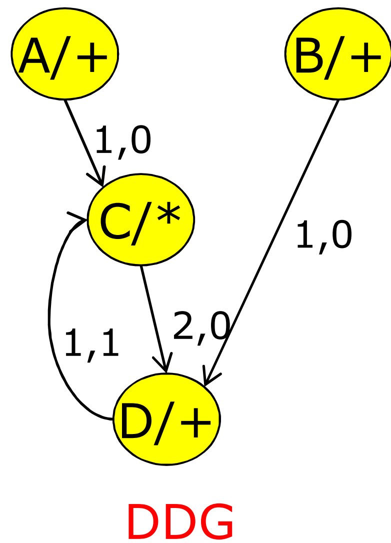
Para el SchR de multiplicación:

Nodo de cabecera = C
Begin_instance = d(path:A->C) = 0
Begin_time = l(path:A->C) = 1
ne = 1 * 3/2 – 1 = 1/2

Segmentación Software.
Ejemplo método GTSP (Graph Traverse Software Pipelining)

Generación del kernel (II fraccionario)

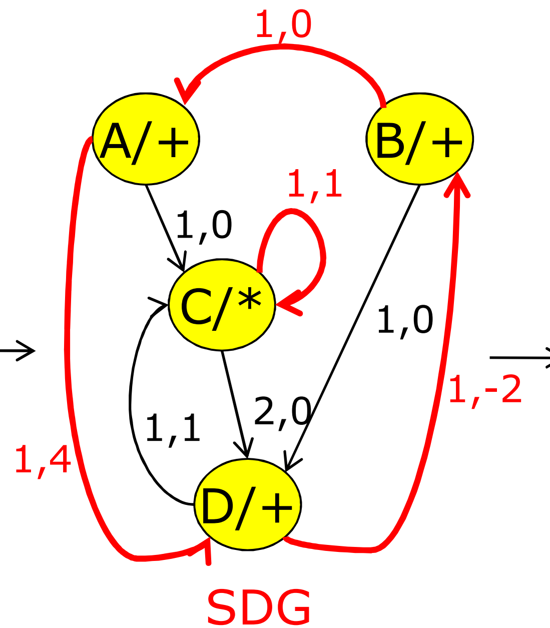
EJEMPLO: **MII = 3/2**



DDG

MIIdatos = 3/1
MIIrecursos = 3/2

Schedule on graph



SDG

MIIdatos = 3/1

Schedule on graph

ADD	ADD	MUL
A_i	B_{i-1}	C_{i-1}
B_i	D_{i-3}	C_i
C_{i-2}	A_{i+1}	

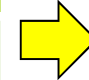
II = 3

Segmentación Software. Ejemplo método GTSP (Graph Traverse Software Pipelining)

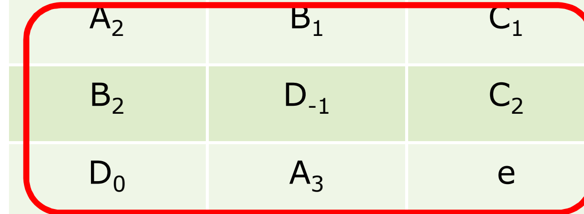
Generación del kernel (II fraccionario)

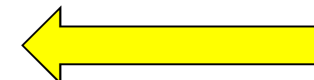
EJEMPLO: **MII = 3/2**

SchR	MIIdatos	ne	SchR + ne
ADD	3/2	0	A B D A
MUL	3/1	1/2	C e/2 C = C C e C C



Thread 1 ADD	Thread 2 ADD	Thread 3 MUL
A_0		
B_0		C_0
D_{-2}	A_1	e
A_2	B_1	C_1
B_2	D_{-1}	C_2
D_0	A_3	e
A_4	B_3	C_3
B_4	D_1	C_4
D_2	A_5	e





A_2	**B_1**	**C_1**
B_2	**D_{-1}**	**C_2**
D_0	A_3	

KERNEL
2 iteraciones!!

Segmentación Software. Ejemplo método GTSP (Graph Traverse Software Pipelining)

Algoritmo heuristico de scheduling

- Objetivo: Generación del SchR para cada tipo de recurso.
- Cada paso del algoritmo agrega un arco el SDG.
- Agrega operaciones comenzando por las más críticas
- 3 condiciones:
  - a) Cada Schr debe conectar todos los nodos de un tipo dado.
  - b) Distancia del Schr debe ser <= número de recursos de un tipo dado.
  - c) SchR debe preservar MII del DDG (condición no obligatoria si contradice a) ó b)).

Segmentación Software.
Ejemplo método GTSP (Graph Traverse Software Pipelining)

Algoritmo heuristico de scheduling

```
SDG = DDG
{NAN} = {V} of SDG
{AN} = ∅
For each uft in {FU_TYPE}
        distance = number_of_FU(uft)
        SchR(uft) = create_empty_rec(distance)
/** proceso iterativo **/
While NAN ≠ ∅
        V = choose_node(NAN)                       /*Heurística de orden*/
        Include_node_in_recurrence(V, SDG, II)     /*Heurística de posición*/
        if (failed)
                increment(II)
                restart_whole_proccess
        {NAN} = {NAN} – {V}
        {AN} = {AN} + {V}
```

Segmentación Software. Ejemplo método GTSP (Graph Traverse Software Pipelining)

Heurística de orden

- Se elige un nodo de NAN que opertenezca a mas recurrencias y a las más restrictivas mediante una matriz de mínimas distancias λ(u,v) para todos los nodos.

$$\lambda(u,v) = MAX_{\forall\, p:u \to v} \left(\frac{D * l(p) - N * d(p)}{N}\right); \qquad \text{dado un MII} = N/D$$

- Si ambos valores estan definidos, λ(u,v) y λ(v,u) entonces los nodos u y v pertenecen a la misma recurrencia de datos.

- Si u y v pertenecen a la misma recurrencia, entonces
  - Si λ(u,v) + λ(v,u) = 0 entonces la recurrencia es la mas restrictiva.
  - Si λ(u,v) + λ(v,u) > 0 entonces la recurrencia no es restrictiva.

Segmentación Software.
Ejemplo método GTSP (Graph Traverse Software Pipelining)

Heurística de alocación

- Una vez elegido un nodo se lo inserta en el SchR apropiado y en el lugar mas apropiado.
- El candidato se obtiene comparando los slack de todos los ejes del SchR.

  $Slack(s,t) = d(s,t) - \lambda(s,t);$ con s y t nodos de la recurrencia

- Si slack(s,t) = 0 el path es de los mas restrictivos, o sea, su cociente l(p)/d(p) = MII
- Si slack(s,t) > 0 se puede insertar el nodo v siempre que

$$Slack(s,t) \geq \lambda(v,t)$$

Evaluación de la técnica

Puesto que una nueva iteración de un loop es iniciada cada MII ciclos con esta técnica, el tiempo total de ejecución será de N * MII, más el tiempo de *arranque* necesario para alcanzar la condición de equilibrio (software pipeline fill time - t_{fill})

Máxima aceleración de un loop con algún ciclo en su grafo de dependencias:

$$S_{max}(\text{soft-pipe}) = (N * T_L) / (N*MII + t_{fill}) \text{ aprox.} = T_L / MII$$

Desventajas de los métodos

- En tiempo de compilación no es fácil determinar las direcciones de carga y almacenamiento de una operación para inferir su independencia.

Ejemplo: si B(i) es 5,

S1 : A(B(i)) = 23.6;
S2 : X = A(5) * C;

Hay conflictos al paralelizar S1 y S2

- Incertidumbre sobre las verdaderas restricciones de tiempo de determinados recursos del sistema

Las referencias a memoria pueden ocasionar fallos de cache o memoria virtual entonces no se puede saber a priori el tiempo necesario para acceder a un dato

Planificación global o de traza

- Los programas de propósito general (Sistemas operativos, Compiladores, Aplicaciones varias, etc.) se caracterizan por:
    - Bloques básicos pequeños
    - saltos de comportamiento irregular
- Es difícil encontrar suficientes instrucciones independientes para ocupar todas las unidades funcionales

La planificación global permite:

- Ir mas allá de los bloques básicos
- Planificar entre iteraciones sucesivas de los lazos

Planificación global o de traza

- Diferencias con la planificación de bloque básico
  - **Bloque básico:** extrae paralelismo dentro de los límites del bloque
  - **Global:** Intenta encontrar el mayor número posible de instrucciones independientes en el grafo de dependencias del programa

Mueve instrucciones sin tener en cuenta los límites de los bloques

- **Objetivo:** Mantener ocupado el mayor número de unidades funcionales en cada ciclo de reloj

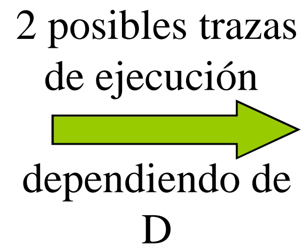
- El límite para mover instrucciones son las dependencias de datos verdaderas

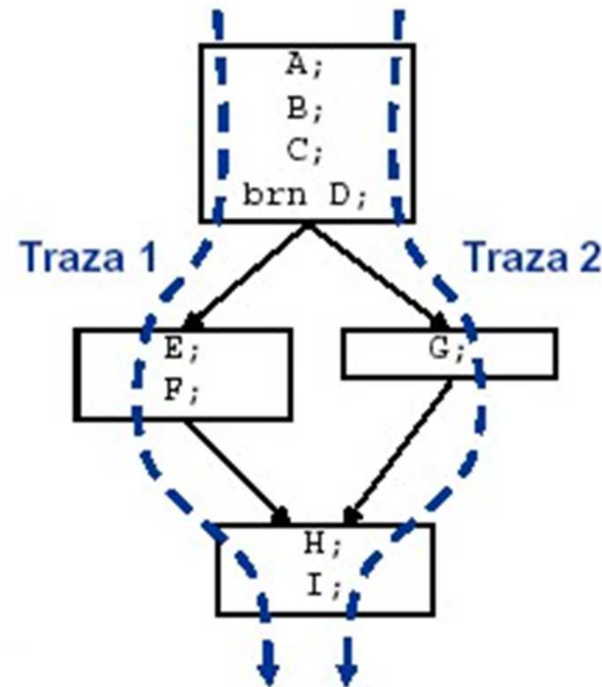
Planificación global o de traza

Traza: Camino posible a través de una sección de código.

La secuencia de instrucciones en la traza depende del resultado de los saltos en la secuencia.

```
A;
B;
C;
If (D) {
    E;
    F;
}
else {
    G;
}
H;
I;
```

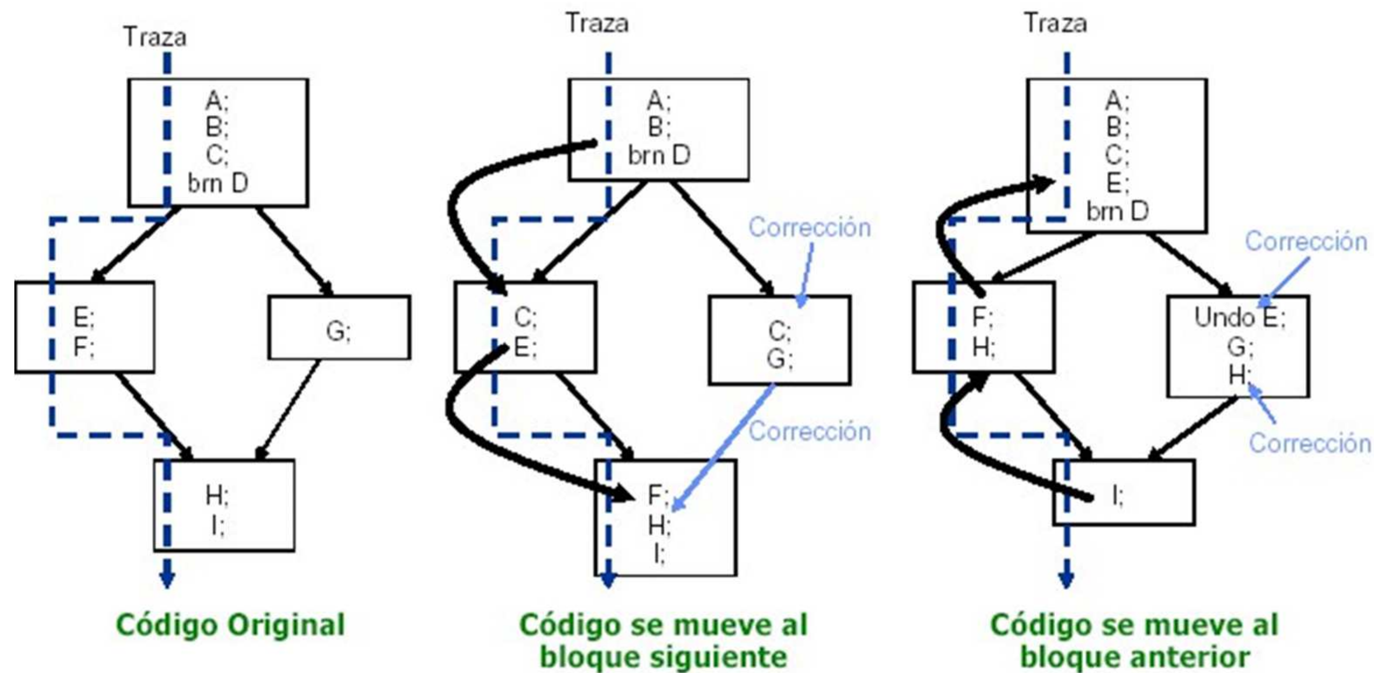




La planificación de trazas reduce el tiempo de ejecución de algunas trazas a expensas de otras, de modo que, conviene optimizar las trazas mas usadas (usando predicción de saltos)

Correcciones y compensación de código

Es necesario agregar código de compensación que deshaga los efectos no esperados del movimiento de instrucciones entre bloques básicos para el caso de que la predicción falle.



Metodología de planificación de traza

- Considera un procedimiento (función) a la vez
- Selecciona trazas para planificar usando predicciones de salto
- Cada vez que se planifica una traza se añade código de compensación
- En siguientes planificaciones se trabaja con los resultados de pasos anteriores
- Problemas
  - Excesivo tiempo para planificar trazas: n saltos -> 2^n trazas
  - Problema de corrección en el código planificado:

```
a[i] = a[i]+1;
If (b!=NULL)
    b->sum = b->sum + a[i]
```

Si la instrucción en el cuerpo del if pasa a ejecutarse en primer lugar habrá problemas

- Para código de propósito general es difícil aplicar planificación de traza