Planificación estática

- Planificación de bloques básicos
- Planificación de bucles
- Planificación global

Planificación de bloques básicos..

- Técnica sencilla. Eficiencia limitada por
  - El pequeño tamaño de los bloques básicos
    - ✓ Tamaño medio (en MIPS 3000):
      - 8 instrucciones en (7) programas de propósito general
      - 32 instrucciones en (3) programas de orientación científica
  - Las *verdaderas* dependencias de datos (RAW)
    - ✓ Las otras se pueden eliminar redenominando registros
- Ganancia en velocidad que proporciona
  - Menor que 2 en programas de propósito general
  - Menor que 4 en programas de orientación científica

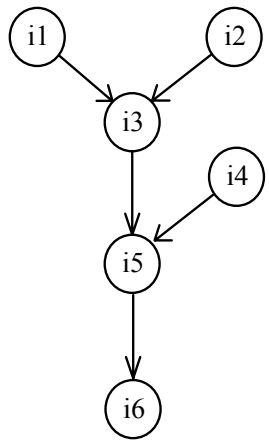
Planificación de bloques básicos: grafo de dependencias de datos (DDG)

- Muestra las precedencias impuestas por ese tipo de dependencias
- A la derecha se tiene el grafo que corresponde al bloque básico

```
i1: ini:  add r3, r1, r2
i2:       sub r4, r1, r2
i3:       mul r5, r3, r4
i4:       mul r7, r6, r6
i5:       sub r8, r7, r5
i6:       jn eti
```

En forma texto:

{(i1,i3), (i2,i3), (i3,i5), (i4,i5), (i5,i6)}



Planificación de bloques básicos..

- La **construcción del DDG** es la **tarea** del planificador **que más tiempo consume**
  - n instrucciones => n*(n-1)/2 pares de instrucciones por analizar
- Tipos de planificadores de bloques básicos
  - Planificadores de listas (*list schedulers*)
    - ✓ Los más usados
  - Planificadores basados en criterios
    - ✓ Planifican, en primer lugar el primer elemento (la primera instrucción) que mejor cumpla una secuencia de criterios

Planificadores de listas

- Los elementos (instrucciones en nuestro caso) se planifican en pasos. En cada **paso**
  - 1º/ Regla de **selección** para crear una lista de elementos elegibles
  - 2º/ Regla para **elegir** de la lista anterior **el mejor** conjunto de elementos (una instrucción en nuestro caso) para planificar en este paso

Ejemplo de planificador de listas: algoritmo de Warren..

- En cada paso, sobre el grafo de dependencias de datos
  - Crear el conjunto de nodos elegibles. Es elegible
    - ✓Todo nodo que no tienen predecesor
    - ✓Aquel nodo cuyos predecesores no bloquean su ejecución porque ya han producido los datos y están disponibles
      - Esta condición vendrá dada por:
        valor del contador de tiempo ≥
        ≥ tiempo más temprano de ejecución (v. trp. sgte.)

Ejemplo de planificador de listas: algoritmo de Warren..

- Elegir el que se va a planificar y quitarlo del grafo
  - ✓ El que está en el camino crítico
    - El camino más largo hasta el final del bloque básico
    - La longitud del camino se calcula sumando las latencias de ejecución de los arcos del camino
- Aumentar en 1 el contador de tiempo
- Calcular el **tiempo más temprano de ejecución** del nodo sucesor al que se ha quitado del grafo
  - ✓ Sumar el contador de tiempo (CT) y la latencia entre el nodo suprimido y su sucesor
    - En futuros pasos el nodo será **elegible si valor del CT ≥ tiempo más temprano de ejecución**

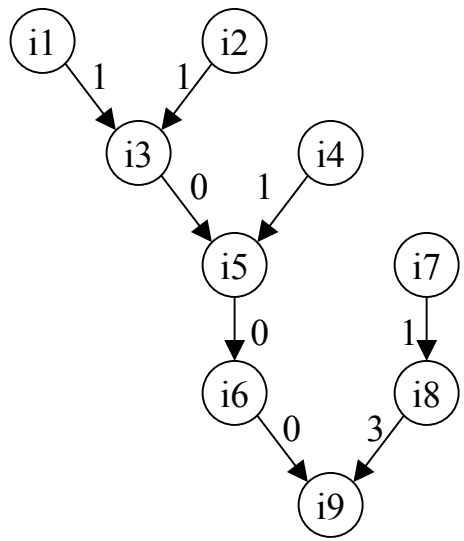
Ejemplo de planificador de listas: algoritmo de Warren..

- Suponemos que al siguiente código

```
i1:     l     r10, b (r20)
i2:     l     r11, c (r20)
i3:     add   r12, r10,  r11
i4:     l     r13, d (r20)
i5:     sub   r14, r12,  r13
i6:     st    r14, a (r20)
i7:     l     r15, e (r20)
i8:     cmp   r16, r15,  0
i9:     bc    r16,  ...
```

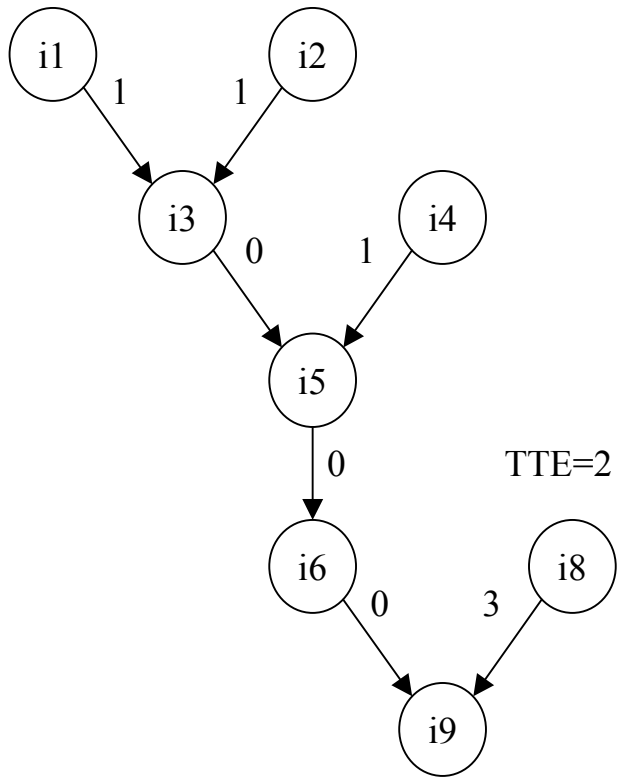
corresponde el **grafo de dependencias de datos** de la derecha

- En los arcos del grafo se indican las latencias
- Aunque no hay dependencia entre i6 e i9, i9 debe ser la última instrucción (si no no estaríamos en un bloque básico)



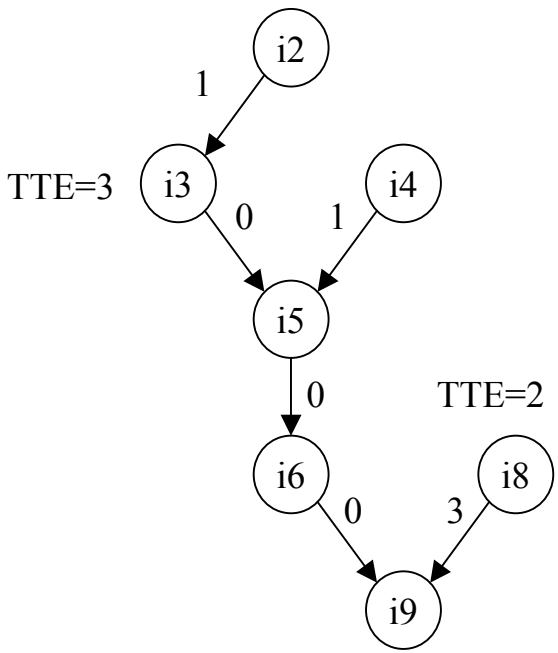
Ejemplo de planificador de listas: algoritmo de Warren..

- Contador de tiempo = 0
- Paso1:
  - Nodos elegibles: los que no tienen predecesor: {i1, i2, i4, i7}
  - Longitudes respectivas caminos de estos nodos: 1, 1, 1, 4 => **Nodo planificado i7**
  - CT = CT+1 (CT=1)
  - Tiempo más temprano de ejecución de i8 (sucesor de i7 en el grafo) = CT+1 = 2



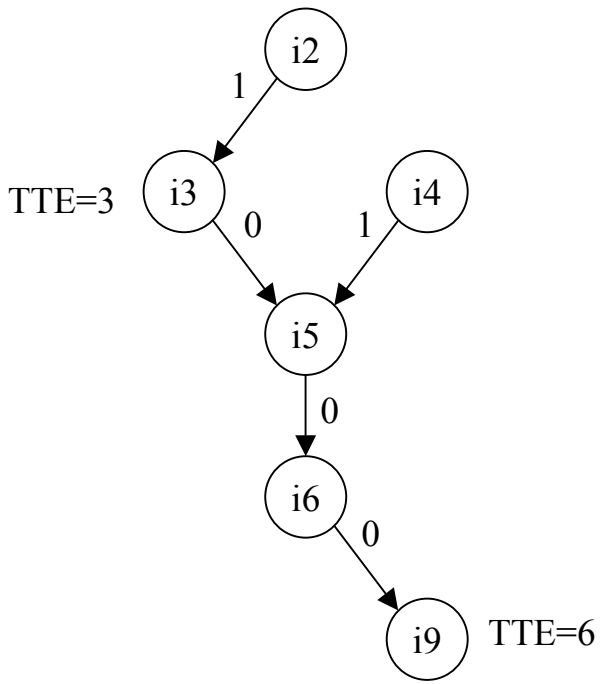
Ejemplo de planificador de listas: algoritmo de Warren..

- Paso 2:
  - Nodos elegibles {i1, i2, i4}
    (i8 no es elegible porque
    CT=1 < TTE(i8)=2)
  - Longitudes respectivas caminos:
    1, 1, 1 => **Nodo planificado i1**
    (se deshace el empate
    escogiendo el primero)
  - CT = 2
  - TTE(i3) = CT+latencia(i1,i3) =
    = 2+1 = 3



Ejemplo de planificador de listas: algoritmo de Warren..

- Paso 3:
  - Nodos elegibles {i2, i4, i8} (CT=2 ≥ TTE(i8)=2)
  - Longitudes respectivas caminos: 1, 1, 3 => **Nodo planificado i8**
  - CT = 3
  - TTE(i9) = CT+latencia(i8,i9) = = 3+3 = 6



Ejemplo de planificador de listas: algoritmo de Warren

- En pasos sucesivos se planifican, por este orden: i2, i4, i3, i5, i6, i9. Por tanto, el bloque básico planificado queda

```
i7:     l     r15, e (r20)
i1:     l     r10, b (r20)
i8:     cmp   r16, r15,  0
i2:     l     r11, c (r20)
i4:     l     r13, d (r20)
i3:     add   r12, r10,  r11
i5:     sub   r14, r12,  r13
i6:     st    r14, a (r20)
i9:     bc    r16,  ...
```

- Este bloque consume 9 ciclos y el original 15. Así pues se ha producido una mejora de 15/9 = 1,67

Planificación de bloques básicos

- Los planificadores de bloques básicos son usados por **procesadores encauzados** y por los primeros procesadores superescalares
  - Los VLIW no los utilizan porque necesitan una planificación más ambiciosa, es decir:
    - ✓ Planificación de bucles y
    - ✓ Planificación global

Planificación de bucles

- Los bucles son la **principal fuente de paralelismo de instrucciones**
- **Dos técnicas** fundamentales para planificar bucles
  - **Desenrollamiento** de bucles
    - ✓ Interviene como componente de métodos de planificación más sofisticados
      - Encauzamiento software y planificación de trazas
  - **Encauzamiento software** (*software pipelining*)
    - ✓ Técnica estándar de planificación de bucles

Latencias supuestas

- Latencias supuestas, en lo sucesivo, para el MIPS

Instrucción que produce el resultado	Instrucción que utiliza el resultado	Latencia (en ciclos de roloj)
Operación FP	Operación FP	3
Operación FP	*Store* doble	2
Load doble	Operación FP	1
Load doble	*Store* doble	0
Load entera	Operación entera	1
Hueco de retardo	Cualquiera	1

Ejemplo de bucle por desenrollar

- Suma de un escalar S a un vector X (valores en doble precisión):
  - **for** (i=1, i<=1000, i++) X[i] = X[i] +S
  - Supuestos:
    - ✓El escalar, S, está contenido en F2 (F2#F3)
    - ✓Inicialmente R1 apunta al primer componente del vector
    - ✓Por simplicidad, 8(R2) apunta al último componente

Compilación básica del bucle ejemplo

- Compilación básica (supuestos pág. anterior)

		Ciclo de emisión (relativo)
loop:	L.D F0, 0(R1)	1
	ADD.D F4, F0, F2	3
	S.D F4, 0(R1)	6
	DADDUI R1, R1, #-8	7
	BNE R1, R2, loop	9
	NOP	10

Planificación básica del bucle ejemplo

		Ciclo de emisión (relativo)
loop:	L.D F0, 0(R1)	1
	DADDUI R1, R1, #-8	2
	ADD.D F4, F0, F2	3
	BNE R1, R2, loop	4
	S.D F4, 8(R1)	6

- S.D al hueco de retardo
  - ✓ Al quedar detrás de DADDUI el desplazamiento pasa a ser 8
- S.D se emite un ciclo más tarde por su dependencia con ADD.D
- DADDUI se coloca antes de ADD.D para conseguir:
  - ✓ Eliminar la parada de ADD.D por su dependencia con L.D
  - ✓ Eliminar la parada de BNE por su dependencia con DADDUI

Desenrollamiento de bucles

- Idea: **Concatenar varios cuerpos del bucle** en un bloque básico
- **Ventajas**:
  - Más instrucciones en el bloque => **más posibilidades de planificarlo** => menos paradas
  - Menos pasadas por el nuevo bucle => **ahorro instrucciones de control del bucle** (en el ej., DADDUI y BNE)
- **Inconvenientes**:
  - Se necesitan **más registros**
    - ✓ No hay que utilizar los mismos en cada copia del bucle si no queremos perder las posibilidades de planificarlo
  - **Mayor** tamaño de **código**

Bucle desenrollado sin planificar y planificado

Bucle desenrollado no planificado		Bucle desenrollado y planificado	
	Ciclo de emisión		Ciclo de emisión
loop: L.D F0, 0(R1)	1	loop: L.D F0, 0(R1)	1
ADD.D F4, F0, F2	3	L.D F6, -8(R1)	2
S.D F4, 0(R1)	6	L.D F10, -16(R1)	3
L.D F6, -8(R1)	7	L.D F14, -24(R1)	4
ADD.D F8, F6, F2	9	ADD.D F4, F0, F2	5
S.D F8, -8(R1)	12	ADD.D F8, F6, F2	6
L.D F10, -16(R1)	13	ADD.D F12, F10, F2	7
ADD.D F12, F10, F2	15	ADD.D F16, F14, F2	8
S.D F12, -16(R1)	18	S.D F4, 0(R1)	9
L.D F14, -24(R1)	19	S.D F8, -8(R1)	10
ADD.D F16, F14, F2	21	DADDUI R1, R1, #-32	11
SD F16, -24(R1)	24	S.D F12, 16(R1)	12
DADDUI R1, R1, #-32	25	BNE R1, R2, loop	13
BNE R1, R2, loop	27	SD F16, 8(R1)	14
NOP	28		

Encauzamiento software..

- Tomemos
  - Cuerpo del bucle

    fload f10, 0 (r2)
    fmul f10, f2, f10
    fstore 200(r2), f10
    addi r2, r2, 1
  - **Procesador ILP:**
    - ✓ Unidades funcionales separadas para **FP, FX** (operaciones enteras), ***load* y *store***
      - En todas el **intervalo de iniciación es 1 ciclo**
    - ✓ **Latencias** por dependencias de datos: ***fmul,* dos ciclos;** el resto cero ciclos

- Ejecución de una iteración

Ciclo	Instrucción	
c	*fload*	*f10, 0(r2)*
c+1	*fmul*	*f10, f2, f10*
c+2	*addi*	*r2, r2, 1*
c+3	*nop*	
c+4	*fstore*	*199(r2), f10*

- La siguiente iteración puede empezar en el ciclo c+1

c+1	*fload*	*f11, 0(r2)*
c+2	*fmul*	*f11, f2, f11*
c+3	*addi*	*r2, r2, 1*
c+4	*nop*	
c+5	*fstore*	*199(r2), f11*

Encauzamiento software..

- El resultado para 7 pasadas será

Ciclo	Iteración						
	1	2	3	4	5	6	7
c c+1 c+2 c+3	fload fmult addi nop	 fload fmult addi	 fload fmult	 fload			
c+4 **c+5** **c+6**	**fstore**	**nop** **fstore**	**addi** **nop** **fstore**	**fmult** **addi** **nop**	**fload** **fmult** **addi**	**fload** **fmult**	**fload**
c+7 c+8 c+9 c+10				fstore	nop fstore	addi nop fstore	fmult addi nop fstore

Encauzamiento software..

- Los ciclos c+4, c+5 y c+6:
  - Utilizan todo el paralelismo disponible
  - Siguen un **patrón repetitivo** => Pueden ser reemplazados por un bucle
- El código del encauzamiento software queda dividido en tres partes
  - **Código de inicio** (*start-up code, prologue* o *prelude*)
  - **Bucle del patrón repetitivo**
  - **Código de finalización** (*epilogue* o *postlude*)

Encauzamiento software

- Problema principal:
  - **Encontrar el patrón repetitivo** que pueda ejecutarse en el menor número de ciclos
    - ✓ No siempre es tan sencillo como en el ejemplo anterior
- Un **desenrollamiento previo** del bucle suele ser fundamental para un **encauzamiento software más efectivo**

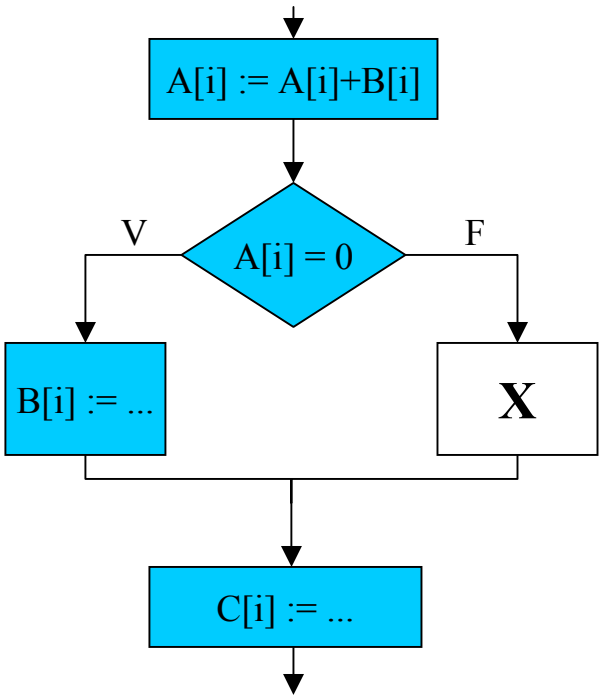
Planificación global

- Planificación **más allá de los límites de los bloques básicos y de los bucles**
- Tareas propias de la planificación global
  - Planificación con **movimiento de código atravesando los límites de los bloques básicos**
  - Planificación de bucles
  - Manejar llamadas a procedimientos dentro de bucles, manejar dependencias de control, dar soporte a la política de emisión de instrucciones (típico de procesadores superescalares), etc.

Técnica ejemplo de planificación global: planificación de trazas

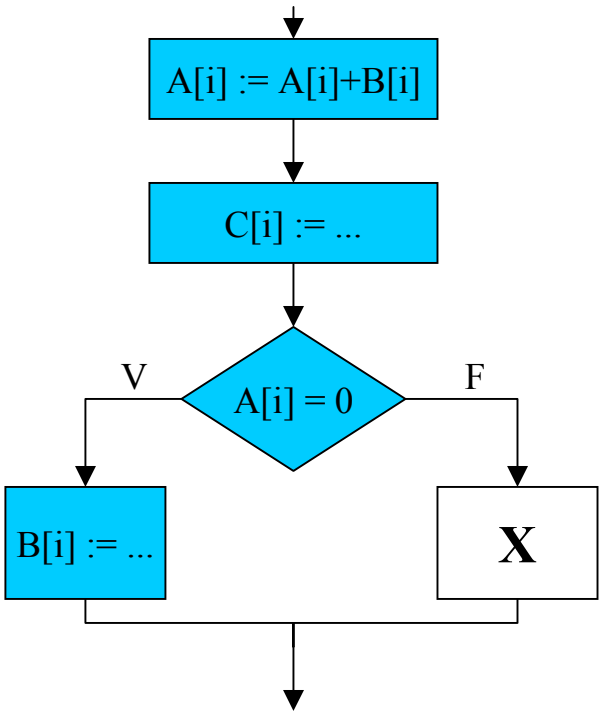
- 1º: **selección** de trazas
  - Traza: posible **secuencia de ejecución de operaciones dentro de un bucle** (puede incluir bifurcaciones y saltos)
  - Se selecciona la traza **más probable**; después, la siguiente más probable, etc. hasta que toda operación del programa pertenezca a alguna traza seleccionada
- 2º: **compactación** de trazas
  - **Las operaciones de cada traza se paralelizan** agrupándose en un número menor de instrucciones grandes (VLIW)

Ejemplo de selección de una traza



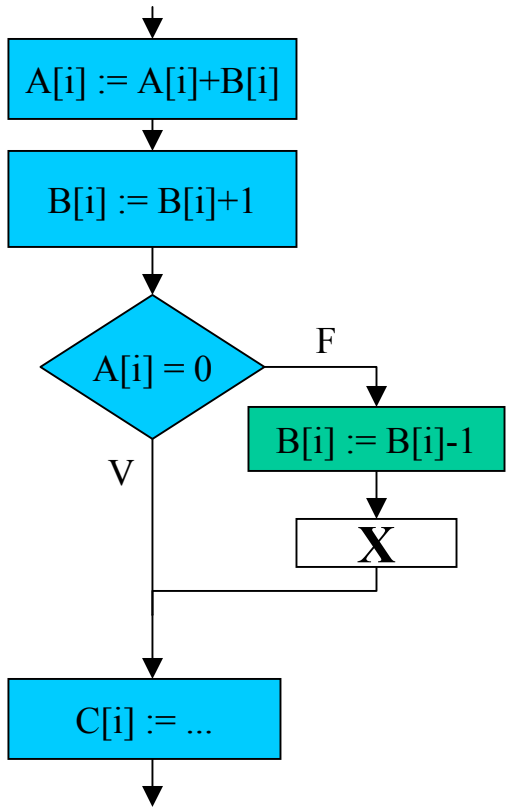
- Se ha seleccionado la traza con **fondo azul** porque se prevee **más probable** que la condición sea verdadera
- Una vez seleccionada hay que compactar la traza
  - La compactación supone **mover código hacia arriba**

Movimiento de código en planificación de trazas



- **Si "*C[i]* := ..." no tiene dependencias**, puede colocarse antes de la bifurcación

Movimiento de código en planificación de trazas



- Si *"B[i] := ..."* no tiene dependencias:
  - Se puede colocar antes de la bifurcación
    - ✓ Movimiento **especulativo** de código: sólo **útil si previsión acertada**
  - Suele ser preciso **añadir** **código compensatorio** (*B[i] := B[i]-1*) por si se sigue el camino no previsto (***bookkeeping code***)

Planificación de trazas: principales dificultades

- El **obstáculo más importante** para mover instrucciones son las **dependencias de datos** RAW
- Los **movimientos especulativos** de código son **problemáticos**
  - Provocan una considerable **expansión de código**
  - Es clave para la mejora del rendimiento que las trazas previstas correspondan a los caminos que sigue el programa un **alto porcentaje** de veces
    - ✓ En los caminos no previstos el rendimiento se degrada por la ejecución del código compensatorio
  - Surgen dificultades cuando el código que se mueve puede provocar **interrupción** (el código compensatorio no las resuelve)
- Cuando no hay suficiente **paralelismo en las trazas** un **desenrollamiento de bucles** puede suministrar ese paralelismo