Escola Tècnica Superior d'Enginyeria Informàtica

etsinf

Optimización de Código Intermedio

Lenguajes de Programación y Procesadores de Lenguajes

V. 22

Tabla de contenido:

1. Introducción

En general el código intermedio generado por un compilador suele ser bastante ineficiente. Basta observar el código generado para la evaluación de una expresión para darse cuenta de este hecho.

Ejemplo 1:

Código fuente → a := x^2 + y
Código intermedio: → t1 := 2
t2 := x^t1
t3 := t2 + y
a := t3
Código mejorado: → t1 := x * x
a := t1 + y

En el Ejemplo 1 observamos dos tipos de mejoras: talla del código, el número de instrucciones se reduce a la mitad, y eficiencia en la ejecución, hay dos asignaciones menos y una operación de menor coste.

Criterios para optimizar:

- Menor tiempo de ejecución
- Menor tamaño del programa

Estos dos criterios son a menudo contradictorios y requieren algoritmos de generación de código distintos.

Sería más correcto hablar de "mejora del código", puesto que en general no existen algoritmos que nos permitan obtener el programa equivalente más pequeño, o más rápido, y por lo tanto no podremos decir que estamos obteniendo el código "óptimo".

Por ejemplo, podríamos plantearnos un optimizador de talla del código. Bajo este supuesto el código óptimo para cualquier programa que no pare sería:

1: GOTO 1

En consecuencia, un optimizador de código traduciría cualquier programa que no parase en un programa objeto de una única instrucción como la indicada. Ahora la existencia del optimizador sería equivalente a determinar si un programa parará o no, pero esta última cuestión es indecidible por lo tanto no es posible encontrar tal optimizador.

La mayoría de algoritmos de mejora se pueden ver como la aplicación de varias **transformaciones** sobre la representación intermedia del código.

Condiciones que deben cumplir las transformaciones a aplicar:

- Debe preservar el significado del programa.

Considerando el programa como una caja negra, el programa mejorado y el inicial deben producir los mismos resultados para las mismas entradas.

- Debe, como promedio, acelerar o reducir la talla del programa en una cantidad mensurable
  Dado un conjunto amplio de programas las mejoras introducidas en una mayoría de ellos deben ser medibles. En otras palabras, puede que haya algún caso en que las transformaciones introducidas empeoren la talla o la eficiencia del programa.
- Debe valer la pena: Considerar el coste de implantarla y ejecutarlas, y el beneficio que va a producir.

Fuentes de optimización

- **Programa fuente**.
  Optimizaciones cuyos efectos podrían obtenerse con una implementación más apropiada del programa fuente. El programador puede: perfilar el programa, cambiar el algoritmo, mejorar bucles …por ejemplo podría substituir "if TRUE then A else B" por "A"
- **Código intermedio**.
  Hay mejoras que no son posibles en el programa fuente, Ej. El código generado para acceder a los elementos de una matriz dentro de un bucle, calculará varias veces la misma función de acceso. Ejemplos de estas transformaciones son:
  - Subexpresiones comunes
  - Transformaciones algebraicas.
  - Copias de variables
  - Código inactivo
  - Bucles
- **Código objeto.**
  En este nivel el uso adecuado de los registros, así como de las instrucciones propias de la máquina destino pueden suponer beneficios importantes respecto de la eficiencia del programa objeto. Ej. Aprovechar una instrucción de la máquina destino que ya proporciona una suma y un salto condicional en función del resultado.

Ejemplo 2:

Código fuente → var a: array [1..10, 1..20] of integer; {talla de un entero 2 u.m.}
…
while a[i, k] < M do k := k – 1;

Código intermedio: →
100 t1:= i * 20
101 t2 := t1 + k
102 t3 := t2 * 2
103 t4 := a[t3]
104 if t4 < M goto 106
105 goto 108
106 k := k – 1
107 goto 100
108

Código mejorado: → 100 t1:= i * 20
101 t2 := t1 + k
102 t3 := t2 * 2
103 t4 := a[t3]
104 if t4 ≥ M goto 107
105 t3 := t3 - 2
106 goto 103
107

En el 33.2 se observa, sin entrar en detalles de cómo se ha realizado la mejora, que las instrucciones 100, 101 y 102 sólo se calculan una vez en vez de una cada iteración del bucle. Esta mejora no se puede realizar sobre el programa fuente, puesto que la parte optimizada corresponde a la función de acceso a los elementos de una matriz.

Clasificación de las optimizaciones:

- Local: si puede realizarse observando únicamente una parte del programa que se ejecuta secuencialmente sin saltos (bloque básico).
- Global: Para realizarlas se necesita conocer el flujo de control entre las distintas partes del programa.

2. Bloques básicos y grafos de flujo

Bloque básico (bb):

Secuencia de instrucciones consecutivas en las que el flujo de control entra al principio y sale al final, sin detenerse y sin posibilidad de saltar, excepto al final. En consecuencia las instrucciones de un bloque básico serán instrucciones de asignación excepto posiblemente la última instrucción que puede ser una instrucción de salto condicional o incondicional o, una llamada a un procedimiento. Además, cualquier salto a una instrucción del bloque básico desde fuera del bloque básico o desde la última instrucción del bloque básico sólo puede ser a la primera instrucción del bloque básico.

Llamaremos **líder** de un bloque básico a la primera instrucción del bloque básico.

Algoritmo 3.1: Algoritmo de partición de un programa en bloques básicos
Entrada: Secuencia de instrucciones del programa en código intermedio.
Salida: Lista de bloques básicos.

1. Calcular líderes
   1.1 La primera instrucción del programa es un líder.
   1.2 Cualquier instrucción destino de un salto condicional o incondicional o de una llamada a procedimiento es un líder.
   1.3 Cualquier instrucción inmediatamente a continuación de un salto condicional o incondicional o de una llamada a procedimiento es un líder.

2. Un bloque básico lo forma un líder y todas las instrucciones que le siguen hasta el siguiente líder (sin incluir a éste último).

Ejemplo 3:

(1) i := m - 1	(20) t_6 := 4 * i
(2) j := n	(21) x := a[t_6]
(3) t_1 := 4 * n	(22) t_7 := 4 * i
(4) v := a[t_1]	(23) t_8 := 4 * j
(5) if i < j goto (7)	(24) t_9 := a[t_8]
(6) goto (29)	(25) a[t_7] := t_9
(7) i := i + 1	(26) t_{10} := 4 * j
(8) t_2 := 4 * i	(27) a[t_{10}] := x
(9) t_3 := a[t_2]	(28) goto (5)
(10) if t_3 >= v goto (12)	(29) t_{11} := 4 * i
(11) goto (7)	(30) x := a[t_{11}]
(12) j := j - 1	(31) t_{12} := 4 * i
(13) t_4 := 4 * j	(32) t_{13} := 4 * n
(14) t_5 := a[t_4]	(33) t_{14} := a[t_{13}]
(15) if t_5 <= v goto (17)	(34) a[t_{12}] := t_{14}
(16) goto (12)	(35) t_{15} := 4 * n
(17) if i >= j goto (19)	(36) a[t15]:= x
(18) goto (20)	
(19) goto (28)	

El conjunto de líderes de este código son las instrucciones: (1), (5), (6), (7), (11), (12), (16), (17), (18), (19), (20), (28) y (29). Los bloques básicos estarían formados por:

B1: (1) a (4) B2: (5) B3: (6) B4: (7) a (10) B5: (11)

B6: (12) a (15) B7: (16) B8: (17) B9: (18) B10: (19)

B11: (20) a (27) B12: (28) B13: (29) a (36)

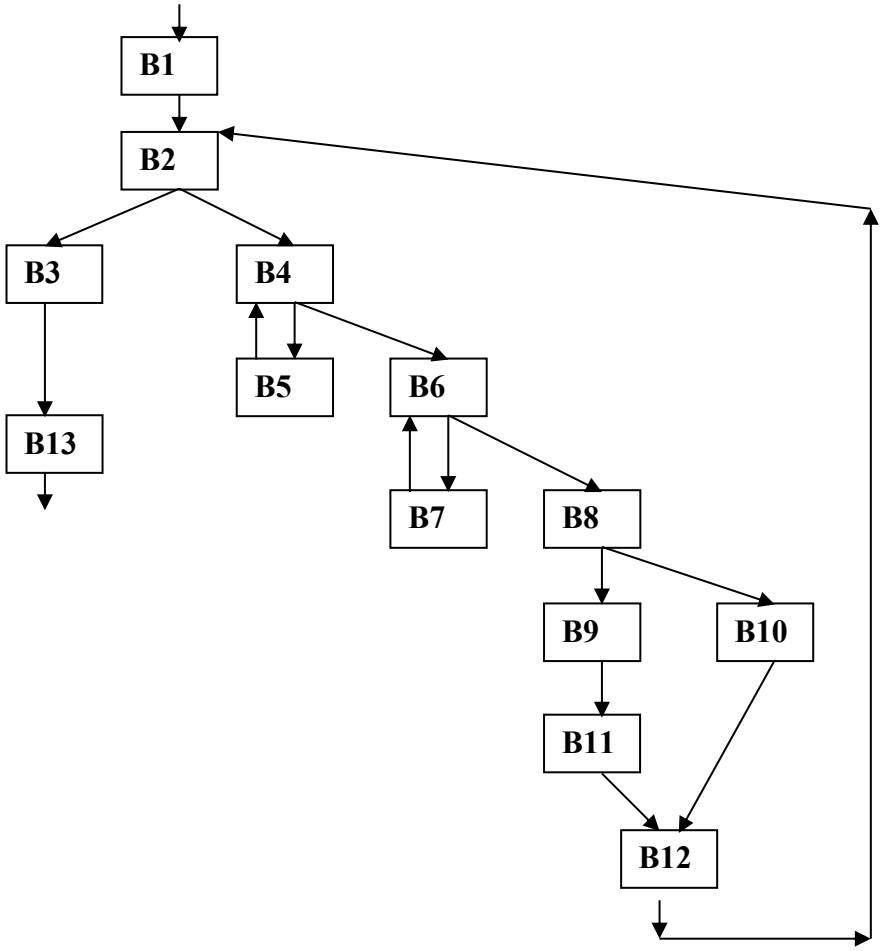
Grafo de flujo

Grafo dirigido que representa el flujo de control de un programa. Cada bloque básico será un **nodo** del grafo. El **nodo inicial** es el bloque básico que contiene la primera instrucción del programa. Habrá una **arista dirigida** del nodo BX al BY, (BX → BY) si BY puede ir inmediatamente después de BX en alguna secuencia de ejecución del programa. Es decir:

a) Hay un salto condicional o incondicional desde la última instrucción de BX a la primera de BY, ó

b) BY sigue inmediatamente a BX en el orden del programa, y BX no termina con un salto incondicional.

En este contexto una llamada a un procedimiento (CALL n) se puede considerar como un salto incondicional a la instrucción "n" (primera instrucción del procedimiento). La instrucción de retorno (RET) también se puede considerar como un salto incondicional al bloque que siga, en la secuencia del programa, a la llamada a ese procedimiento. (¿Qué ocurre si hay varias llamadas al procedimiento?)



3. Optimizaciones locales

3.1. Transformaciones algebraicas

Las transformaciones algebraicas tienen por objeto sustituir una expresión por otra algebraicamente equivalente, entendiendo por algebraicamente equivalente que la igualdad entre ambas expresiones se puede establecer a partir de las propiedades de los operadores y operandos que intervienen en ellas.

Se pueden emplear innumerables transformaciones algebraicas. Serán útiles aquellas que simplifiquen las expresiones o sustituyan operaciones por otras equivalentes menos costosas.

1. Simplificaciones algebraicas:

Expresiones de identidad: identidad $\theta\ \alpha = \alpha$

Ej. x + 0 = x ; true and x = x ; y * 1 = y ; …

Propiedad conmutativa: $\alpha\ \theta\ \beta = \beta\ \theta\ \alpha$

Propiedad asociativa: $\alpha\ \theta\ (\beta\ \theta\ \gamma) = (\alpha\ \theta\ \beta)\ \theta\ \gamma$

Operador θ_1 distributivo respecto a θ_2: $\alpha\ \theta_1\ (\beta\ \theta_2\ \gamma) = (\alpha\ \theta_1\ \beta)\ \theta_2\ (\alpha\ \theta_1\ \gamma)$

Operador unario autoinverso: $\theta\ \theta\ \alpha = \alpha$

Como por ejemplo - (-a) = a; not not x = x

Ejemplo 4:

La aplicación sucesiva de algunas de las propiedades anteriores nos permite realizar la siguiente transformación:

A*(B*C)+(B*A)*D+A*E = A*(B*(C+D)+E)

En la expresión de la izquierda hay 5 productos y 2 sumas, y en la expresión de la derecha únicamente hay 2 productos y 2 sumas.

2. Reducción de intensidad.

La mejora de código consiste en sustituir una operación por otra equivalente pero menos costosa, por ejemplo, la sustitución de ciertos productos por sumas o de potencias por productos,…

y := x ^2 → y := x * x; y := x * 2 → y := x + x;

3. Cálculo previo de constantes.

En el código objeto a menudo aparecen operaciones que se pueden realizar en tiempo de compilación, puesto que sus operandos son conocidos en ese momento. Esta técnica también se conoce como plegado (“holding”).

Por ejemplo:

PI := 3.1415

x := 3 * PI

La instrucción anterior puede ser sustituida por x := 6.2830 en el momento de la compilación.

Ejemplo 5

En este ejemplo muestra una utilización de las transformaciones algebraicas:
y := 3
t1 := 2 + y

```
t2 := t1 + z
x  := t2 + 5
```

Por aplicación de las reglas: conmutativa, asociativa y propagación de constantes obtendríamos.

```
x := 10 + z
```

En general la aplicación de las transformaciones algebraicas requiere tomar ciertas precauciones. Por ejemplo:

- Deben aplicarse localmente. En el siguiente ejemplo no se puede utilizar el cálculo previo de constantes:

```
1   i := 0
2   i := i + 1
3   if i<N goto 2
```

- Los números en coma flotante no cumplen siempre todas las propiedades algebraicas. El siguiente ejemplo demuestra que la suma no es asociativa. Esto es una consecuencia del truncamiento y del redondeo al representar un número real en memoria. El programa del siguiente ejemplo ha sido compilado y ejecutado en un compilador comercial:

```
program precision;
var x,y,z:real;
{ cálculo de números reales por Emulación}
begin
  x := 11111111111.0;  y := 2.45;  z := 0.055; {probar: 2.46; 0.055 y 2.46; 0.054}
  writeln(( x + y ) + z ); { 1.1111111114E+10 }
  writeln( x + ( y + z )); { 1.1111111113E+10 }
end.
```

3.2. Transformaciones que preservan la estructura

3.2.1. Introducción al análisis de flujo de datos

Una asignación de la forma:

```
a := b + c
```

decimos que **define** "a" y **usa** "b" y "c". En general una asignación define siempre la variable de la izquierda de la asignación y usa cualquier variable que ocurra en su parte derecha. Otro tipo de instrucciones siempre usan las variables que ocurren en ellas. Ejemplo: if i < v goto L1 usa las variables "i" y "v".

Las siguientes definiciones se pueden aplicar sobre nodos que representan bloques básicos y también sobre nodos que representen instrucciones individuales. En cada caso, los arcos del grafo de flujo indican los posibles caminos que puede seguir el control del flujo del programa.

Una variable está **activa** en un arco del grafo de flujo si existe un camino dirigido desde ese arco a un uso de la variable que no pasa por ninguna definición de la variable. En el siguiente ejemplo las variables activas en cada arco son: (I)→ {y, z, b} ;(II) → {x, b, z}; (III) → {a, x, b, z}

Ejemplo 6:

(1) x := …

(I)

(2) x := y + z

(II)

(3) a := x + b

(III)

(4) y := a + x

(5) if x < 5 goto 2

(6) b := x

Una variable x está activa a la entrada de un nodo n (x ∈ **ent**[n]) si está activa en cualquiera de los arcos de entrada del nodo. De igual forma, una variable x está activa a la salida de un nodo n (x ∈ **sal**[n]) si está activa en cualquiera de los arcos de salida del nodo.

En el Ejemplo 6:

	ent[n]	sal[n]
(1)		y, z, b
(2)	y, z, b	x, b, z
(3)	x, b, z	a, x, b, z
(4)	a, x, b, z	x, y, z, b
(5)	x, y, z, b	x, y, z, b
(6)	x	

Cálculo de **ent[]** y **sal[]** en un bloque básico.

Supondremos, como caso excepcional, que la última instrucción del bloque básico puede ser un salto condicional a la primera instrucción del bloque básico. Las funciones **def[]** y **usa[]** calculan las variables definidas o usadas en cada instrucción de un bloque básico.

def[i] = Variable que se define en la instrucción de asignación “i”.

usa[i] = Conjunto de variables que se usan en la parte derecha de la instrucción “i”.

Las ecuaciones siguientes establecen la relación que existe entre las funciones **ent[]**, **sal[]**,**def[]** y **usa[]**.

$$ent[i] = usa[i] \cup (sal[i] - def[i])$$

$$sal[i] = \bigcup_{s \in Suc[i]} ent[s]$$

La primera de las ecuaciones nos indica que las variables activas a la entrada de una instrucción serán las variables que se usen en la instrucción "i" junto con las que estén activas a la salida menos la variable que se pueda definir en esta instrucción. La segunda ecuación establece que las variables activas a la salida serán las activas a la entrada de todas las instrucciones que puedan seguir a la instrucción "i".

Con el fin de establecer las variables activas en cada punto de un bloque básico perfilaremos la definición de bloque básico. Un bloque básico estará formado por una tupla de tres elementos: (I, E, S): I el conjunto de instrucciones que forman el bloque básico, E el conjunto de variables activas a la entrada del bloque básico, y S el conjunto de variables activas a la salida del bloque. En el caso de que la última instrucción del bloque básico sea un salto condicional a la primera instrucción del bloque básico, entonces S será el conjunto de variables activas en el arco que va a la instrucción que sigue a la instrucción condicional.

Algoritmo 3.2: Cálculo de ent[] y sal[] en un bloque básico ($\{ins_i\}_1^n$, E, S).

para todo i ∈ 1..n **hacer** ent[i] := {}; sal[i] := {};
repetir
  para i := n **hasta** 1 **hacer**
    si i = n **entonces** sal[i] := $\cup_{s \in Suc[i]} ent[s] \cup S$ **si no** sal[i] := $\cup_{s \in Suc[i]} ent[s]$
    ent[i] := usa[i] ∪ (sal[i] – def[i])
  fin para
hasta ningún ent[i] o sal[i] cambie.

Ejemplo 7: Consideremos el bloque básico ({2,3,4,5}, {y, z, b}, {x}) del Ejemplo 6. si aplicamos el algoritmo para el cálculo de **ent[]** y **sal[]** obtendremos:

			1ª iteración		2ª iteración		3ª iteración	
	def	**usa**	**ent[]**	**sal[]**	**ent[]**	**sal[]**	**ent[]**	**sal[]**
x := y + z	x	y, z	y, z, b	x, b	y, z, b	x, b, z	y, z, b	x, b, z
a := x + b	a	x, b	x, b	a, x	x, b, z	a, x, z, b	x, b, z	a, x, z, b
y := a + x	y	a, x	a, x	x	a, x, z, b	x, y, z, b	a, x, z, b	x, y, z, b
if x < 5 goto 2		x	x	x	x, y, z, b	x, y, z, b	x, y, z, b	x, y, z, b

3.2.2. Algunas transformaciones que preservan la estructura

Un bloque básico calcula un conjunto de expresiones que corresponden a los valores de las variables activas al finalizar el bloque. Dos bloques básicos son **equivalentes** si calculan el mismo conjunto de expresiones. Bajo este punto de vista veamos técnicas que transforman bloques básicos en otros bloques básicos equivalentes.

1.- Eliminación de subexpresiones comunes.

Una ocurrencia de una expresión E se denomina **subexpresión común**, si E ha sido previamente calculada y los valores de las variables dentro de E no han cambiado desde el cálculo anterior. Se puede evitar recalcular la expresión si se puede utilizar el valor calculado previamente.

Ejemplo 8:

(1) a := b + c		a := b + c
(2) b := a - d	→	b := a – d
(3) c := b + c		c := b + c
(4) d := a - d		d .= b

Observar que la instrucción (1) y la (3) no calculan las mismas expresiones. En cambio, a – d es una subexpresión común a las instrucciones (2) y (4).

2.- Propagación de copias

Tras una asignación de copia, i := d, se utilizará siempre d en lugar de i.

Aunque en un principio parece que el código no mejore, nos facilitará la posible eliminación posterior de asignaciones a i (ver "eliminación de código inactivo")

(1)x := t3		(1)x := t3
(2) t9 := t5		(2)t9 := t5
(3)a[t2] := t9	→	(3)a[t2] := t5
(4)a[t4] := x		(4)a[t4] := t3

Ahora si la variable t9 no esta activa a la salida del bloque entonces la instrucción (2) puede eliminarse. Esto ocurre a menudo con las variables temporales que introduce el generador de código intermedio.

3. Eliminación de código inactivo

Llamamos código inactivo a la secuencia de instrucciones que calculan valores que nunca llegan a utilizarse o realizan acciones que nunca llegan a ejecutarse. Este código puede aparecer como consecuencia de alguna de las transformaciones anteriores.

Dado un bloque básico (I, E, S) la aplicación reiterativa de las siguientes reglas, hasta que el bloque básico no sufra ninguna modificación, permite eliminar las instrucciones inactivas del bloque básico.

Algoritmo 3.3: Eliminación de instrucciones inactivas

Para toda i ∈ I si la instrucción "i" define la variable "a" y a ∉ **sal**[i] **entonces** I := I – {ins_i }

Para toda a ∈ E **si** a ∉ **ent**[1] **entonces** E := E – {a}

Ejemplo 9:

Dado el bloque básico

B = ({ f := a + a; g := f * c; f := a + b; g := a * b}, {a, b, c}, {f, g})

la aplicación de estas reglas produce:

B' = ({ f := a + a; f := a + b; g := a * b}, {a, b, c}, {f, g})

B'' = ({ f := a + b; g := a * b}, {a, b}, {f, g})

4. Renombrar variables temporales

Siempre se puede transformar un bloque básico en otro equivalente en el que cada instrucción que define una variable temporal, emplea una variable nueva. En este caso se dice que el bloque básico está en forma normal. Esta modificación permite una mejor utilización de los registros en el momento de generar código objeto.

Ejemplo 10:

si "u" es una variable temporal nueva entonces
t := b + c → u := b + c
y, además, se cambian todos los usos posteriores de t por u.

5. Intercambiar dos instrucciones adyacentes independientes.

Dos instrucciones,

t1 := b + c

t2 := x + y

se pueden intercambiar si y sólo si (ni x ni y son t1) y (ni b ni c son t2)

Un bloque básico en forma normal permite todos los intercambios de instrucciones posibles.

3.2.3. Grafos dirigidos acíclicos: GDAs

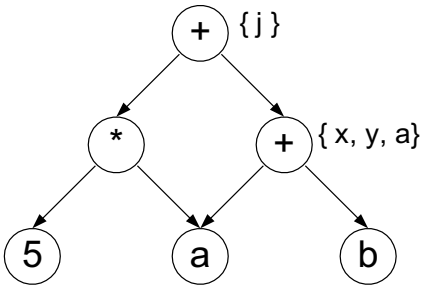
Una representación adecuada para mejorar el código de un bloque básico es un grafo acíclico dirigido. Este tipo de grafo tiene cierta similitud con un árbol de sintaxis abstracta, en cuanto que los nodos "internos" del grafo los ocupan los operadores al igual que en el árbol de sintaxis abstracta.

Ejemplo 11: el siguiente bloque básico, donde suponemos que las variables activas a la salida son {j, a, y}, generaría el GDA:

x := a + b
j := 5 * a
j := j + x

y := a + b
a := x

→



En el ejemplo podemos observar que los nodos del grafo están ocupados por variables (posiblemente constantes) cuyo valor se supone conocido a la entrada al bloque o por operadores. Junto al valor del nodo aparecen los nombres de las variables que contienen, al final del bloque, el valor de la expresión asociada al subgrafo correspondiente a ese nodo. Observar también que, si una variable se define varias veces en el bloque, sólo aparecerá en el GDA su última definición. Por último, hay que distinguir entre las variables que *ocupan* un nodo (representan el valor inicial de la variable al entrar en el bloque básico) que destacaremos utilizando un subíndice 0, y las variables que se *definen* en el bloque, que aparecerán en la lista de variables asociadas a los nodos.

Algoritmo 3.4: Construcción del GDA a partir de un bloque básico.

ENTRADA: Un bloque básico β = (I, E, S)

SALIDA: GDA

USA:

Cada nodo del GDA tendrá:

Una etiqueta: etiq

Un hijo derecho y un hijo izquierdo

Una lista de variables que toman el valor representado por el nodo: lista_var:

CreaNodo(v, i ,d): Función que crea un nodo etiquetado *v*, hijo izquierda *i*, e hijo derecha *d*.

Ult_nodo(v): Función que devuelve el último nodo que contiene, en su lista de variables (o en su etiqueta si no está en ninguna lista) a la variable *v*

METODO:

```
Forall variable x ∈ usa[β] ∪ def[β]  do
    Ult_nodo(x) := NULL

Forall instrucción Si ( x := y op z ) ∈ I  do  {
   if ult_nodo(y) = NULL   then CreaNodo(y, NULL, NULL)
   if ult_nodo(z) = NULL   then CrearNodo(z, NULL, NULL)
   if  ¬∃ nodo | Etiqueta(nodo) = op ∧ HijoIzq = Ult_nodo(y)   ∧
            HijoDer(nodo) = Ult_nodo(z) )
   then   nodo := CrearNodo(op, Ult_nodo(y), Ult_nodo(z))
   if Ult_nodo(x) ≠ NULL then {
      Quitar x de la lista de variables de Ult_nodo(x)
      Añadir x a la lista de variables de nodo, que pasa a ser el Ult_nodo de x
   }
}
```

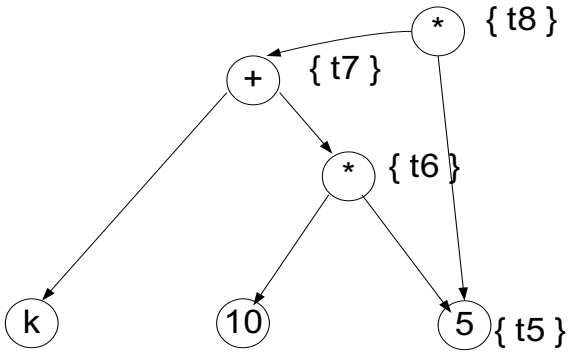
3.2.4. Reconstrucción del código intermedio a partir de un GDA

El proceso de reconstrucción es una tarea no determinista en la que la obtención de un bloque básico equivalente exige el respeto de ciertas reglas:

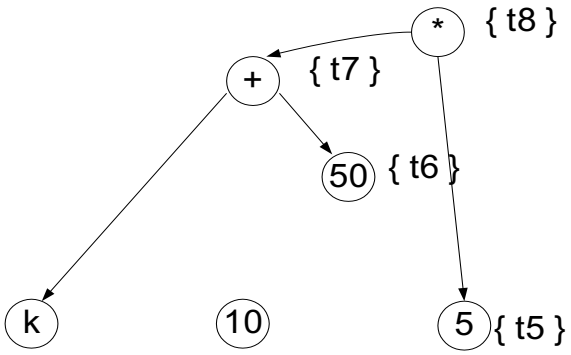
1. **Eliminación de código inútil.** Si un nodo **sin ascendientes** no tiene en el campo lista_vbles **ninguna variable activa** a la salida del bloque, eliminar el nodo. Repetir este paso hasta que no se elimine ningún nodo.

2. **Plegado de constantes**. Si un nodo tiene como descendientes directos dos hojas cuyas etiquetas son constantes, sustituir el nodo por una hoja cuya etiqueta sea el resultado del cálculo con las constantes y cuya lista_vbles sea la misma que la del nodo. Las hojas del nodo en cuestión no se eliminan, sólo se suprimen los arcos de este nodo a estas hojas.

BB = (P, {}, {k})
t5 := 5
t6 := 10 * t5
t7 := k + t6
t8 := t7 * t5



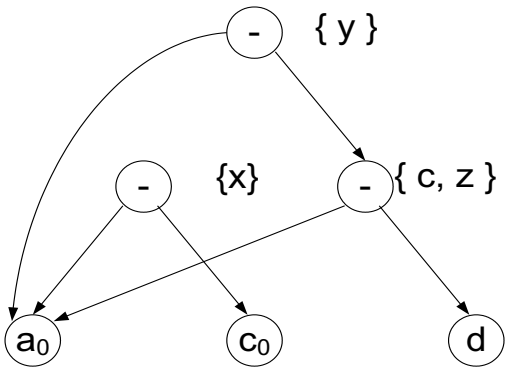
El GDA quedaría:



3. El orden de reconstrucción debe ser un **orden topológico**, es decir, sólo se generará código para un operador y sus hijos cuando se haya generado el código necesario para evaluar los subgrafos correspondientes a sus hijos.

BB = (P, {a, c, d}, {x, y, z})
x := a – c
c := a – d
y := a – c
z := a – d

➔



Las dos siguientes secuencias respetan el orden topológico:

x := a – c		z := a - d	
z := a – d	(1)	x := a – c	(2)
y := a – z		y := a - z	

4. No se generará código para definir una variable X (que aparece en lista_vbles de algún nodo) que también aparezca como **etiqueta de un nodo hoja** (X_0) hasta haber generado el código que corresponde a todos los usos del valor representado por el nodo hoja de esa variable (X_0).

   La siguiente secuencia, obtenida a partir del anterior GDA, incumple esta regla:

   c := a – d
   x := a – c
   y := a – c (3)
   z := c

5. Si un nodo contiene **más de una variable** en el campo lista_vbles se elegirá como variable, para contener el valor de la expresión, aquella que esté activa a la salida del bloque. Si no hubiera ninguna se elegirá cualquier variable. Si a la salida hay más de una activa se generarán tantas instrucciones de copia como sea necesario.

   En el caso de las hojas:

   - Si la etiqueta es una variable y no se modifica dentro del bloque, se considerará ésta junto a su lista de variables y se aplicará el mismo criterio.
   - Si la etiqueta es una constante, y ninguna variable en el campo lista_vbles está activa a la salida del bloque, no se generará ninguna instrucción para evaluar esta hoja.

   Las secuencias anteriores (1) y (2) cumplen esta regla, en cambio la secuencia (3) no.

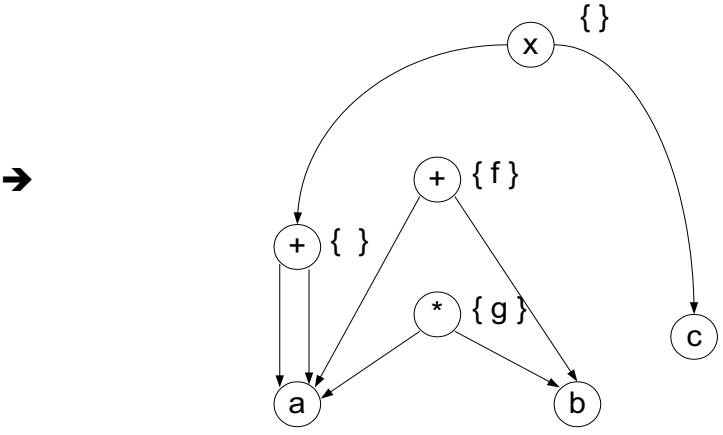
6. Si un nodo **no contiene ninguna variable** en el campo lista_vbles y se necesita almacenar temporalmente su valor, se creará una nueva variable temporal.

7. En los casos en que haya que seguir un **determinado orden**, éste deberá hacerse constar en el GDA. Por ejemplo, las instrucciones de control siempre ocupan la última posición en un bloque básico, la asignación a un elemento de un array si posteriormente se usa el array en el bloque, …

   B = (P, {a, b, c}, {f, g})

   f := a + a

   g := f * c
   f := a + b
   g := a * b



la aplicación iterativa de estas reglas produce:

f := a + b
g := a * b

Si dado un bloque básico se construye su GDA y luego se reconstruye el código respetando las reglas anteriores, el código obtenido no tiene subexpresiones comunes, no hay instrucciones de copia eliminables localmente, no tiene código inútil y permite obtener todos los órdenes de instrucciones válidos posibles.

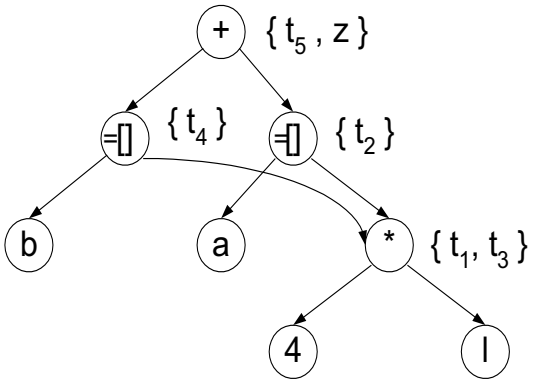
Ejemplo 12:

Código fuente: a: array [1..100] of integer; {talla integer 4 u.m.}
……
z := a[i] + b[i]
……

Bloque básico: (P, {a, i, b}, {z})

P = t1 := 4 * i
t2 := a[t1]
t3 := 4 * i
t4 := b[t3]
t5 := t2 + t4
z := t5

GDA



Código reconstruido:

t1 := 4 * i
t2 := a[t1]
t4 := b[t1]
z := t2 + t4

4. Optimizaciones globales

4.1. Optimización de saltos

El generador de código intermedio produce en muchas ocasiones instrucciones de salto que son fácilmente mejorables, veamos a continuación algunos casos. Observar que las modificaciones propuestas nunca prescinden de la etiquetas, aunque puedan parecer superfluas, es posible que sean destino de otros saltos del programa.

a) GOTO L1 → GOTO L2
….. → …..
L1: GOTO L2 → L1: GOTO L2

b) IF a < b GOTO L1 → IF a < b GOTO L2
….. → …..
L1: GOTO L2 → L1: GOTO L2

c) GOTO L1
L1: ….. ➔ L1: …..

d) IF a < b GOTO L1 ➔ IF a ≥ b GOTO L2
GOTO L2 ➔ L1: …..
L1: …..

e) Supongamos que el único salto a L1 es el que aparece en el código.

GOTO L1		IF a < b GOTO L2
…..		GOTO L3
GOTO L4	➔	…..
L1: IF a < b GOTO L2		GOTO L4
L3:		L3:

4.2. Detección de bucles

Dominantes

Un nodo **d** de un grafo de flujo **domina** a otro nodo **n** (d dom n) si todo camino desde el nodo inicial del grafo de flujo a **n**, pasa por **d**

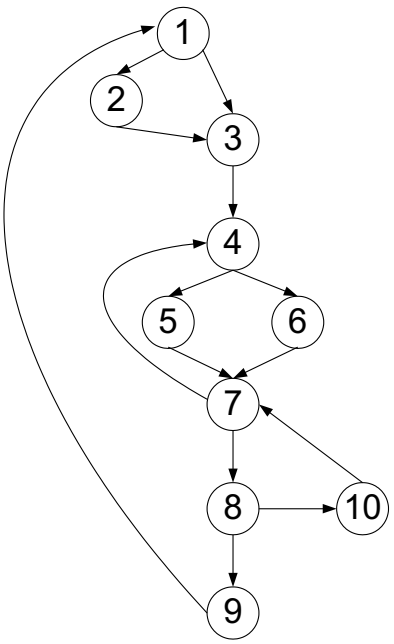
En particular supondremos que todo nodo se domina a si mismo. La relación de dominio es reflexiva, transitiva y establece una relación de orden parcial entre los nodos.

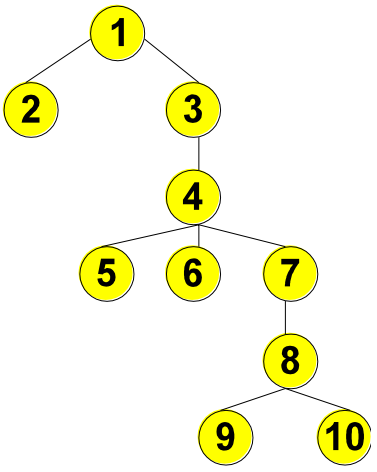
Para representar la información sobre dominadores, se suele utilizar un **árbol de dominación**:

El nodo raíz es el correspondiente al nodo inicial del grafo de flujo, y cada nodo **d** domina solo a sus descendientes en el árbol.

Ejemplo 13:

Las siguientes figuras muestran un grafo de flujo y su correspondiente árbol de dominación.





Bucles naturales

Una aplicación de la información sobre dominadores es facilitar la detección de los bucles de un grafo de flujo objeto de mejora. Por bucles entendemos aquellas estructuras que en algún sentido vuelven a comenzar en el mismo punto y que tienen un único punto de entrada desde fuera de la estructura. Una definición que recoge esta idea de una forma más precisa es la de bucle natural.

Un conjunto de nodos de un grafo de flujo forma un bucle natural si cumplen las siguientes dos propiedades:

1. El bucle debe tener un solo punto de entrada (encabezamiento). Este punto de entrada domina todos los nodos dentro del bucle, o no sería la única entrada al bucle.

2. Debe haber al menos una forma de iterar el bucle, es decir, desde cualquier nodo del bucle debe haber al menos un camino de regreso al encabezamiento.

Podemos detectar los bucles si encontramos las aristas de retroceso: Aristas cuyas cabezas dominen a sus colas. Diremos que n → d es una **arista de retroceso** si el nodo **d** (cabeza) domina al **n** (cola).

Dada una arista de retroceso n → d, llamamos **bucle natural** de la arista, a **d** (encabezamiento del bucle) más el conjunto de nodos que pueden alcanzar **n** sin pasar a través de **d**.

Algoritmo 3.5: Construcción del bucle natural asociado a una arista de retroceso.

Entrada: Grafo de flujo *G* y una arista de retroceso *n ->d*.

Salida: Conjunto de todos los nodos que constituyen el *bucle natural* asociado a *n --> d*.

Método:

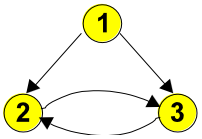
```
procedure inserta(m);
If m no está en bucle then
begin bucle := bucle ∪ {m}; push( m ) end;
    /* programa principal */
pila := vacía;
bucle:= { d };
inserta( n );
while pila no sea vacía do
begin
    m := pop ;
    for cada predecesor inmediato p de m do inserta( p )
end
```

Ejemplo 14: Aristas de retroceso y bucles naturales del grafo del Ejemplo 13.

Arista de retroceso	Nodos del bucle natural
7 → 4	{ 4,5,6,7,8,10 }
10 → 7	{ 7,8,10 }
9 → 1	{ 1,2,3,4,5,6,7,8,9,10 }

El bucle natural 10 →7 consta de los nodos 7, 8 y 10 (alcanzan 10 sin pasar por 7). El bucle natural 9 →1 consta de todos los nodos. (ver camino 10→7→8→9) Aunque pueda parecerlo el conjunto {4, 5, 6, 7 } no constituyen un bucle natural puesto que el nodo 4 no sería el único punto de entrada.

Obsérvese que en el siguiente grafo de flujo no hay ninguna arista de retroceso en consecuencia no hay ningún bucle natural. Aunque exista iteración entre los nodos 2 y 3.



Pre-encabezamiento

Añadimos un nuevo bloque vacío inicialmente, llamado pre-encabezamiento, que solo tiene como sucesor al encabezamiento del bucle, de forma que todas las aristas que entraban al encabezamiento desde fuera del bucle ahora entran al pre-encabezamiento. Las aristas que llegan al encabezamiento pero tienen su origen dentro del bucle, no se modifican.

4.3. Extracción de código invariante

Invariante: cálculos cuyo valor no cambia mientras el control permanece dentro del bucle.

OBJETIVO: Sacar del bucle las expresiones invariantes (tomarán el mismo valor independientemente del número de veces que se ejecute el bucle).

Algoritmo 3.6: Detección de código invariante

Repetir

Marcar las instrucciones cuyos operandos sean constantes, no se definan dentro del bucle, o tengan una sola definición dentro del bucle y ésta ya esté marcada como invariante. Las instrucciones se numerarán respecto del orden de detección.

Hasta no se marque ninguna instrucción

Ejemplo 15

```
(1) a:= 5 + N
(2) i := i +1
(3) b := a * 4
(4) arr[i] := b
(5) if a < N goto (1)
(6) x:= t
```

En este ejemplo marcaríamos primero la instrucción (1) y a continuación la (3).

No todas las instrucciones invariantes pueden ser extraídas del bucle de forma que el comportamiento del programa objeto no cambie.

Ejemplo 16:

```
(1) a:= 5
(2) if b > 2 goto (4)
(3) a := 4
(4) b := 2  * b
(5) if b < N goto (2)
(6) x := a
```

En este bucle la instrucción (3) es invariante pero no puede ser extraída del bucle sin que el comportamiento del código cambie. Si sacáramos la instrucción (3) fuera del bucle, al pre-encabezamiento, para los valores N = 2 y b = 3, **x** terminaría valiendo 4, en cambio, con el orden inicial **x** valdría 5.

Algoritmo 3.7: Extracción de código invariante

Para toda Seleccionar las instrucciones según su orden de marcado. Sea I_i una instrucción marcada que defina **x** y cumpla:

i) Está en un bloque que domina todas las salidas del bucle, (o **x** no está activa a la salida del bucle).

ii) **x** no se define en otra parte del bucle.

iii) Todos los usos de **x** en el bucle solo pueden ser alcanzados por esa definición de **x**.

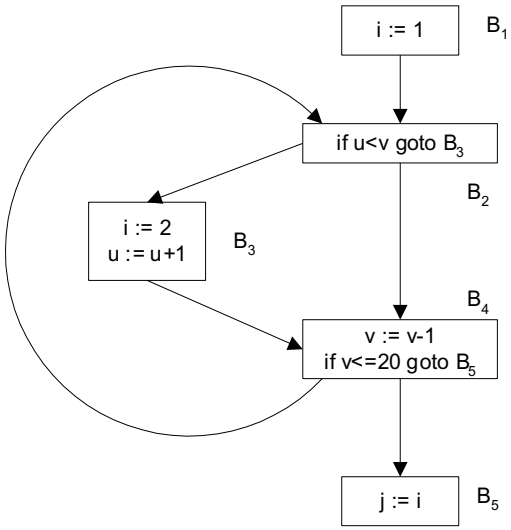
iv) Los valores de los operandos son constantes o se definen ya en el pre-encabezamiento.

hacer Extraer I_i al final del pre-encabezamiento.

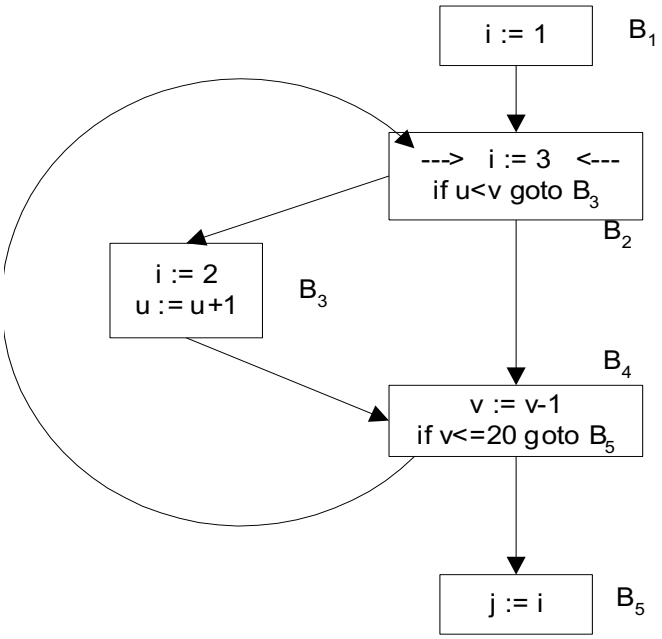
En el Ejemplo 15 las instrucciones quedarían de la siguiente forma:

```
(0) a:= 5 + N
(1) b := a * 4
(2) i := i +1
(4) arr[i] := b
(5) if a < N goto (2)
(6) x:= t
```

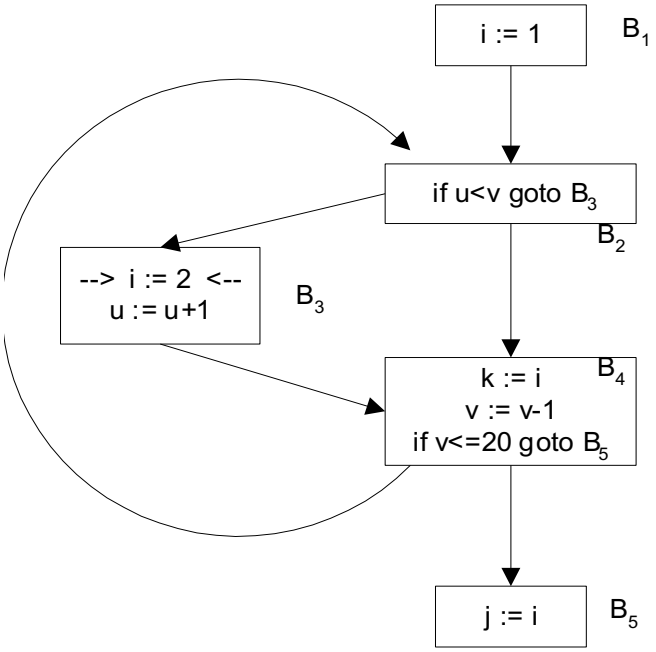
En los siguientes ejemplos se muestra la necesidad de imponer las restricciones i) a iii) del algoritmo de extracción de código.



No cumple la condición i)



No cumple la condición ii)



No cumple la condición iii)

4.4. Reducción de intensidad en variables de inducción

Una variable **x** se denomina **variable de inducción** de un bucle L, si durante la ejecución del bucle, cada vez que la variable x cambia de valor en el bucle, se incrementa o decrementa por una constante.

Una **variable básica de inducción** es aquella variable i cuya definición en el bucle es de la forma i := i ± c, donde c es una constante (o al menos un invariante del bucle). En adelante supondremos que hay una única definición de **i** en el bucle. Los algoritmos siguientes se pueden extender al caso de más de una definición.

El conjunto de variables j cuya relación con **i** en el bucle es de la forma j := i c + d (c y d son invariantes del bucle) forma la familia de variables de inducción derivadas de i, a este conjunto de variables en las que incluiremos a la propia **i** lo denotaremos por **Familia (i)**. Para poder establecer la relación lineal entre las variables de inducción y la variable básica utilizaremos la terna (i, c, b) cuando j = c*i + b. La variable j vendrá identificada por la terna (i,c,d). Puesto que el código vendrá en código tres direcciones, la relación directa entre las variables obligará a que c sea 1 o d sea cero. Pero en general, si la relación no es directa no se cumplirá lo anterior.

Ejemplo 17:

```
(0) i := i + 3
(1) t1 := 2 * i
(2) j := t1 + 5
(3) t2 := 3 * j
(4) k := t2 + 6
(5) m := a[j]
(6) a[k] := m
(7) if i < N goto (0)
```

En este ejemplo la variable básica de inducción es **i**, mientras que t1, representada por (i,2,0); j por (i,2,5); t2 por (i,6,15); y k por (i,6,21) son variables de inducción de la Familia(i). Donde j = 2i + 5 y k = 6i + 21. La variable de inducción básica i lleva asociado el triple (i,1,0).

En el ejemplo vemos que si una variable j pertenece a la Familia(i), y hay otra variable k cuya definición en el bucle es de la forma k := d*j + e, entonces k también pertenece a la Famila(i) ya que k := c*d*i + d*b + e.

Algoritmo3.8: ***Detección de variables de inducción***

Entrada Bucle con información sobre el alcance de las variables y cálculos que son invariantes dentro del bucle

Salida Familias de variables de inducción y sus ternas asociadas

Método

1) Encontrar todas las variables básicas de inducción **i**.

   Asociar a cada una el triple (i, 1, 0).

2) Buscar las variables k con **una sola definición** dentro del bucle de la forma:

   k := j*b, k := b*j, k := j/b, k := j±b, k := b±j

   donde *b* es una cte. (o invariable dentro del bucle) y *j* una variable de inducción.

   Si j es variable de inducción básica

   entonces k está en la familia de j

   asociar a k la terna (j, b, 0) [(j, 1/b, 0), (j, 1, ±b)]

   sino /* j pertenece a familia(i) */

   si (¬∃ definición de i entre la definición de j y la de k) ∧

   (¬∃ ninguna definición de j fuera del bucle que alcanza la definición de k)

   entonces k está en la familia de **i**

   asociar a k la terna (i, e, f) donde e y f se calculan a partir de la terna de j y de la definición de k.

Para el código del Ejemplo 17 las ternas serían:

i (i, 1, 0)
t1 (i, 2, 0)
j (i, 2, 5)
t2 (i, 6, 15)
k (i, 6, 21)

Ejemplo 18: En este ejemplo se puede observar la necesidad de las dos condiciones del algoritmo: (¬∃ definición de i entre la definición de j y la de k) ∧ (¬∃ ninguna definición de j fuera del bucle que alcanza la definición de k): la variable **k** no pertenece a Familia(i), aunque si la **j**.

10 j:=3*i	10 i:=1
11 i:=i+1	11 j:=1
12 k:=2*j	12 k:=2*j
13 if i < N goto 10	13 j:=3*i
	14 i:=i+1
	15 if i < N goto 12

Una vez encontradas las familias de variables de inducción, se modifican las instrucciones que calculan las variables de inducción, para que empleen sumas o restas en lugar de multiplicaciones. Con esto conseguimos una reducción de intensidad en las operaciones de cálculo de las variables de inducción.

Algoritmo 3.9: Reducción de intensidad en variables de inducción.

Entrada Bucle con las definiciones de alcance de las variables y las familias de variables de inducción

Salida Bucle mejorado.

Método

Para toda variable de inducción básica **i** hacer

Para todo variable de inducción **j** de terna (i, c, d).

hacer

1) Crear una variable temporal **s**

2) Sustituir las asignaciones a j, por j := s {a partir de aquí los usos de s = usos de j}

3) Si en el bucle la asignación a la variable básica es i := i + n

añadir a continuación de esta asignación s := s + c*n /* c*n cte.*/

añadir **s** a Familia(i) /* terna (i, c, d) */

4) Poner la inicialización de **s** al final del pre-encabezamiento:

```
s := c * i          /* si c es 1 poner s := i */
s := s + d          /* omitir si d es 0 */
```

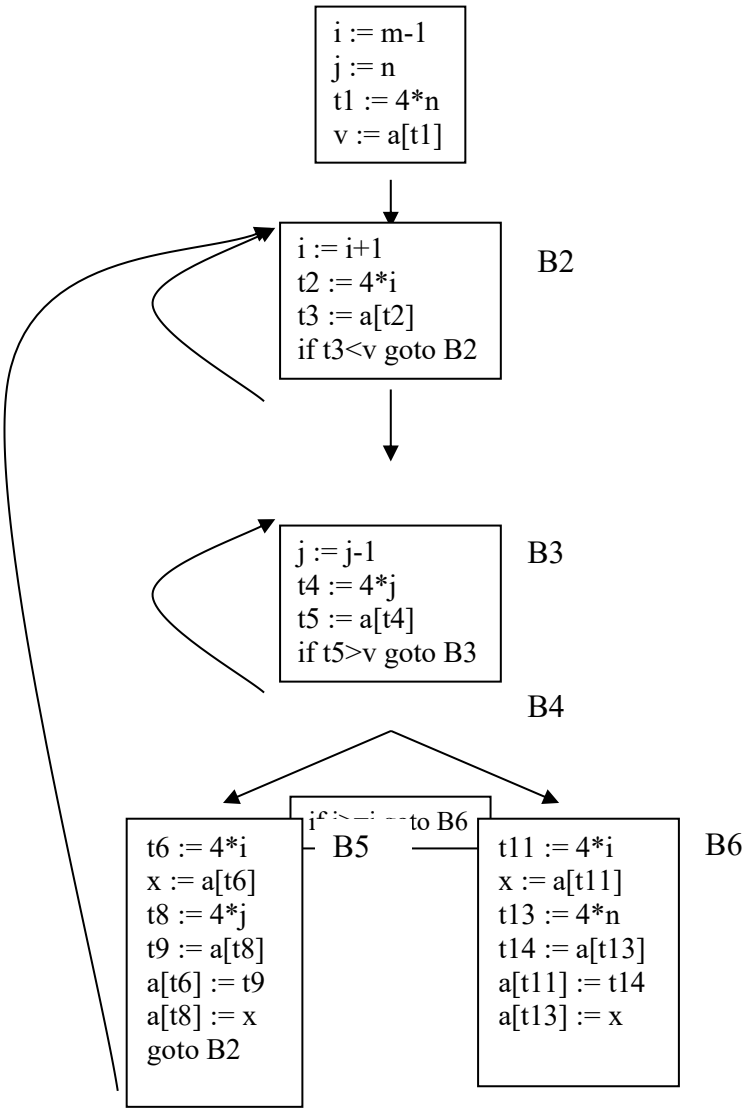
Si c no es la unidad la instrucción s := s + c*n no es código tres direcciones pero como c y d son invariantes del bucle el producto se puede realizar en el pre-encabezamiento.

Ejemplo 19: La aplicación del algoritmo de reducción de intensidad sobre las variables de inducción del código del Ejemplo 17 producirá:

```
        s1 := 2 * i
        s2 := 2 * i
        s2 := s2 + 5
        s3 := 6 * i
        s3 := s3 + 15
        s4 := 6 * i
        s4 := s4 + 21
```

```
(0)     i := i + 3
        s4 := s4 + 18
        s3 := s3 + 18
        s2 := s2 + 6
        s1 := s1 + 6
(1)     t1 := s1
(2)     j := s2
(3)     t2 := s3
(4)     k := s4
(5)     m := a[j]
(6)     a[k] := m
(7)     if i < N goto (0)
```

Ejemplo 20:



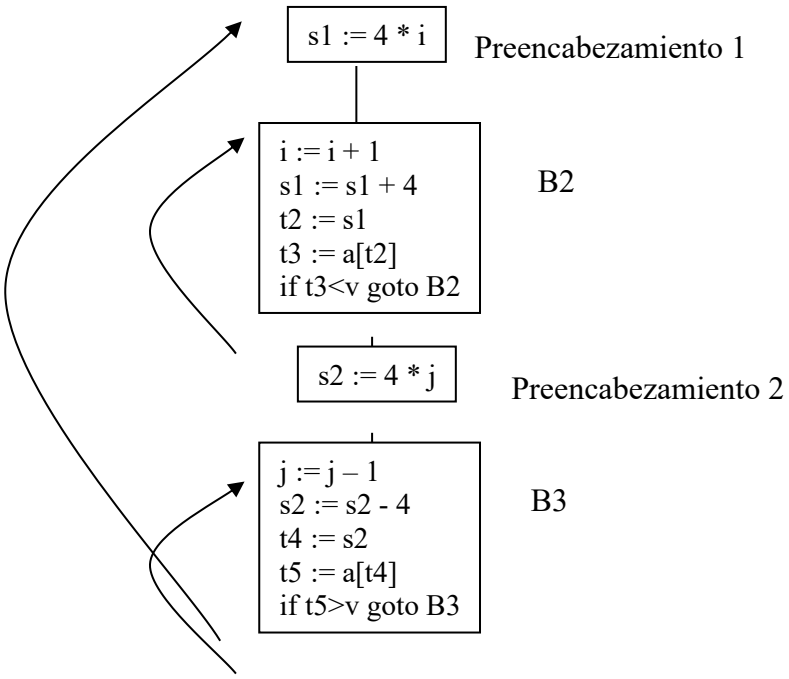
Bucles B2, B3 y B2-B3-B4-B5

En el bucle B2 la variable de inducción básica es **i** y la variable t2 ∈ Familia(i), teniendo como terna a (i, 4, 0).

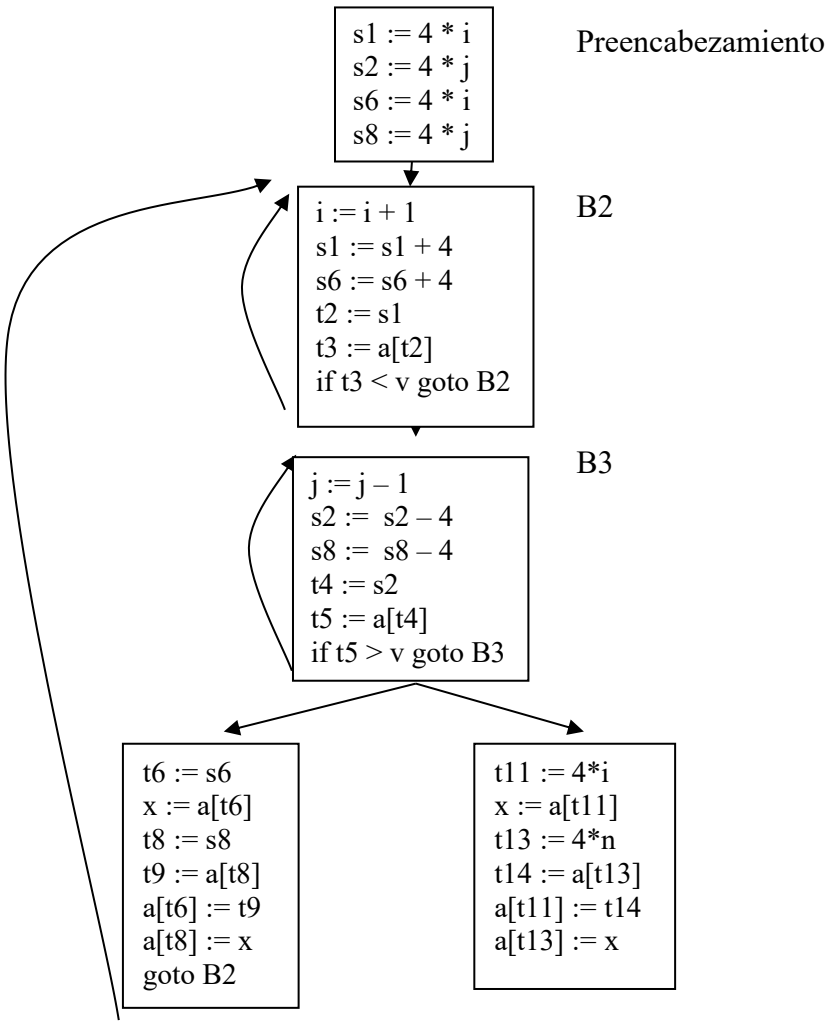
En el bucle B3 la variable de inducción básica es **j** y la variable t4 ∈ Familia(j), y tiene como terna a (j, 4, 0).

En el bucle B2-B3, B4, B5 las variables de inducción básica son **i** y **j** teniendo que t2, t6 ∈ Familia(i) y t4, t8 ∈ Familia(j).

Si aplicamos el algoritmo de reducción de intensidad a los bucles B2 y B3 obtendremos:



Si aplicáramos el algoritmo directamente al bucle B2, B3, B4 y B5 obtendríamos:



4.5. Eliminación de variables de inducción

Después de la reducción de intensidad, a menudo se observa que algunas variables de inducción solo se usan en su propia definición o que se usan a lo sumo para realizar algún test. En estos casos se puede/n suprimir alguna/s variable/s de inducción.

En el Ejemplo 20 si a la salida de los bucles B2 y B3 no estuvieran activas ni **i** ni **j** ambas se usarían sólo en su propia definición.

En el bucle B2, B3, B4 y B5 ambas se usan en su propia definición y en el test de B4.

Para la eliminación de estas variables de inducción que ya no son necesarias, se aplica, como continuación del algoritmo de reducción de intensidad, el algoritmo de eliminación de variables de inducción que se presenta a continuación.

Algoritmo 3.10: Eliminación de variables de inducción

Entrada Bloques básicos que forman el bucle.

Salida Bucle mejorado

Método

0) Si **i** es v.i. no activa a la salida del bucle Y (solo se usa dentro del bucle en su propia definición) O (no se usa en el bucle y se define en una instrucción de copia) entonces Eliminar su definición.

Eliminar también las definiciones de estas variables incorporadas al pre-encabezamiento.

Repetir 0) mientras quede alguna v.i. que cumpla esa condición.

1) Para toda v. básica de i. **i** que se usa sólo para calcular v.i. de su familia y/o saltos condicionales.

hacer

Elegir una j ∈ Familia(i)

/* con triple (i, c, d) lo más simple */

modificar cada comprobación en que aparezca i para utilizar j

Ej.: if i oprel x goto B , donde x **no** es una variable de inducción y **c** suponemos que es positiva se sustituye por:

```
r := c * x              /* r := x si c es 1 */
r := r + d              /* se omite si d es 0 */
if j oprel r goto B     /* r temporal */
```

En el caso, posible, de que **x** sea constante dentro del bucle las instrucciones que evalúan **r** pueden pasar al pre-encabezamiento.

Si x fuera una variable de inducción de terna (i, c1, d1) y c <> c1 o d <> d1 puede no valer la pena la transformación.

finpara

para toda v.i. cuyo único uso sea su propia definición

hacer borrar su definición.

finpara

Aplicar otra vez el paso 0)

2) Para toda v.i. **j** para la que el algoritmo de reducción de intensidad introdujo una instrucción j := s

hacer

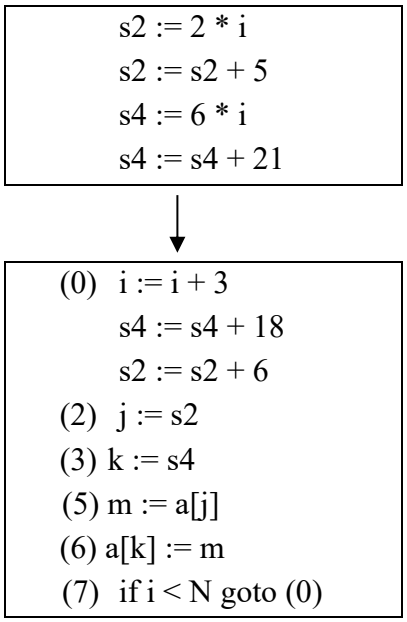
a) Comprobar que no se define s entre j := s y cualquier uso de j

b) Sustituir usos de j por usos de s

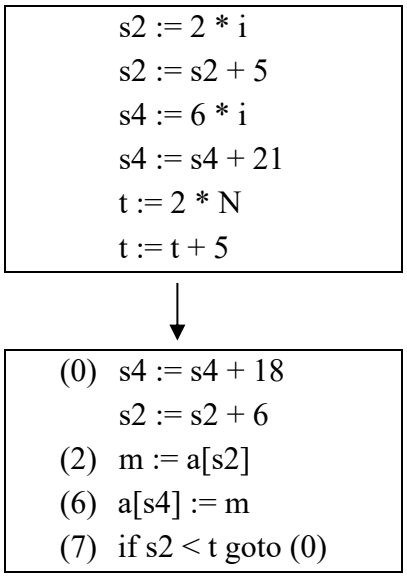
c) Borrar j := s

finpara

En el Ejemplo 17 el paso 0) del algoritmo producirá, bajo el supuesto de que t1 y t2 no tienen otros usos por tratarse de temporales:

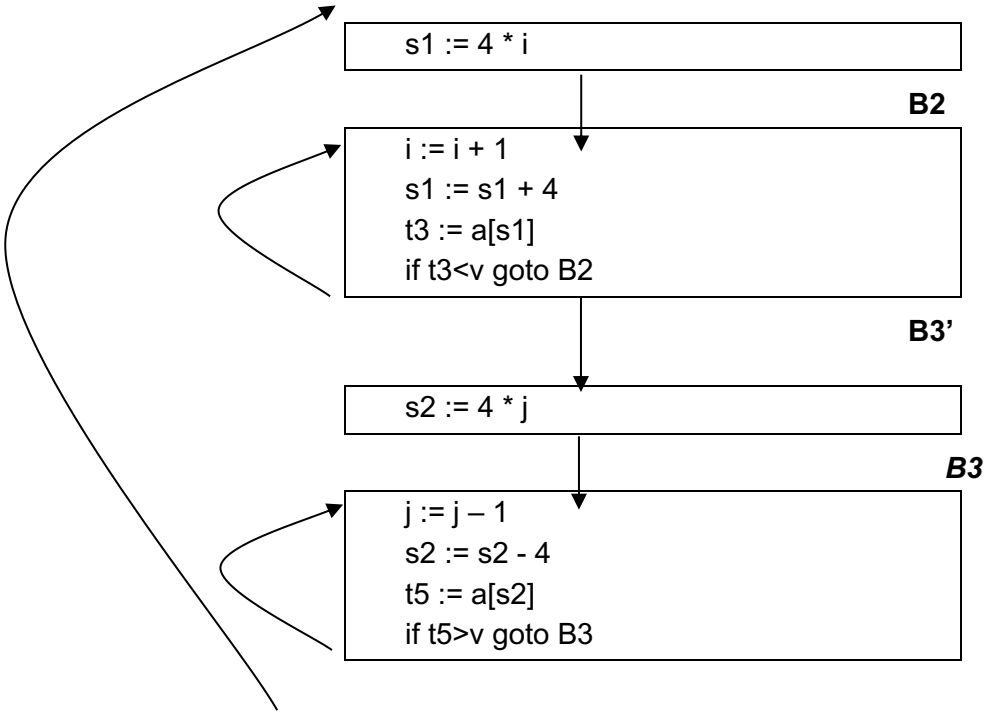


Al finalizar el algoritmo quedará:



En el Ejemplo 20, el algoritmo anterior aplicado a los bucles B2 y B3 por separado, producirá:

B2'



Nótese que las variables de inducción básicas no pueden ser eliminadas de los bucles B2 ni B3, puesto que a la salida de estos bucles ambas variables están activas. Por otro lado si aplicamos los algoritmos anteriores al bucle B2', B2, B3', B3, B4 y B5 las únicas variables de inducción son la **i** y la **j**. En cambio, ni **s1**, ni **s2** son variables de inducción puesto que están definidas en el bucle dos veces cada una de ellas.

Ejemplo 21:

Código fuente →
```
var a: array [1..20, 1..10] of integer; {talla de un entero 2 u.m.}
…
while a[i, k] < M do k := k – 1;
…
```

Suponemos que las variables **k** y **i**, y el array **a[]** se usan después de esta instrucción. También suponemos, por simplicidad, que para el array su posición relativa en memoria se ha obtenido incluyendo la parte constante de la función de acceso.

Código intermedio: →
```
100 t1:= i * 10
101 t2 := t1 + k
102 t3 := t2 * 2
103 t4 := a[t3]
104 t5 := M
105 if t4 < t5 goto 107
106 goto 111
107 t6 := 1
108 t7 := k – t6
109 k := t7
110 goto 100
111
```

Los bloques básicos de este código son:

B1: 100 – 105
B2: 106
B3: 107 – 110
B4: 111 …

Las optimizaciones locales, construcción del GDA y reconstrucción del código, aplicadas a los bloques B1 y B3 producirán:

Código intermedio: →

```
100 t1:= i * 10
101 t2 := t1 + k
102 t3 := t2 * 2
103 t4 := a[t3]
104 if t4 < M goto 106
105 goto 108
106 k := k – 1
107 goto 100
108
```

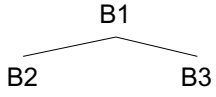
Un análisis de los saltos correspondientes a las instrucciones 104 y 105 permiten una mejora de código de acuerdo al caso 4.1 d). Después de esta modificación tendremos:

```
100 t1:= i * 10
101 t2 := t1 + k
102 t3 := t2 * 2
103 t4 := a[t3]
104 if t4 ≥ M goto 107
105 k := k – 1
106 goto 100
107
```

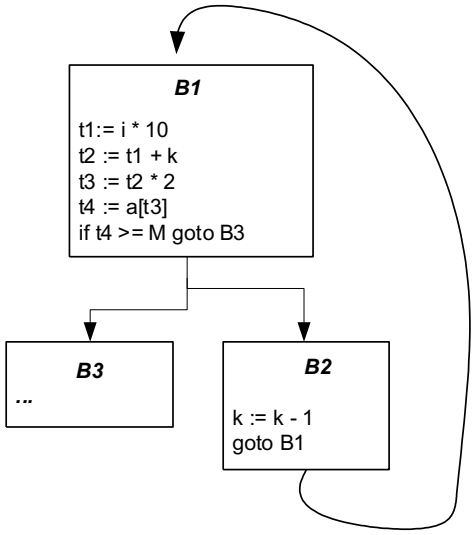
Los nuevos bloques básicos de este código son:

B1: 100 – 104
B2: 105 – 106
B3: 107 …

El nuevo grafo de flujo y el árbol de dominación son:



La única arista de retroceso es la B2 → B1 y el bucle natural asociado: {B1, B2}.



Código invariante del bucle {B1, B2}:

t1 := i * 10 podemos pasarlo al pre-encabezamiento, puesto que **i** y 10 son invariantes del bucle. Además t1 cumple las condiciones suficientes para ser extraído del bucle.

Esto es equivalente a modificar únicamente la instrucción 106:

106: goto 100 a 106: goto 101.

Variable básica de inducción y terna asociada: k → (k, 1, 0)
Variables de inducción y ternas asociadas: t2 → (k, 1, t1)
t3 → (k, 2, 2 * t1)

El algoritmo de reducción de intensidad introducirá las variables s2, s3 quedando:

t2, s2 → (k, 1, t1)
t3, s3 → (k, 2, 2*t1)

100 t1:= i * 10

```
101 s2 := k
102 s2 := s2 + t1
103 s3 := 2 * k
104 t10 := 2 * t1
105 s3 := s3 + t10
106 t2 := s2
107 t3 := s3
108 t4 := a[t3]
109 if t4 ≥ M goto 114
110 k := k – 1
111 s3 := s3 – 2
112 s2 := s2 - 1
113 goto 106
114
```

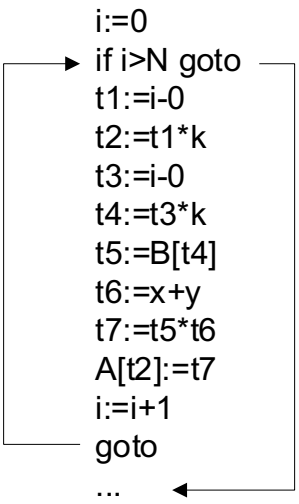
Eliminación de variables de inducción.
Paso 0) del algoritmo: la variable t2 no está activa a la salida del bucle, por lo tanto se puede eliminar la instrucción 106. Lo mismo es aplicable a s2 := s2 - 1. La variable **k** no se puede eliminar puesto que está activa a la salida del bucle. Así mismo, eliminamos del pre-encabezamiento las instrucciones 101 y 102. El paso 1) no es aplicable en este ejemplo puesto que no hay otros usos de la variable **k** en el bucle. El paso 2) del algoritmo eliminará la instrucción 107 y modificará la 108, quedando:

```
101 t1:= i * 10
102 s3 := 2 * k
103 t10 := 2 * t1
104 s3 := s3 + t10
105 t4 := a[s3]
106 if t4 ≥ M goto 110
107 k := k - 1
108 s3 := s3 - 2
109 goto 105
110
```

Compárese este código con el inicial: dentro del bucle hay 3 instrucciones de copia menos y 2 productos menos. Inicialmente en cada iteración se ejecutaban 10 instrucciones y en el código mejorado se han reducido a 5 instrucciones.

Ejercicios.

*__1.-__ Dado el siguiente fragmento de código intermedio:



a) Obtener el conjunto de bloques básicos y construir el grafo de flujo asociado.

b) Indicar la relación de dominación entre nodos, y calcular todas las aristas de retroceso y los bucles naturales asociados.

c) Aplicar al código intermedio, justificando cada paso, las técnicas de optimización de eliminación de subexpresiones comunes, propagación de copia, eliminación de código inactivo, extracción de código invariante y reducción de intensidad y/o eliminación de variables de inducción.

*__2.-__ Dado el siguiente fragmento de código intermedio

```
(102)  t3 := t2 * 2
(103)  t4 := a[t3]
(104)  t5 := M ;
(105)  if t4 < t5 goto 107
(106)  goto 110
(107)  t6 := k - 1
(108)  k := t6
(109)  goto 100
```

a) Determina los bloques básicos, el grafo de flujo y el árbol de dominación correspondiente al código intermedio generado.
b) Optimiza el código intermedio usando transformaciones locales. Para ello, construye el GDA asociado, y no olvides indicar en cada mejora la técnica que se utiliza.
c) Indica las aristas de retroceso y sus bucles naturales asociados, las variables de inducción y sus familias de variables de inducción (con las ternas asociadas)
d) Termina de optimizar el código intermedio usando transformaciones globales. Supón que los únicos usos de la variable “k” son los que aparecen en el fragmento de código.

*__3.-__ Considera el siguiente bloque de código intermedio:

```
(10) total := 0
(11) i := 1
(12) t3 := i
(13) t4 := t3 * 2
(14) t5 := B[t4]
(15) t6 := i
(16) t7 := t6 * 4
(17) t8 := A[t7]
(18) t9 := t5 + t8
(19) t10 := t9 + total
(20) total := t10
(21) t11 := i
(22) t12 := t11 + 1
(23) i := t12
(24) if i <> 11 goto 12
```

a) Determina los bloques básicos, el grafo de flujo y construye los GDAs de cada uno de los bloques básicos. Reconstruye el código optimizado a partir de los GDAs.

b) Tras localizar los bloques básicos que forman el bucle, indica las variables de inducción y sus familias de variables de inducción (con las ternas asociadas).

c) Aplica los algoritmos de reducción de intensidad y de eliminación de variables de inducción.

***4.-** Considera el siguiente bloque de código intermedio donde los destinos de los saltos se indican mediante etiquetas explícitas. Supón que las variables **j**, y **k** no están activas a la salida del bloque de código.

```
        t1 := 0
        j  := t1
        t2 := 9
L1:     if j > t2 goto L3
        t3 := 0
        k  := t3
        t4 := 9
L2:     if k > t4 goto L4
        t5 := 5
        t6 := 10 * t5
        t7 := t6 + k
        t8 := t7 * 2
        t9 := b[t8]
        t10:= 5
        t11:= 10 * t10
        t12:= t11 + j
        t13:= t12 * 10
        t14:= t13 + k
        t15:= t14 * 2
        t16:= a[t15]
        t17:= t9 + t16
        t18:= 5
        t19:= 10 * t18
        t20:= t19 + k
        t21:= t20 * 2
```

```
        b[t21]:= t17
        k   := k + 1
        goto L2
L4:     j   := j + 1
        goto L1
    L3:
```

a) Obtener el grafo de flujo
b) Obtener el árbol de dominación
c) A partir de la construcción del GDA, aplicar las optimizaciones locales adecuadas.
d) Empleando los algoritmos estudiados, aplica las optimizaciones globales adecuadas.

***5.** Dado el siguiente bloque básico, aplicad las optimizaciones locales mediante la construcción de su GDA y la reconstrucción del código intermedio. Suponed que, a la salida del bloque básico, solo están activas las variables a, b, d, e y f .

$t_1 := a$
$t_2 := 5$
$a := t_1 + t_2$
$t_3 := 5$
$t_4 := b * t_3$
$b := t_4$
$t_5 := t_3 * 1$
$t_6 := t_3 * t_5$
$d := t_6$
$e := a + t_2$
$t_7 := 1$
$t_8 := b - t_7$
$f := e - d$
… … …

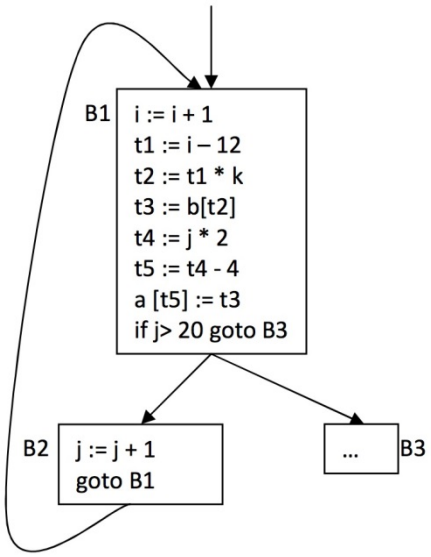
***6.** Dado el siguiente fragmento de código intermedio, y suponiendo que todas las variables están activas en el resto de programa, construid el grafo de flujo, localizad los bucles y, para cada uno de ellos, extraed el código invariante y calculad los triples de sus variables de inducción.

```
09   ...
10   k := 10 * d
11   c := k − 5
12   a := c + 2
13   j := c − 2
14   if c > 5 goto 19
15   e := e + c
16   a := a + 1
17   if a < 20 goto 14
18   goto 22
19   j := j + 2
20   f := f + j
22   if j < 30 goto 14
```

```
22    c := c + k
23    if c < 10 goto 12
24    z := k * c
25    …
```

***7.** Dado el fragmento de código intermedio representado en la siguiente figura y, considerando que a la salida de este bucle solo están activas las variables a y b.

a) Aplicad el algoritmo de reducción de intensidad sobre las variables de inducción.
b) Aplicad el algoritmo de eliminación de variables de inducción al código resultante del apartado anterior.



***8.-** Dado el siguiente fragmento de código intermedio

```
(100)  i := 5
(101)  j = 20
(102)  sum := 0
(103)  t1 := j * 4
(104)  t3 := N * 6
(105)  t4 := t1 + t3
(106)  t5 := B[t4]
(107)  sum := sum + t5
(108)  j := j - 2
(109)  if i>20 goto 115
(110)  t6 = i * 2
(111)  t7 := A[t6]
(112)  sum := sum + t7
(113)  i := i + 2
(114)  goto 103
(115)
```

a) Determina los bloques básicos que forman el/los bucle/s. Extrae el código invariante. Indica

las variables de inducción y sus ternas.
b) Aplica el algoritmo de reducción de intensidad.
c) Aplica el algoritmo de eliminación de variables de inducción. Supón que a la salida del bucle solo está activa la variable sum

9.- Dado el siguiente fragmento de código, y suponiendo que a la salida del bloque básico solo están activas las variables i, j, x, y, y res. Construye el GDA y, aplicando las optimizaciones locales posibles, reconstruye el código intermedio.

```
t2 := 15
t3 := t2 * 1
i:= t3
t5:= i*x
t6:= A[t5]
t7 := 0
t8 := 15 + t7
t9 := t8 * 1
j:= t9
t10:= j*y
t11:= A[t10]
t12 = 2
t13:= x + t12
x := t13
t16 =  t6 + t11
res = t16
```

10.- Optimizar el código intermedio del siguiente bloque. Construir el GDA correspondiente y a partir de él, reescribir el código.

(BB, {a, b, x}, {j, a, y})

```
BB =  {x := a + b
       j := 5 * a
       j := x + j
       y := a + b
       a := x
       j := j + a}
```

b) Realiza las mismas operaciones pero sustituyendo la última línea por la instrucción

j := a + b.

11.- Dado el siguiente fragmento de ETDS (solo se muestra la parte de generación de código intermedio):

S → **repeat** S_1 **until** B		S.aux := SI; CompletaLans(B.lf, S.aux); CompletaLans(B.lv, SI);
→ **id**		RS.ident := ident;

		RS	

RS	→	**:=** E	ObtenerTds (RS.ident, RS.pos); Emite(RS.pos **:=** E.pos);
RS	→	**[** E_1 **]:=** E_2	ObtenerTds (RS.ident, RS.orig, RS.tallaelem); RS.des := CreaVarTemp; Emite(RS.des **:=** E_1.pos ***** RS.tallaelem); Emite(RS.orig **[**RS.des**]** **:=** E_2.pos);

B	→	E_1 **oprel** E_2	B.lv := CreaLans(SI); Emite (**if** E_1.pos **oprel** E_2.pos **goto** --); B.lf := CreaLans(SI); Emite (**goto** --);

E	→	**id**	RE.ident := ident;
		RE	E.pos := RE.pos;
E	→	**cte**	E.pos:=CreaVarTemp; Emite(E.pos **:=** num);
E	→	E_1 **op** E_2	E.pos:= CreaVarTemp; Emite (E.pos **:=** E_1.pos **op** E_2.pos);

RE	→	**[** E **]**	ObtenerTds (RE.ident, RE.orig, RE.tallaelem); RE.des := CreaVarTemp; Emite(RE.des **:=** E.pos ***** RE.tallaelem); RE.pos := CreaVarTemp; Emite(RE.pos **:=** RE.orig **[**RE.des**]**);
RE	→	**ε**	ObtenerTds (RE.ident, RE.pos);

a) Considerando que los índices de los arrays están definidos entre [0..N] y que la talla de cada uno de sus elementos es una constante "k". Obtén el código intermedio que producirá el ETDS anterior para el siguiente fragmento de programa:

```
i := 0 ;
repeat A[i] := A[i+1] * i;    i:= i + 2;
until i>= N-1 ;
```

b) Determina los bloques básicos, el grafo de flujo y construye los GDAs de cada uno de los bloques básicos. Reconstruye el código optimizado a partir de los GDAs.

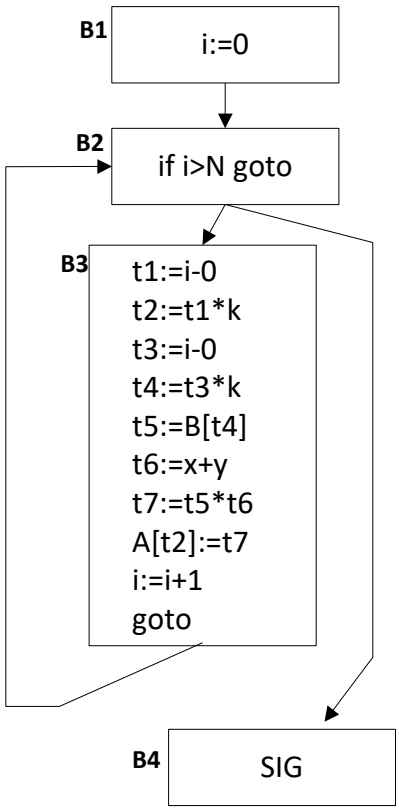
c) Tras localizar los bloques básicos que forman el bucle, indica las variables de inducción y sus familias de variables de inducción (con las ternas asociadas).

d) Aplica los algoritmos de reducción de intensidad y de eliminación de variables de inducción.

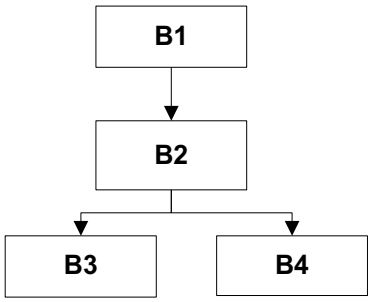
Soluciones de los ejercicios seleccionados.

1.-

a) Bloques básicos y grafo de flujo.

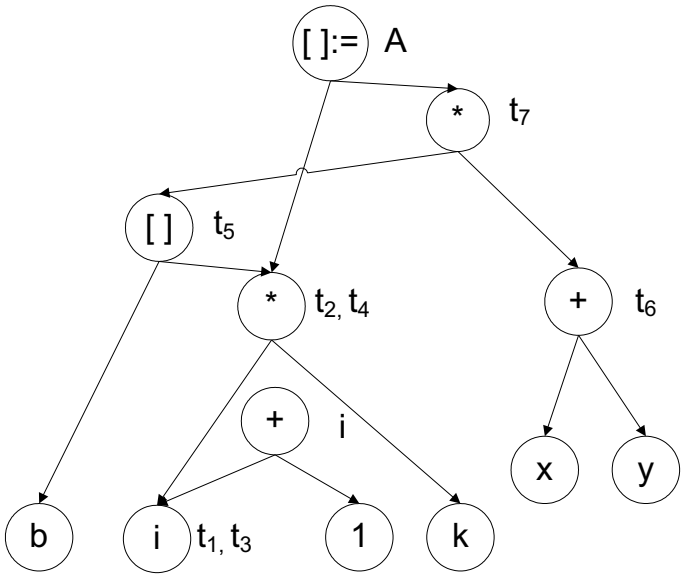


b) Árbol de dominación:

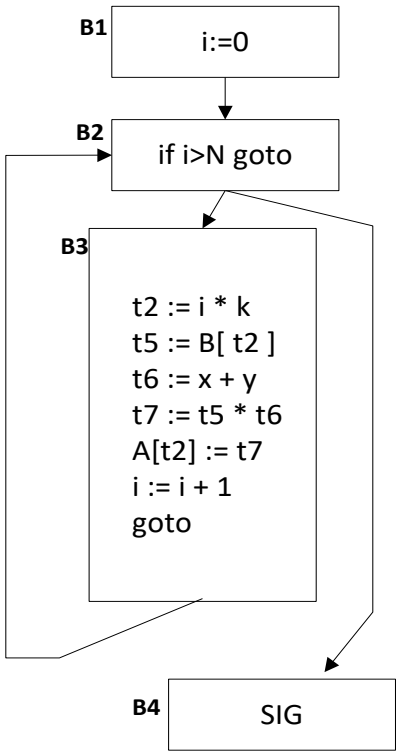


Arista de retroceso B3 → B2
Bucle Natural: B2, B3

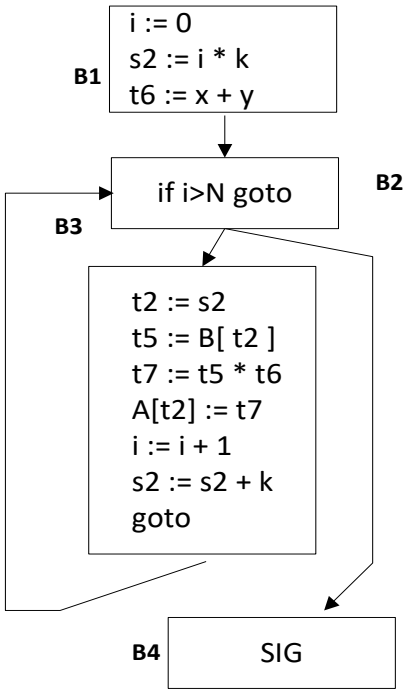
c) A partir de la identidad algebraica i – 0 = i obtendríamos: t1 := i y t3 := i. El GDA asociado al bloque **B3** resultante sería:



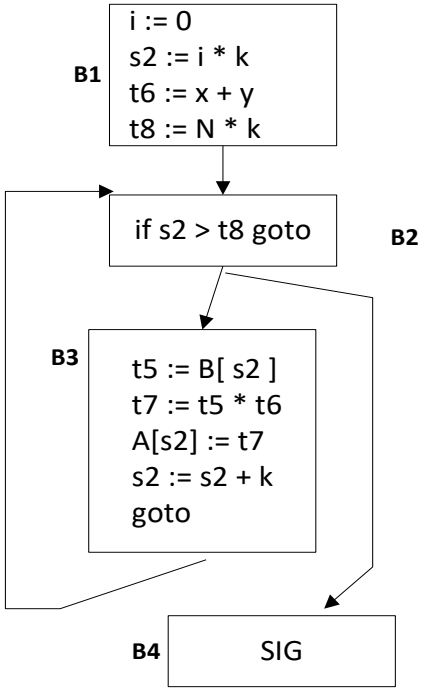
La instrucción a[t2] := t7 (asignación a un elemento de un array) se representa en el GDA mediante un nodo etiquetado con []=. Se puede observar que se trata de un nodo distinto al que usa la etiqueta =[] (uso de un elemento de un array). La reconstrucción del código produciría:



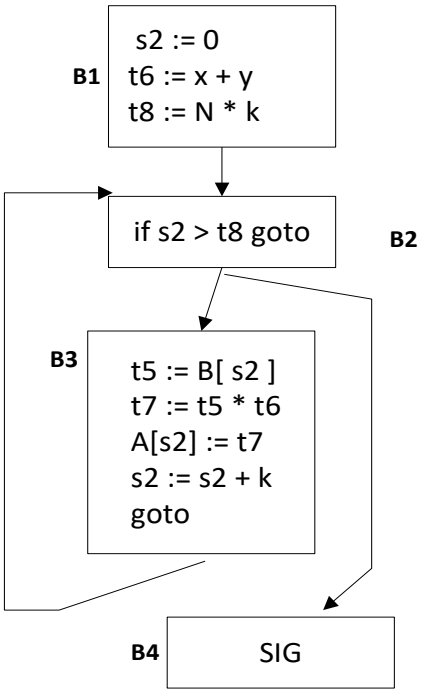
La instrucción t6 := x + y es invariante dentro del bucle y reúne las condiciones para ser extraída fuera del bucle. La extracción de código invariante junto con la reducción de intensidad de las variables de inducción - variable de inducción básica **i** (i, 1, 0) y la única otra variable de la familia(i) es **t2** (i, k, 0)- permitiría obtener:



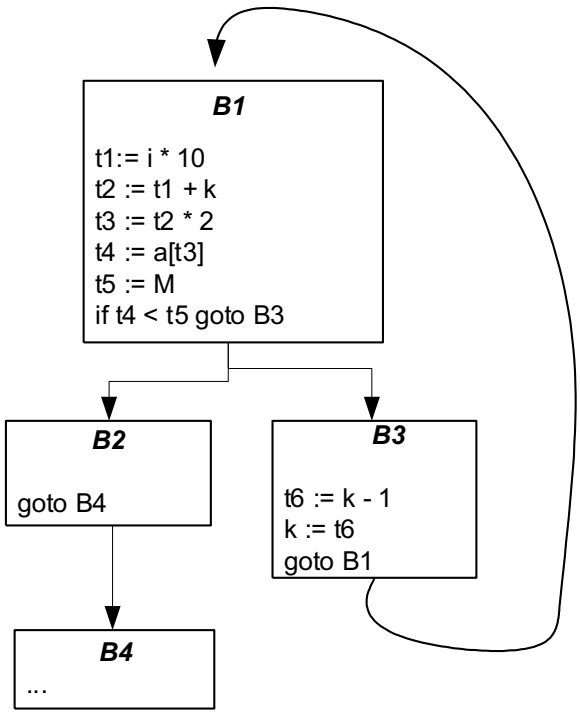
Finalmente, el algoritmo de eliminación de variables de inducción produciría:



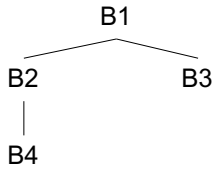
Finalmente se puede realizar una optimización local en **B1**, propagación de copia y transformación algebraica, quedando:



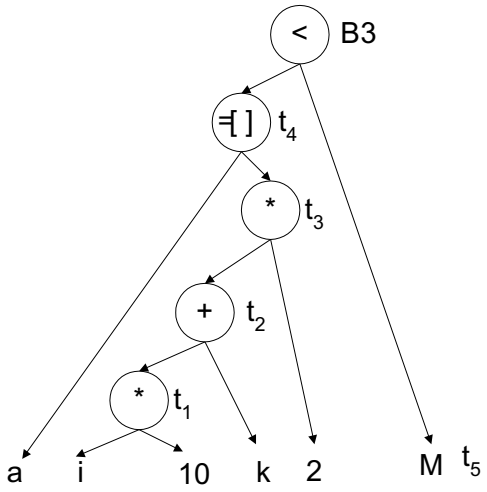
2.- a)

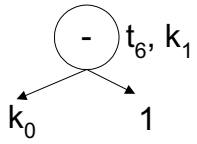


Árbol de dominación:



b) GDA B1 GDA B3





Al reconstruir el código a partir de los dos GDA, se puede observar que se ha realizado una propagación de copia.

B1

```
t1:= i * 10
t2 := t1 + k
t3 := t2 * 2
t4 := a[t3]
if t4 < M goto B3
```

B3

```
k := k - 1
goto B1
```

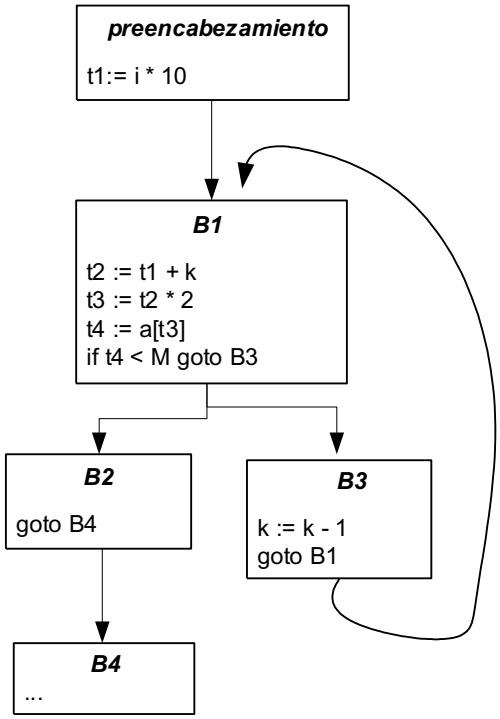
c) Arista de retroceso: B3 → B1

Bucle natural asociado: B3, B1

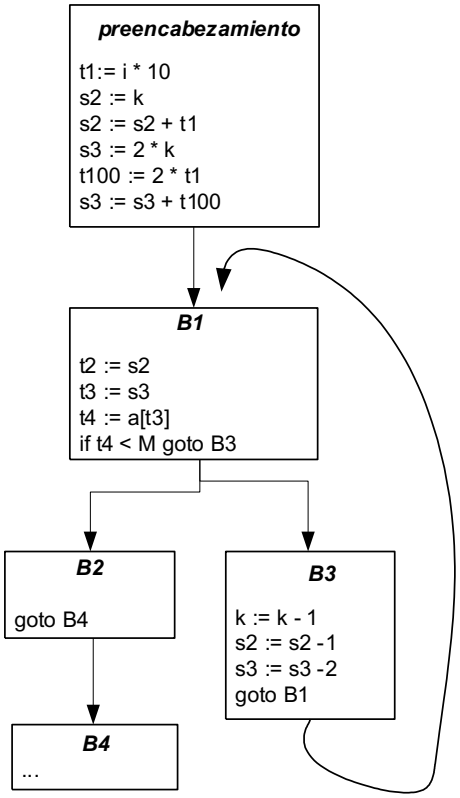
Variable básica de inducción: *k* siendo su terna (k,1,0)

Variables de inducción: *t2* siendo su terna (k, 1, t1) y *t3* con la terna (k, 2, 2*t1)

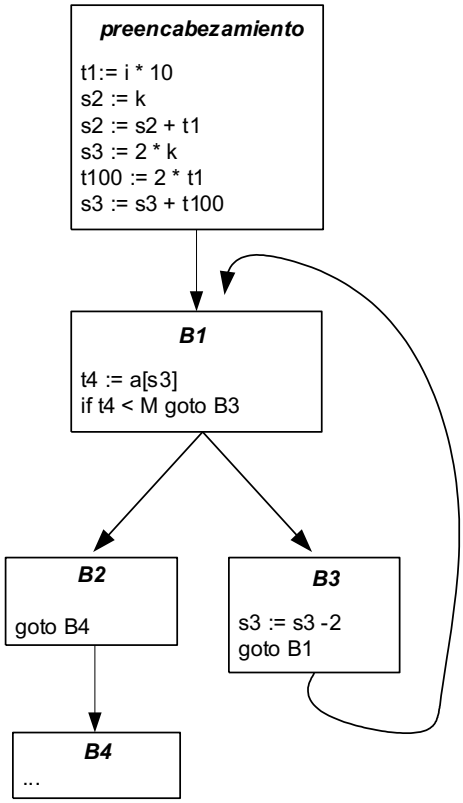
d) Al algoritmo de detección de código invariante aplicado sobre el bucle B1, B3 detecta la instrucción *t1 := i * 10*, que se extrae al preencabezamiento.



Aplicando el algoritmo de reducción de intensidad sobre las variables *t2* y *t3*, para las que se crean las variables temporales *s2* y *s3* respectivamente:

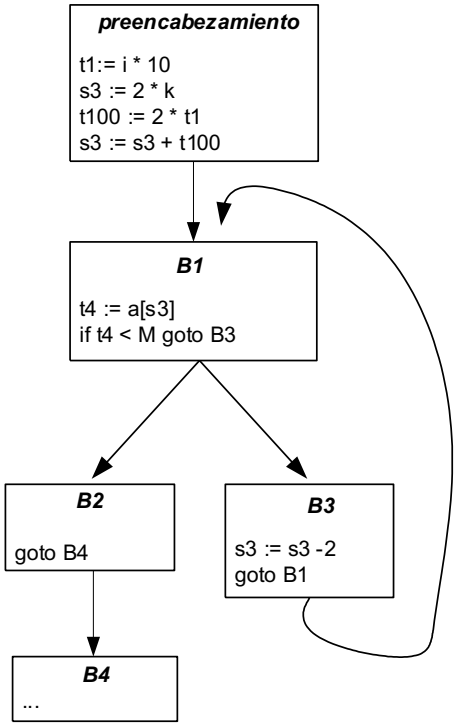


Aplicando el algoritmo de eliminación de variables de inducción quedaría:



La variable *s2* no estaba activa a la salida del bucle y no se usaba más que en su propia definición, por lo que se ha suprimido *s2 := s2-1*. Lo mismo ha ocurrido con la variable *k*.

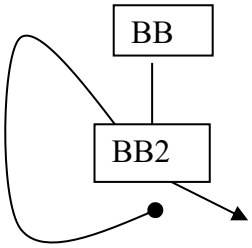
Teniendo en cuenta que la variable *s2* no está activa a la salida del bucle, se podría eliminar código inactivo del bloque correspondiente al preencabezamiento, quedando finalmente:



3.- a)

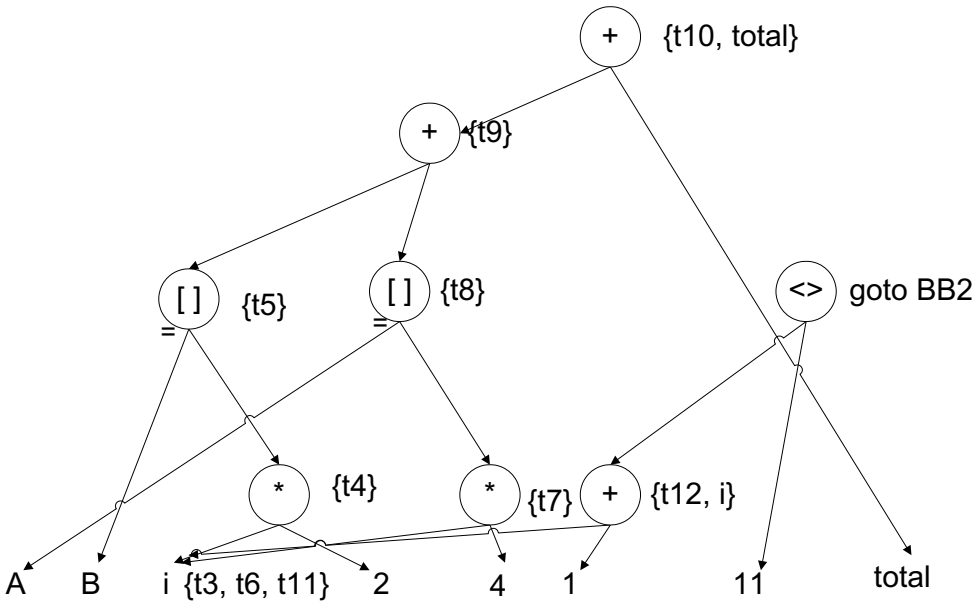
BB1: total := 0 i := 1	BB2: t3 := i t4 := t3 * 2 t5 := B[t4] t6 := i t7 := t6 * 4 t8 := A[t7] t9 := t5 + t8 t10 := t9 + total total := t10 t11 := i t12 := t11 + 1 i := t12 if i <> 11 goto 12

Grafo de flujo



GDA de BB1: (0) {total} (1) {i}

GDA de BB2:



Código reconstruido a partir de los GDAs:

BB1	total := 0 i := 1
BB2	t4 := i * 2 t5 := B[t4] t7 := i * 4 t8 := A[t7] t9 := t5 + t8 total := t9 + total i := i + 1 if i <> 11 goto BB2

b) El bucle está formado únicamente por el bloque básico BB2

Variable de inducción básica: i (i, 1, 0)

Otras variables de inducción: t4 (i, 2, 0) y t7(i, 4, 0)

c) Después de aplicar el algoritmo de reducción de intensidad quedaría:

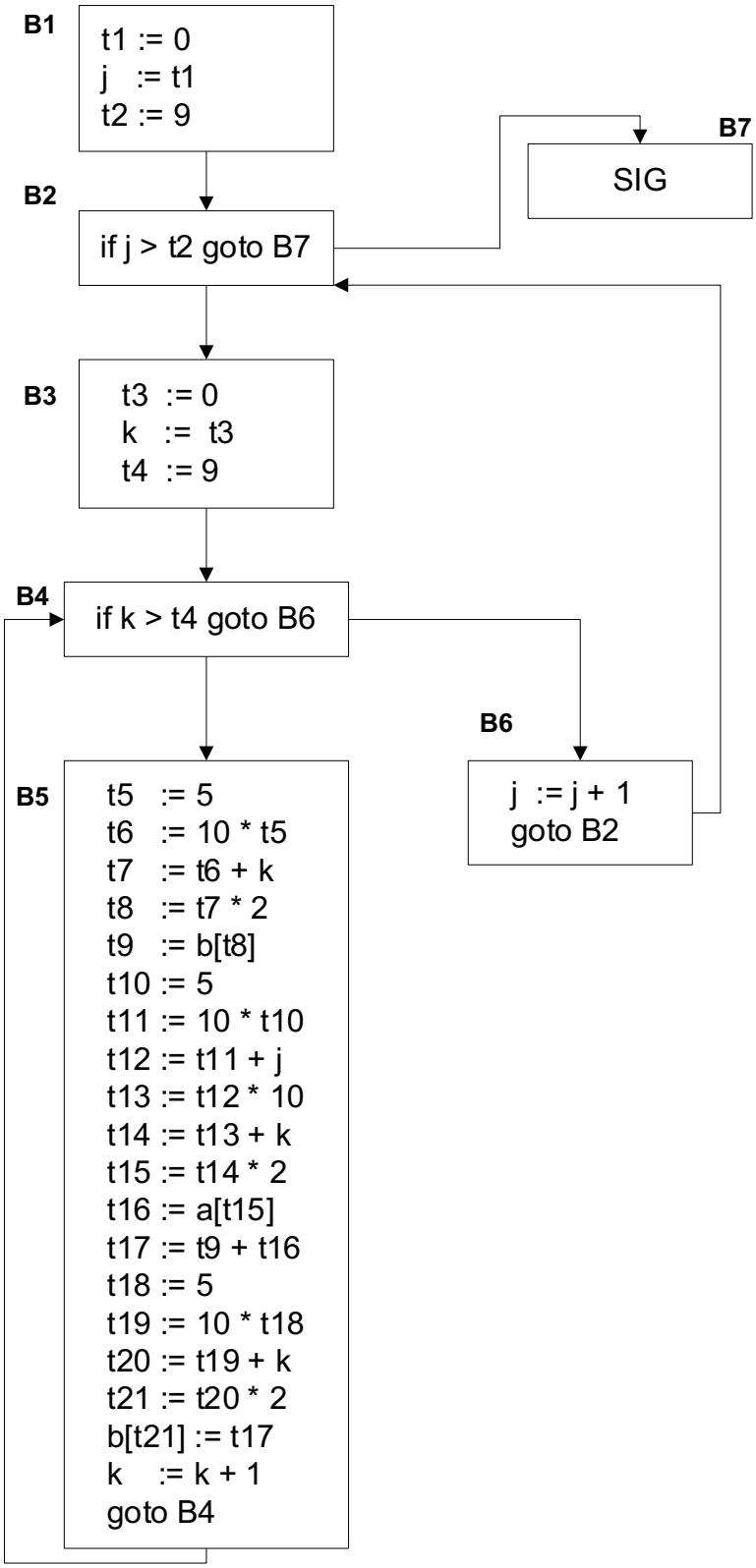
Pre-encabezamiento	s4 := i * 2 s7 := i * 4
BB2	t4 := s4 t5 := B[t4] t7 := s7 t8 := A[t7] t9 := t5 + t8 total := t9 + total i := i + 1 s7 := s7 + 4 s4 := s4 + 2 if i <> 11 goto BB2

Nuevas variables de inducción: s4 (i, 2, 0) y s7(i, 4, 0)

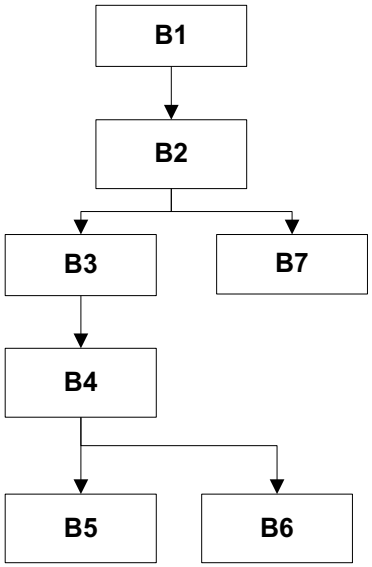
Después de aplicar el algoritmo de eliminación de variables de inducción quedaría:

Pre-encabezamiento	s4 := i * 2 s7 := i * 4 t13 := 11 * 2 (t13 := 22)
BB2	t5 := B[s4] t8 := A[s7] t9 := t5 + t8 total := t9 + total s7 := s7 + 4 s4 := s4 + 2 if s4 <> t13 goto BB2

4. a) Bloques básicos y grafo de flujo



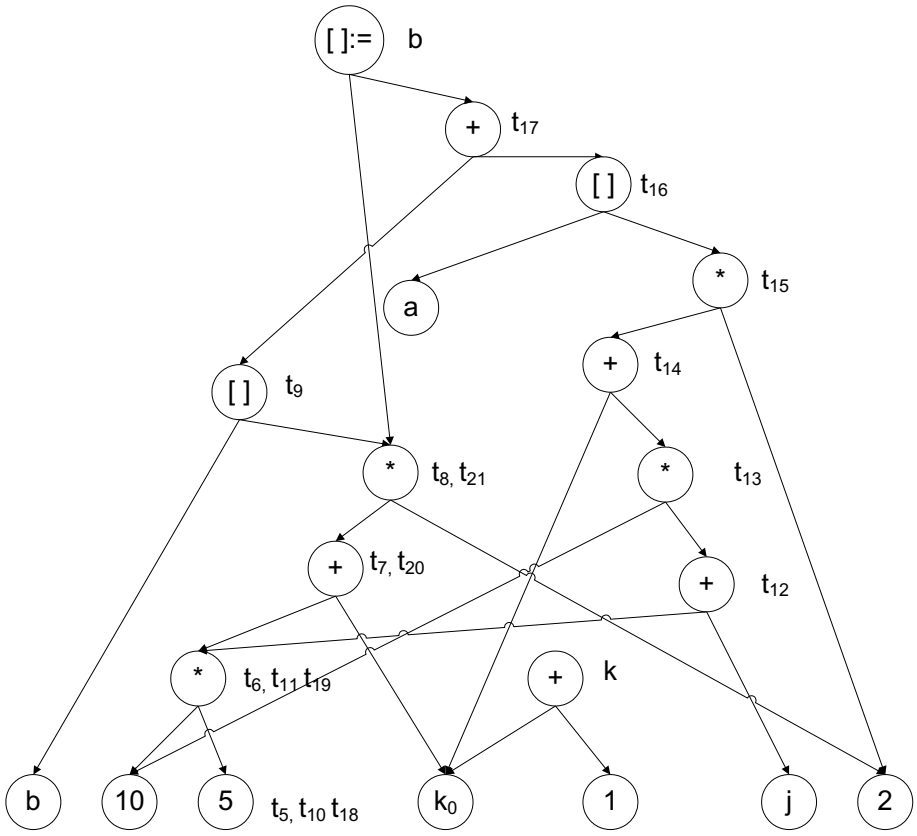
b) Árbol de dominación:



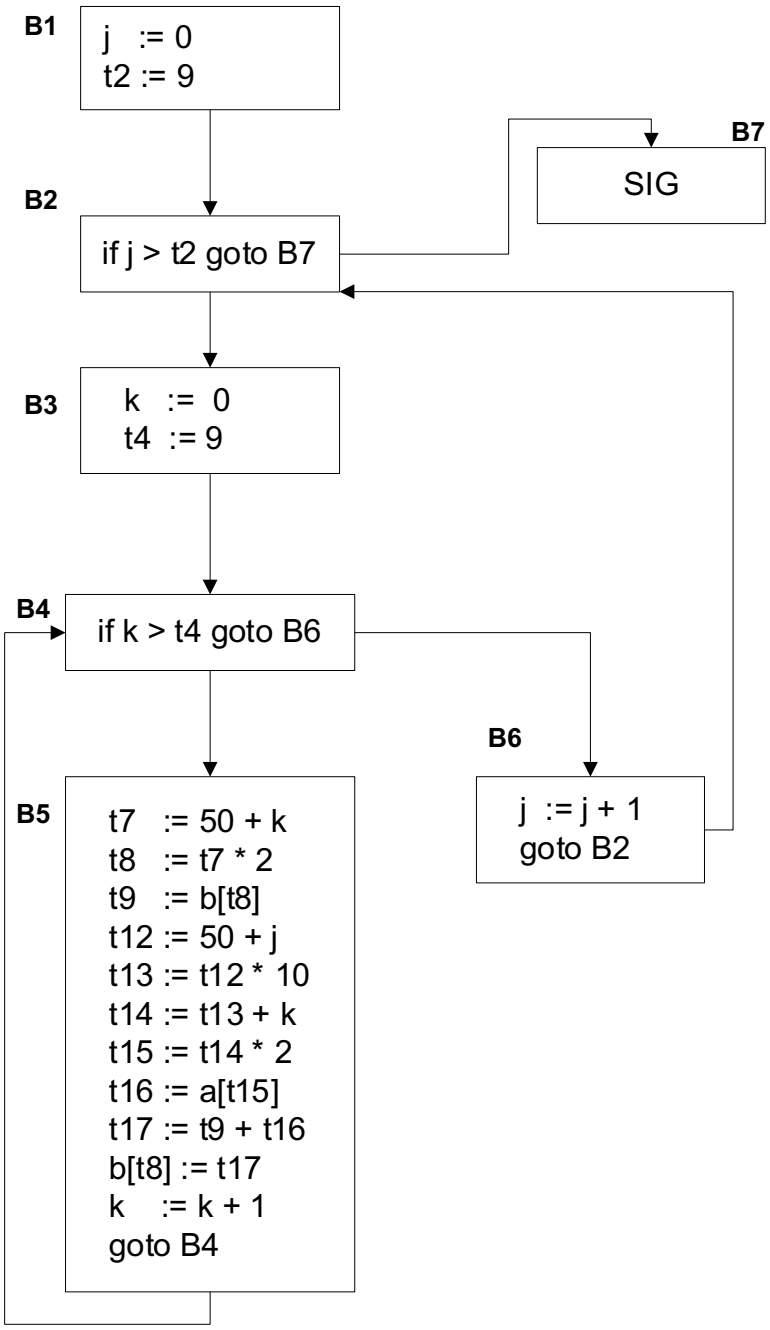
Aristas de retroceso B6 → B2 y B5 → B4
Bucle natural interno: { B4, B5 }
Bucle natural externo: { B2, B3, B4, B5, B6 }

c) Los GDAs de los bloques básicos B1, B3 y B6 son muy simples, igual que la reconstrucción del código correspondiente. El GDA asociado al bloque **B5** sería:

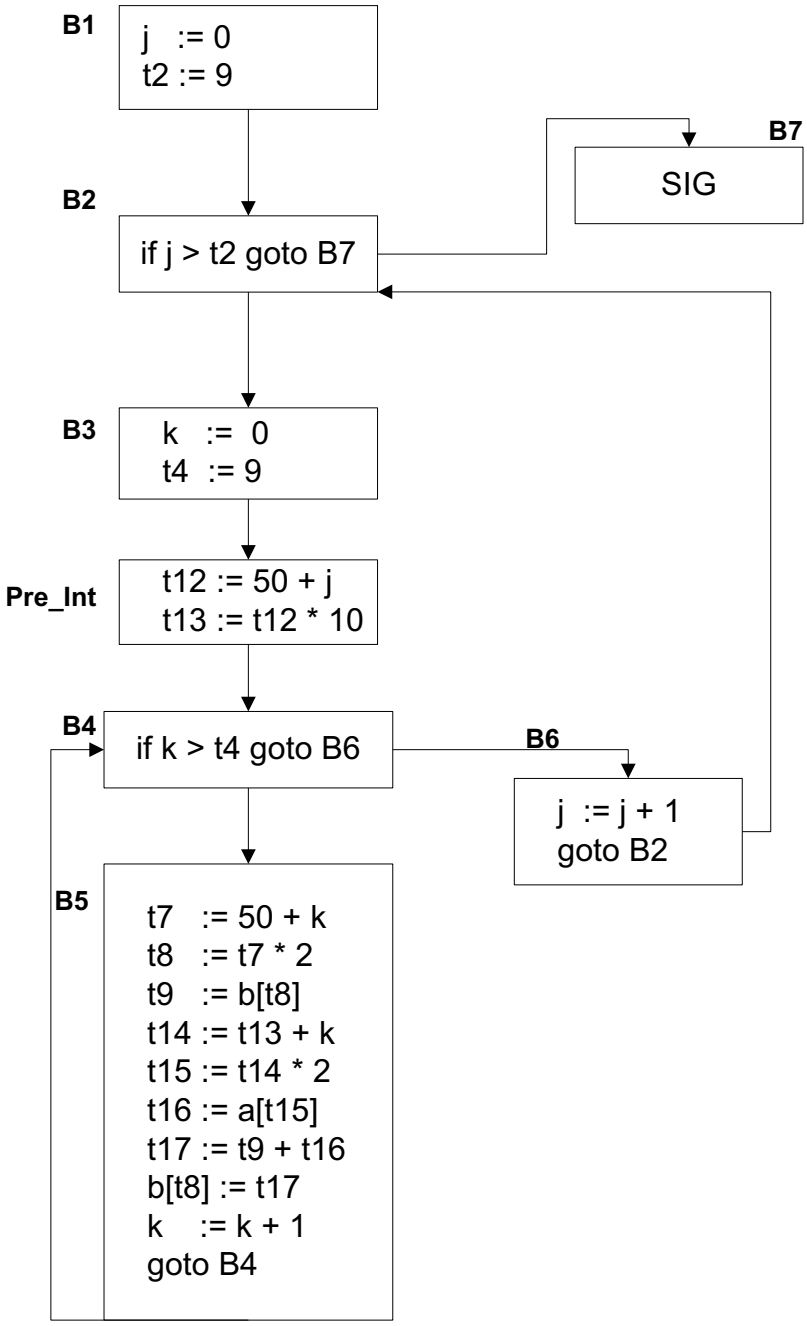
La



instrucción a[t21] := t17 se representa en el GDA mediante el nodo etiquetado con []:=. En el proceso de reconstrucción del código debe ocupar la misma posición que ocupaba en el bloque inicial. La reconstrucción del código produciría:

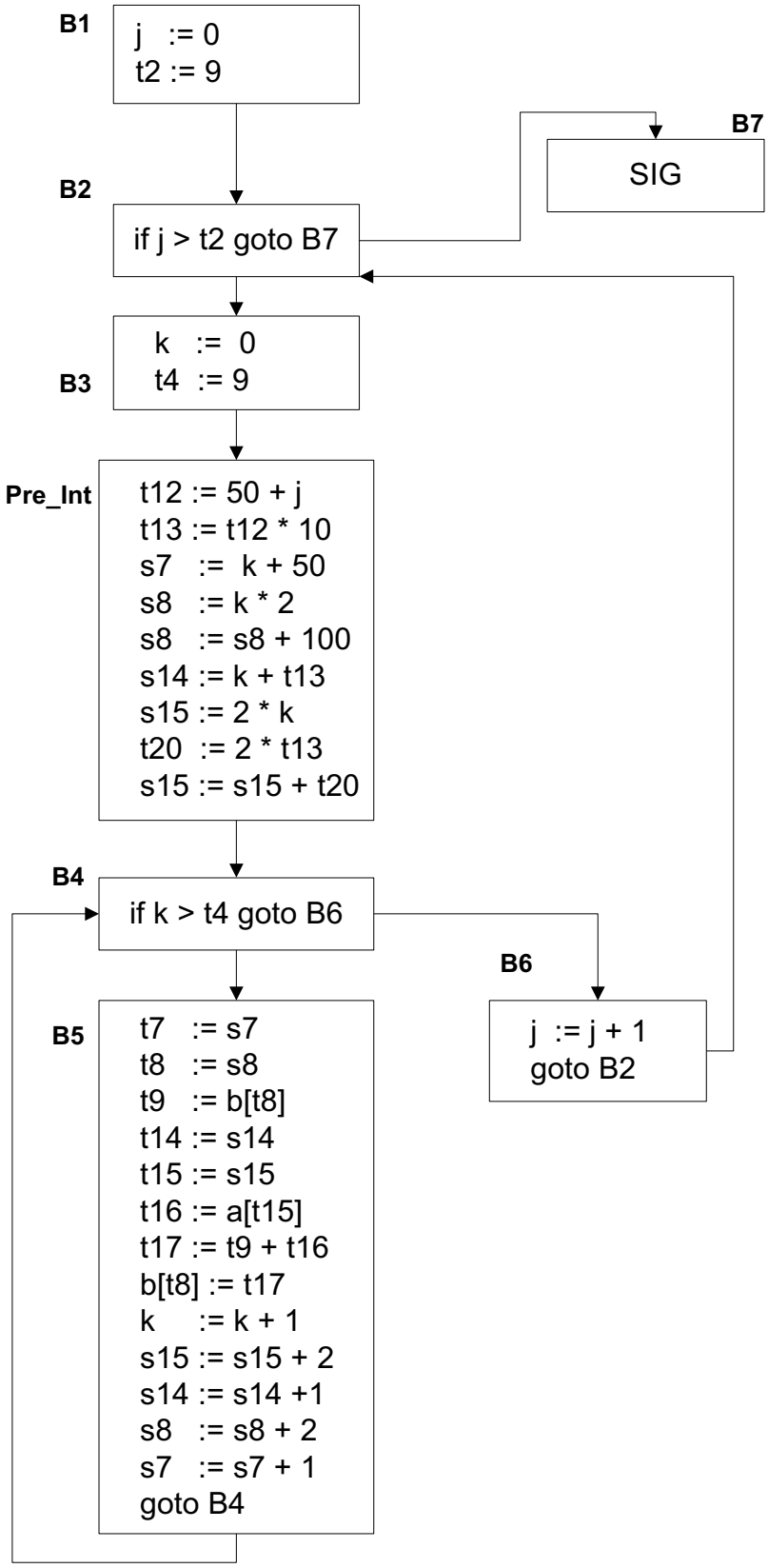


Las instrucciones t12 := 50 + j , t13 := t12 * 10 son invariantes dentro del bucle interno y reúnen las condiciones para ser extraídas fuera de este bucle (ni t2 ni t3 están activas a la salida del bucle) lo que genera el código siguiente:

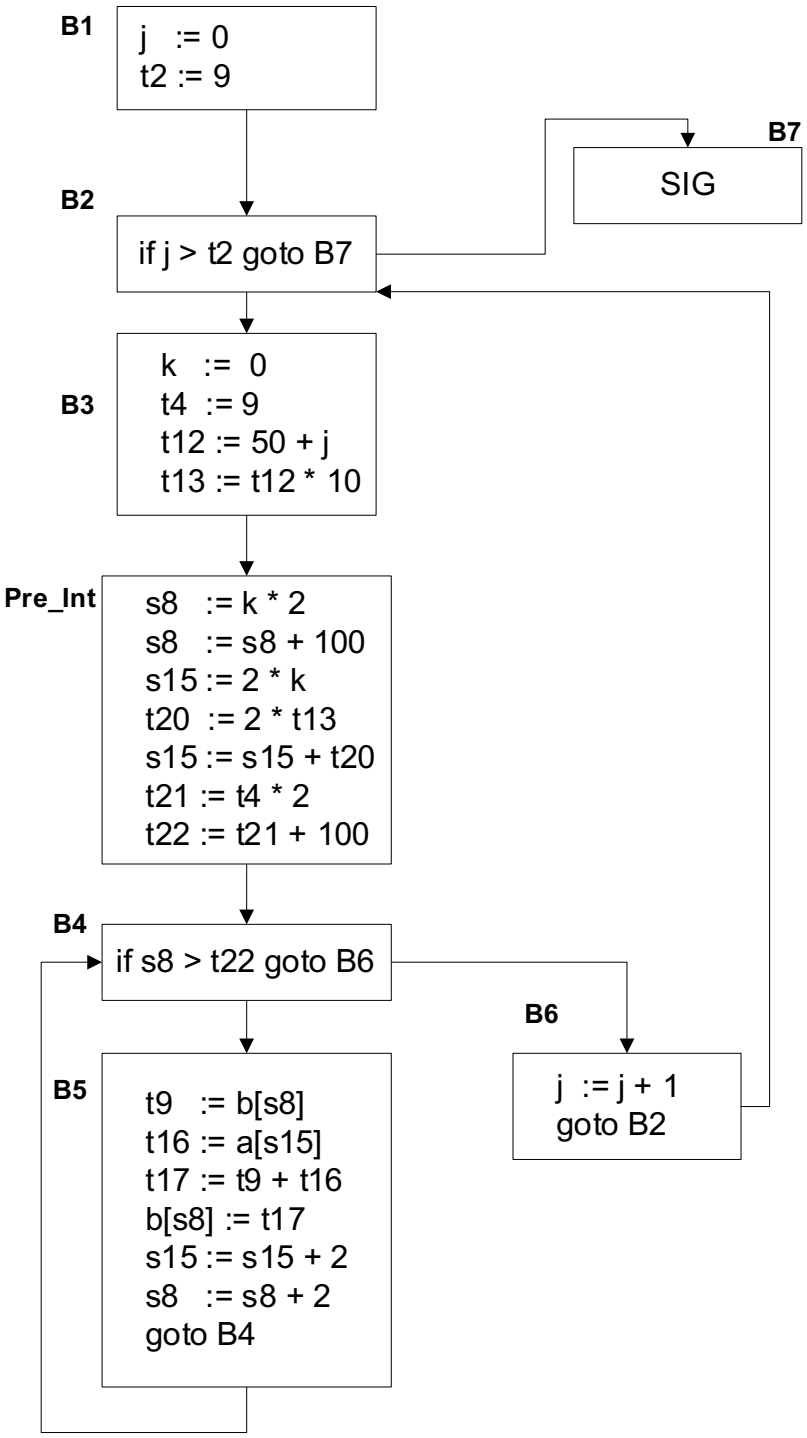


La variable de inducción básica del bucle interno es **k** (k, 1, 0) y las otras variables de la familia(k) son **t7** (k, 1, 50), **t8** (k, 2, 100), **t14** (k, 1, t13) y **t15** (k, 2, 2*t13). t14 es una variable de inducción porque t13 es un invariante de este bucle interno, t15 lo es, por serlo t14.

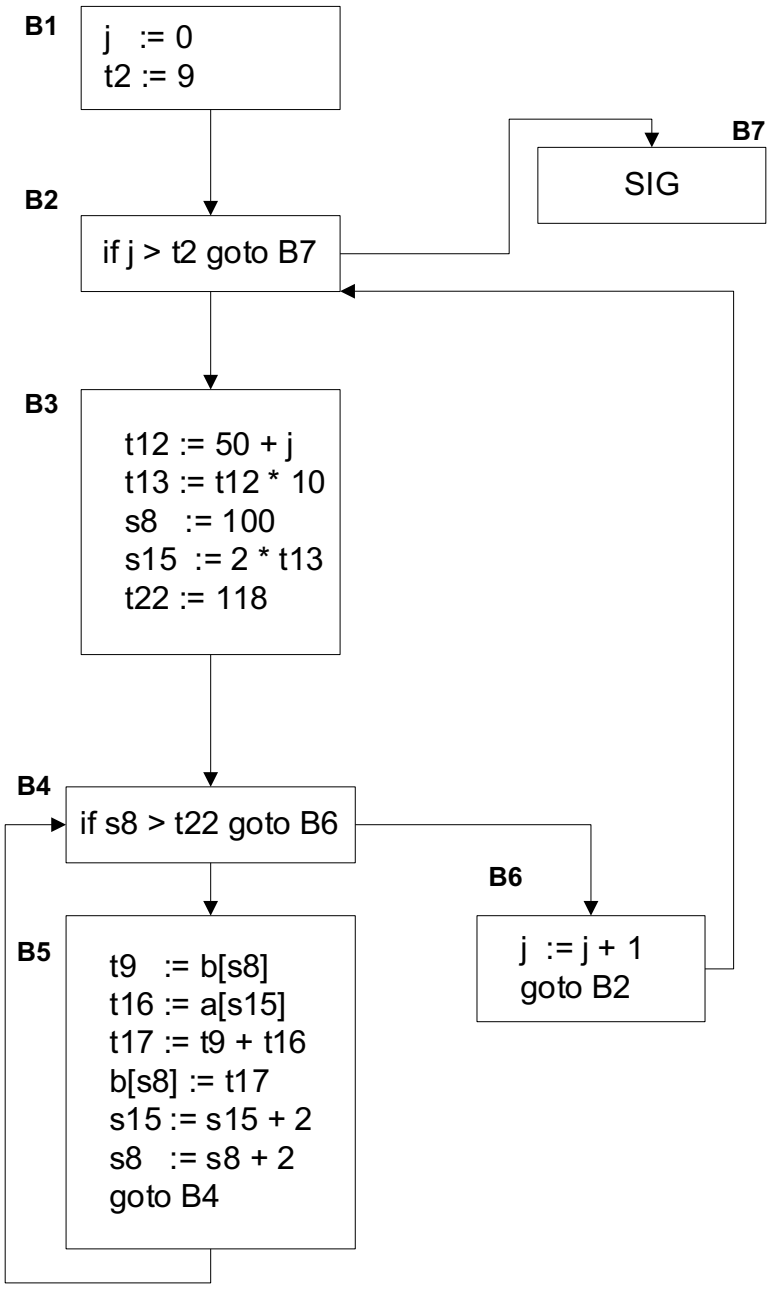
Reducción de intensidad aplicada al bucle interno:



La eliminación de variables de inducción en el bucle interno produciría:



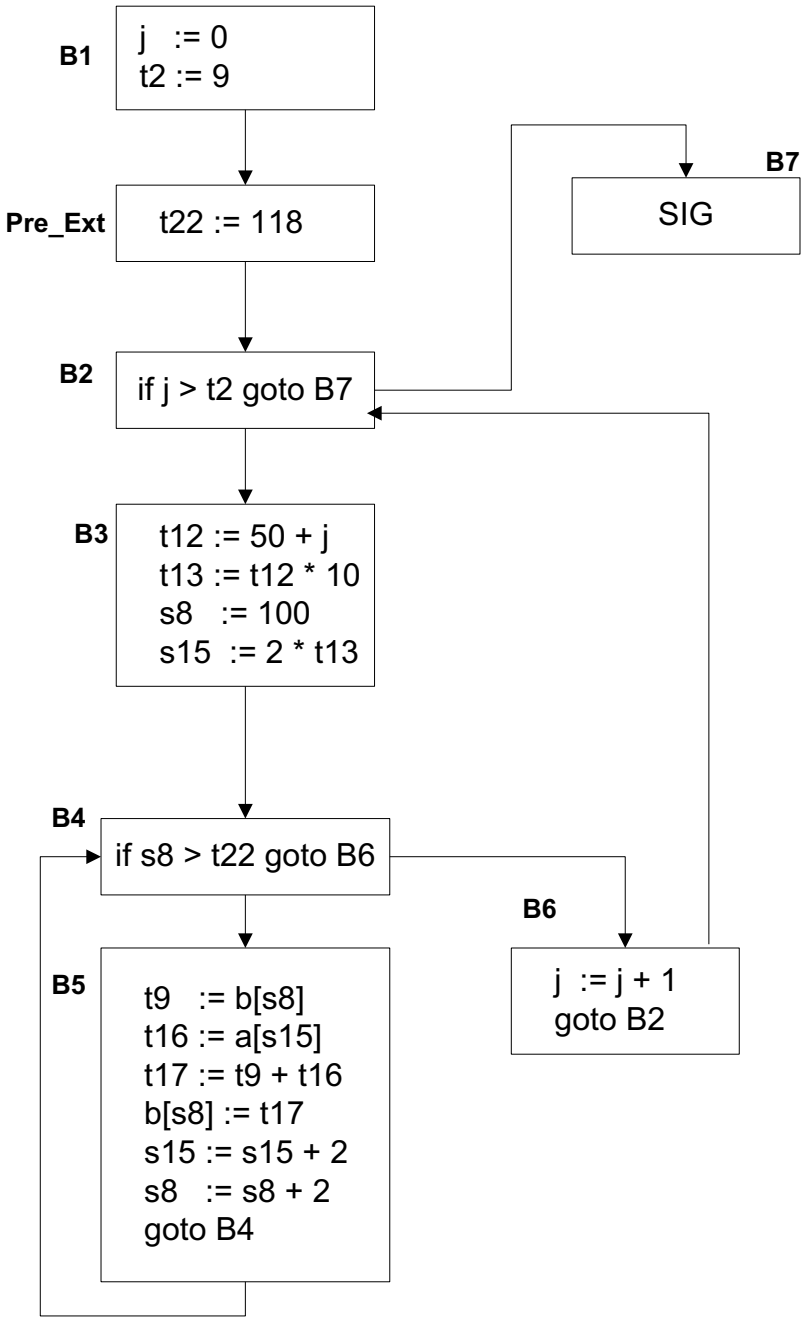
Construyendo el GDA del nuevo bloque básico formado por B3 y el pre-encabezamiento del bucle interno y reconstruyendo el código a partir de este GDA obtendríamos:



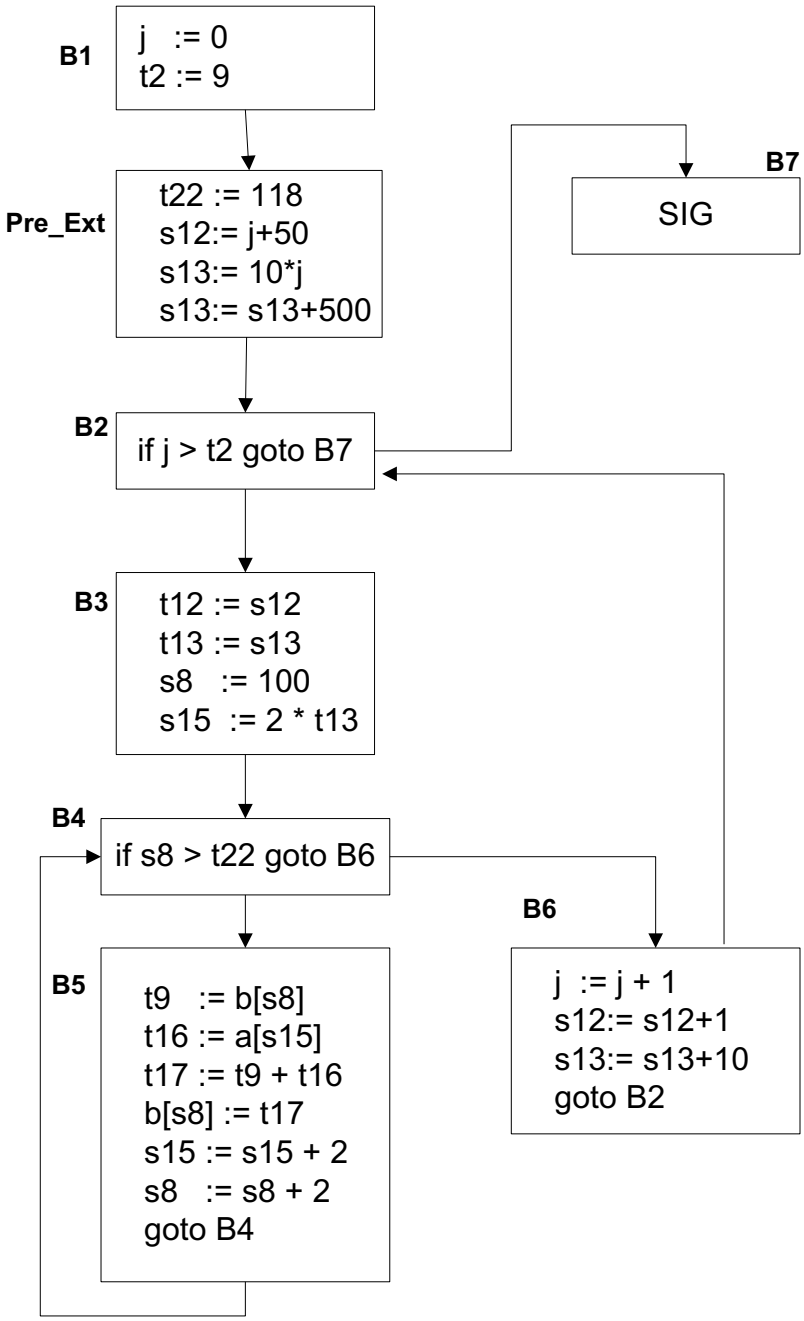
Ahora volvemos a aplicar las optimizaciones globales sobre el bucle externo (B2, B3, B4, B5 y B6).

Marcamos como invariantes las instrucciones s8:=100 y t22:=118, pero solo podemos extraer la segunda, porque la variable s8 se define también en el bloque B5, incumpliendo la segunda de las condiciones que debe cumplir una instrucción invariante para poder extraerse.

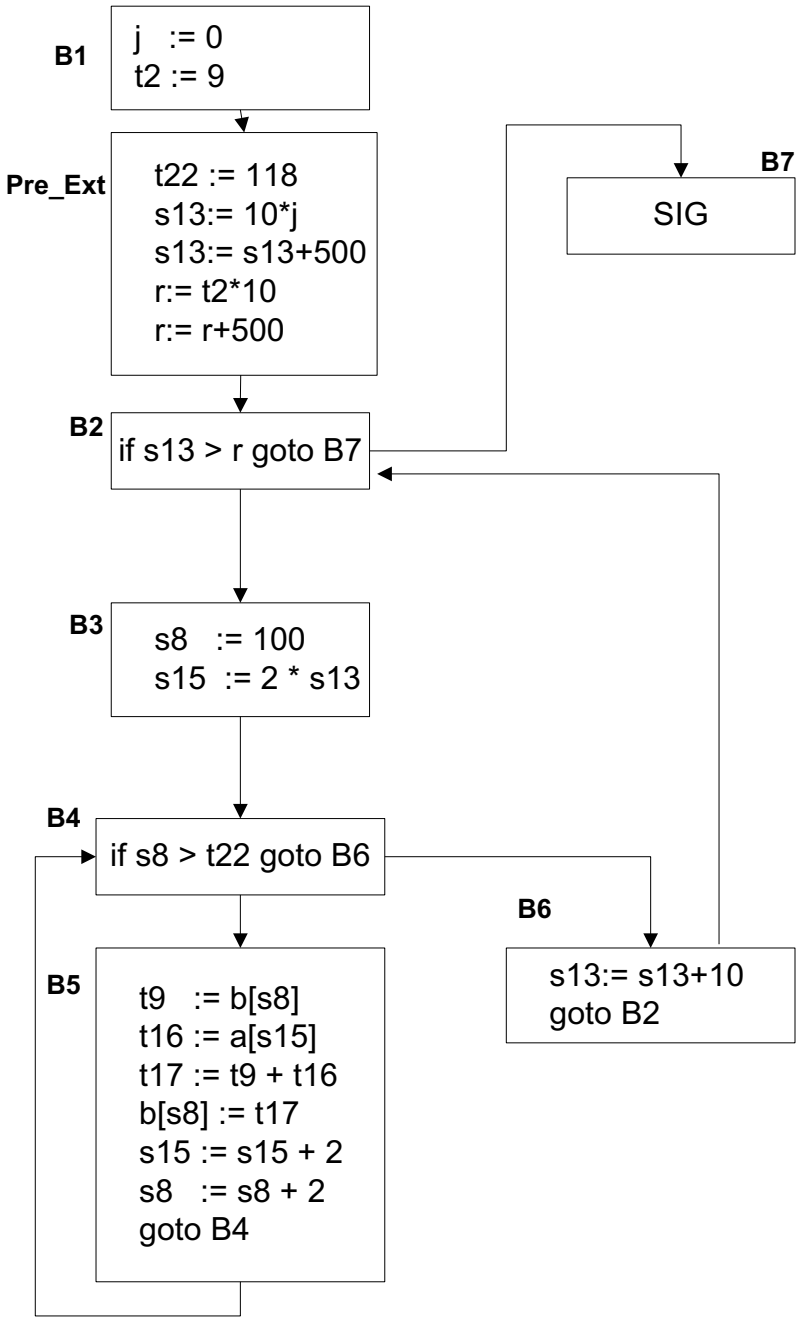
La variable de inducción básica del bucle externo es j (j,1,0), y el resto de variables de su familia son t12 (j, 1, 50), y t13(j,10,500). Se puede observar que la variable s15 no es variable de inducción del bucle externo porque tiene más de una definición en éste (s15:=2*t13 y s15:=s15+2).



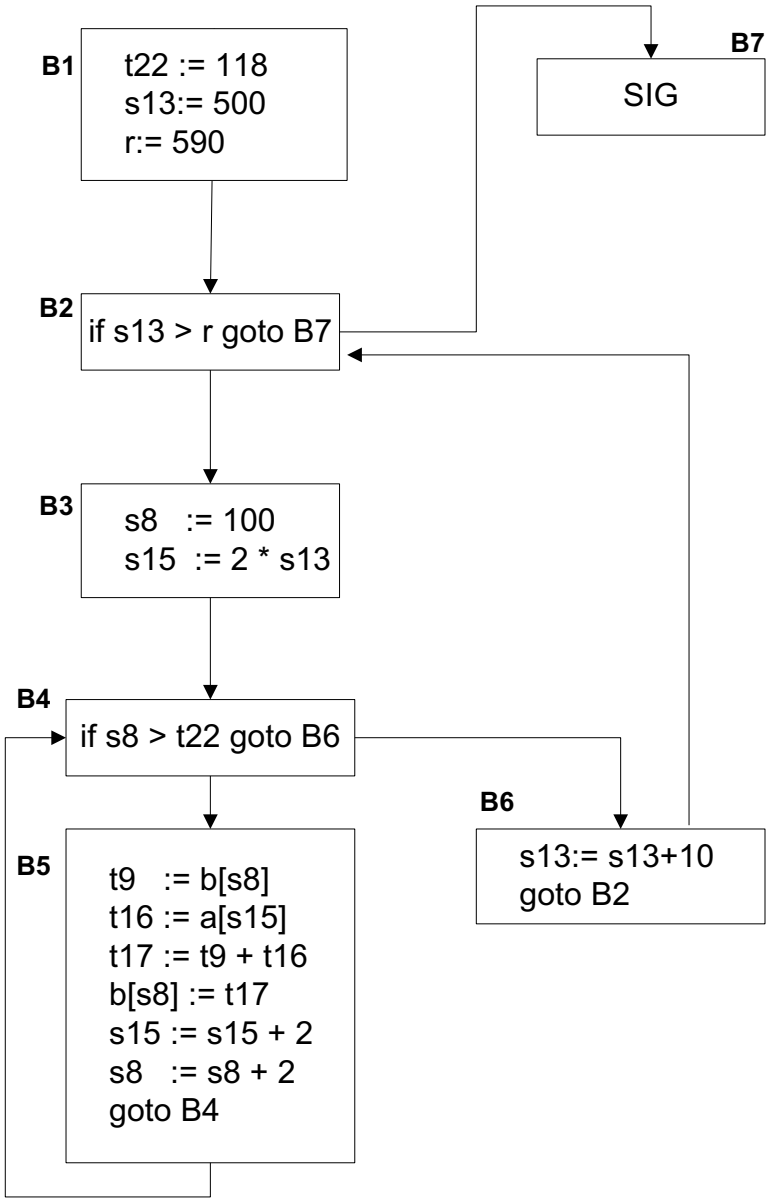
Aplicando el algoritmo de reducción de intensidad sobre las variables de inducción obtendríamos:



A continuación aplicamos el algoritmo de eliminación de variables de inducción. Podemos eliminar la variable j ya que solo se usa en su definición y en la condición del bucle. Para eliminar la variable j del bucle usamos la variable s13 (j,10,500). La variable s12 no está activa a la salida del bucle y solo se usa para su definición (además de la asignación a t12 que pronto podremos eliminar) lo que nos permitirá eliminarla.

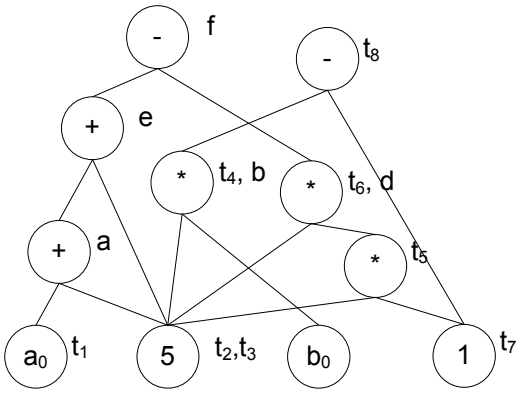


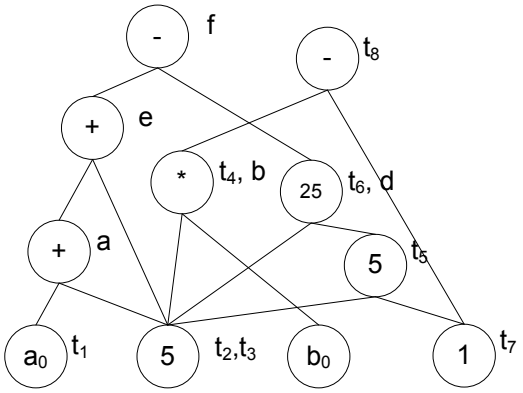
Si de nuevo construimos el GDA para el bloque B1 (que ahora incluye también el preencabezamiento del bucle exterior) y reconstruimos el código, obtendremos


B1
t22 := 118
s13:= 500
r:= 590
B7
SIG
B2
if s13 > r goto B7
B3
s8 := 100
s15 := 2 * s13
B4
if s8 > t22 goto B6
B6
s13:= s13+10
goto B2
B5
t9 := b[s8]
t16 := a[s15]
t17 := t9 + t16
b[s8] := t17
s15 := s15 + 2
s8 := s8 + 2
goto B4

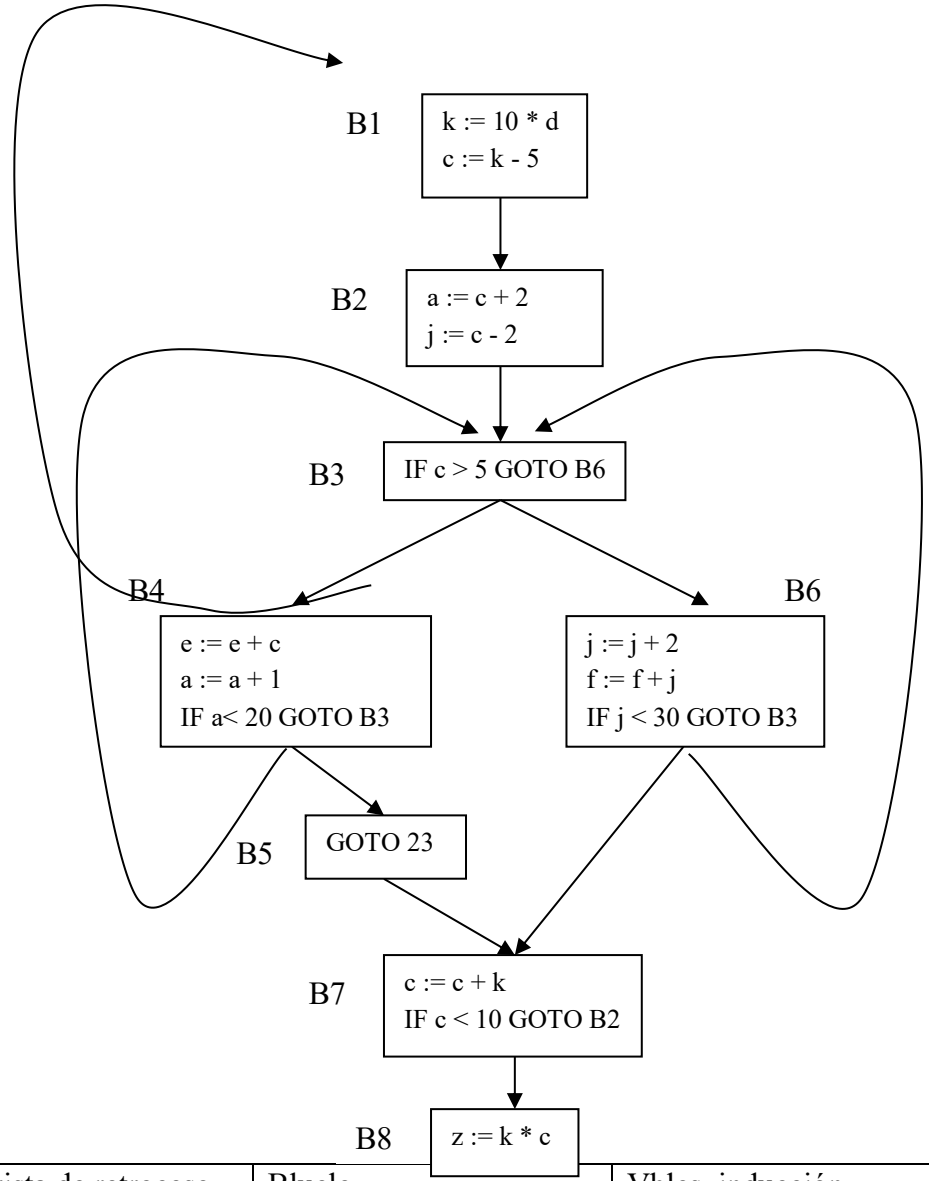
5. Se muestra a continuación el GDA obtenido a partir del código intermedio. En el primer GDA se puede aplicar plegado de constantes, obteniéndose el segundo GDA a partir del cual generamos el siguiente código intermedio:

a := a + 5
e := a + 5
b := b * 5
d := 25
f := e – 25





6.



Arista de retroceso	Blucle	Vbles. inducción
B4 -> B3	B3, B4	e (e, 1, 0) ; a (a,1,0)
B6 -> B3	B3, B6	j (j,1,0)
B7 -> B2	B2, B3, B4, B5, B6, B7	c (c,1,0)

Nota:

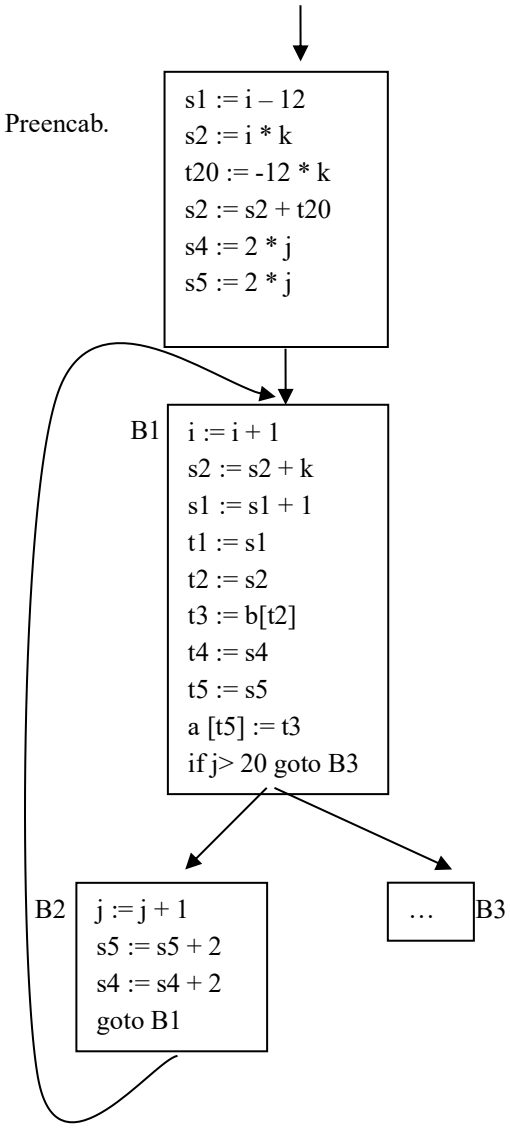
Ni a ni j son variables de inducción del bucle B7->B2 porque hay más de una definición de cada una de ellas dentro del bucle.

7.-

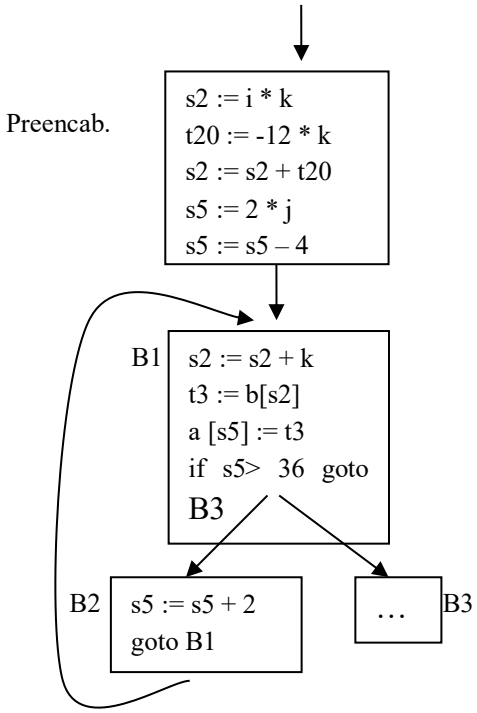
a)

Las variables de inducción son:

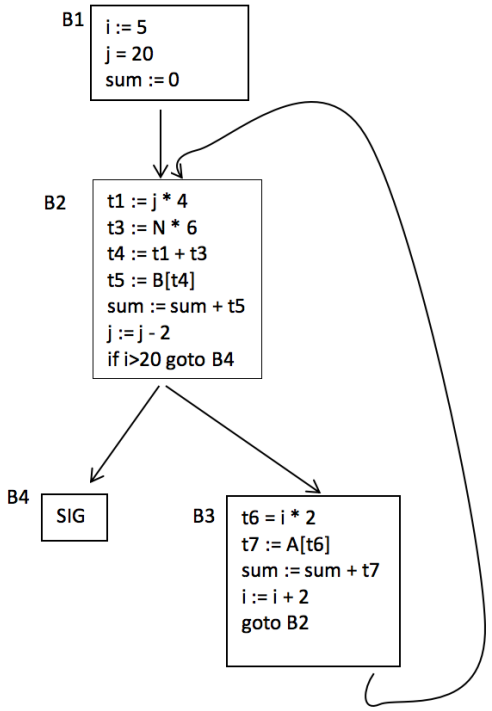
i (i, 1,0) t1 (i, 1, -12) t2 (i, k, -12k)
j (j, 1, 0) t4 (j, 2, 0) t5 (j, 2, -4)



b)

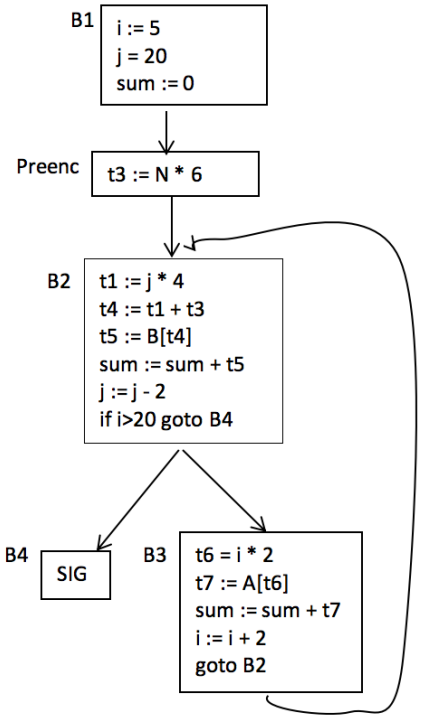


8.-a)

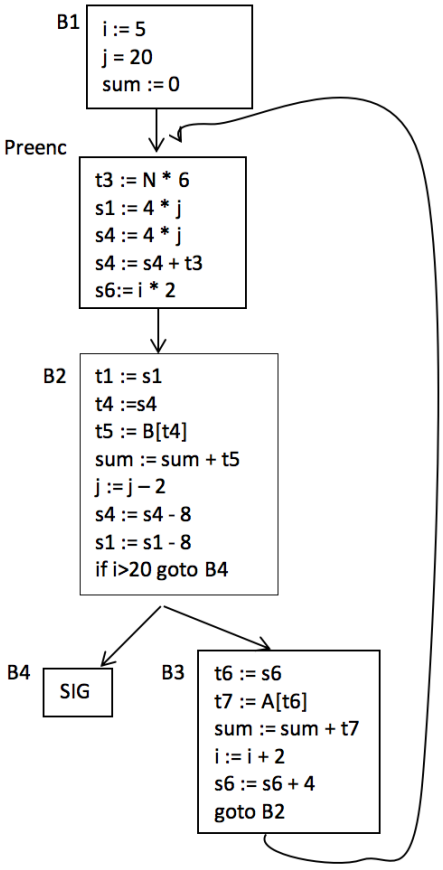


Se extrae del bucle el invariante t3 := N + 6.
Variables de inducción:

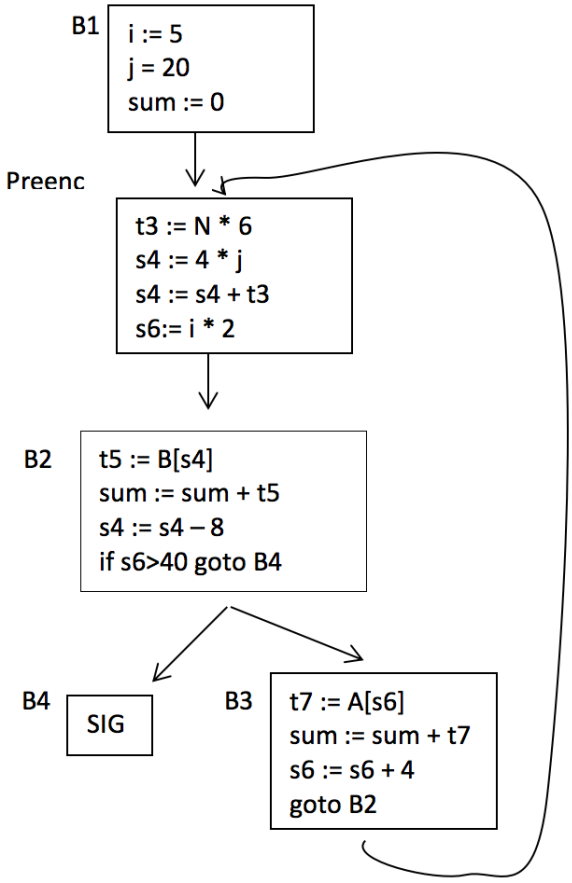
i (i,1, 0), t6 (i, 2, 0)
j (j, 1, 0), t1 (j, 4, 0) , t4 (j, 4, t3)


B1
i := 5
j = 20
sum := 0
Preenc
t3 := N * 6
B2
t1 := j * 4
t4 := t1 + t3
t5 := B[t4]
sum := sum + t5
j := j - 2
if i>20 goto B4
B4
SIG
B3
t6 = i * 2
t7 := A[t6]
sum := sum + t7
i := i + 2
goto B2

b)


B1
i := 5
j = 20
sum := 0
Preenc
t3 := N * 6
s1 := 4 * j
s4 := 4 * j
s4 := s4 + t3
s6:= i * 2
B2
t1 := s1
t4 :=s4
t5 := B[t4]
sum := sum + t5
j := j – 2
s4 := s4 - 8
s1 := s1 - 8
if i>20 goto B4
B4
SIG
B3
t6 := s6
t7 := A[t6]
sum := sum + t7
i := i + 2
s6 := s6 + 4
goto B2

c)


B1
i := 5
j = 20
sum := 0
Preenc
t3 := N * 6
s4 := 4 * j
s4 := s4 + t3
s6:= i * 2
B2
t5 := B[s4]
sum := sum + t5
s4 := s4 – 8
if s6>40 goto B4
B4
SIG
B3
t7 := A[s6]
sum := sum + t7
s6 := s6 + 4
goto B2

Bibliografía.

Este tema ha sido desarrollado tomando como base el capítulo 10 del [Aho, Sethi y Ullman 1990], complementándolo con el capítulo 12 del [Tremblay y Soenson 1985], con el capítulo 11 del [Aho y Ullman 1973 - Vol. 2] y con [A. W. Appel, M. Ginsburg.1998, Modern compiler implementation in C].