

Optimización de Código

Optimización Local

- Las optimizaciones locales se realizan sobre el bloque básico
- Optimizaciones locales
 - Folding
 - Propagación de constantes
 - Reducción de potencia
 - Reducción de subexpresiones comunes

Bloque Básico

- Un bloque básico es un fragmento de código que tiene una única entrada y salida, y cuyas instrucciones se ejecutan secuencialmente. Implicaciones:
 - Si se ejecuta una instrucción del bloque se ejecutan todas en un orden conocido en tiempo de compilación.
- La idea del bloque básico es encontrar partes del programa cuyo análisis necesario para la optimización sea lo más simple posible.

Bloque Básico (ejemplos)

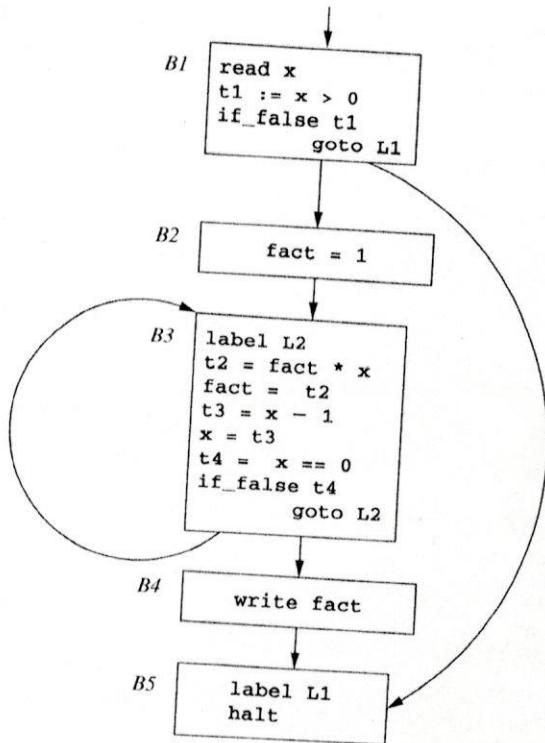
- Separación correcta

```
for (i=1; i<10; ++i) {  
    b=b+a[i];  
    c=b*i;  
}  
a=3;  
b=4;  
goto l1;  
c=10;  
l1: d=3;  
e=4;
```

BB1:	i=1;
BB2:	i<10;
BB3:	b=b+a[i]; c=b*i; ++i
BB4:	a=3; b=4; goto l1;
BB5:	c=10;
BB6:	l1: d=3; e=4;

Figura 8.18

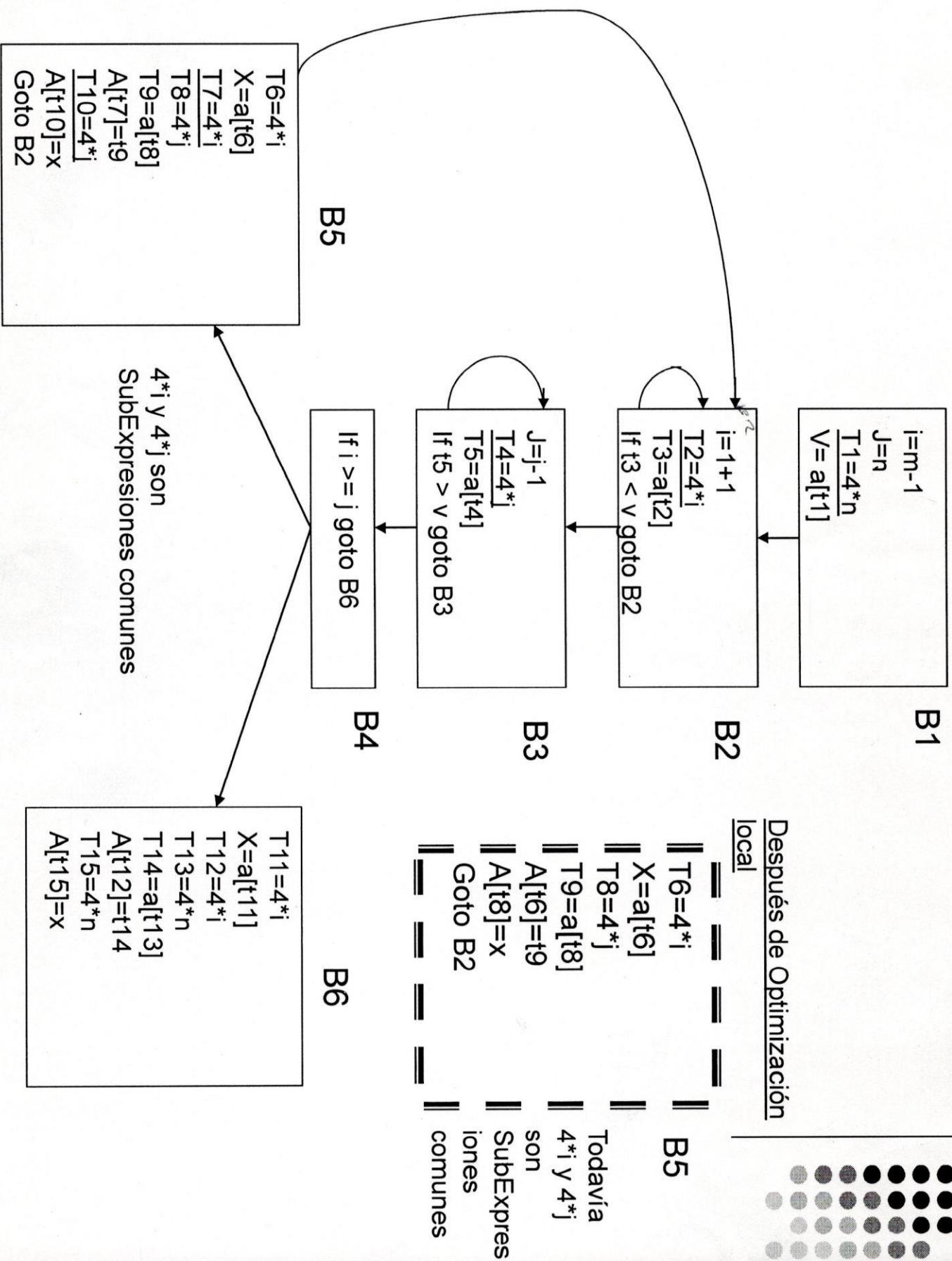
La gráfica de flujo del código intermedio de la figura 8.2



La gráfica de flujo es la principal estructura de datos que necesita el analizador de datos, el cual la utiliza para acumular información que será empleada en las optimizaciones. Diferentes clases de información pueden requerir diferentes clases de procesos. La gráfica de flujo, y la información obtenida puede ser muy variada, dependiendo de las clases de optimizaciones deseadas. Aunque no tenemos espacio en esta brevísima general para describir la técnica de análisis de flujo de datos con detalle (ver las notas y referencias al final del capítulo), puede valer la pena describir la clase de datos que se pueden acumular mediante este proceso.

Un problema estándar de análisis de flujo de datos es calcular, para cada variable, el conjunto de las denominadas **definiciones de alcance** de esa variable al principio de cada bloque básico. Aquí una **definición** es una instrucción de código intermedio que establece el valor de la variable, tal como una asignación o una lectura.²³ Por ejemplo, las definiciones de la variable **fact** en la figura 8.18 son la instrucción **fact = 1** en el bloque básico **B2** (**fact=1**) y la tercera instrucción del bloque **B3** (**fact=t2**). Las definiciones **d1** y **d2**. Se dice que una definición **alcanza** un bloque básico si al principio de ese bloque la variable todavía puede tener el valor establecido por esta definición. En la gráfica de flujo de la figura 8.18 se puede establecer que ninguna definición de **fact** alcanza **B2**, que tanto **d1** como **d2** alcanzan **B3** y que sólo **d2** alcanza **B4** y **B5**. Las definiciones de alcance se pueden utilizar en diversas optimizaciones: en la propagación de valores, por ejemplo, si las únicas definiciones que alcanzan un bloque representan un valor constante.

23. Esto no debe confundirse con una definición de C, la cual es una clase de declaración.



Ensamblamiento (Folding)

- El ensamblamiento es remplazar las expresiones por su resultado cuando se pueden evaluar en tiempo de compilación (resultado constante).
 - Ejemplo: $A=2+3+A+C \rightarrow A=5+A+C$
- Estas optimizaciones permiten que el programador utilice cálculos entre constantes representados explícitamente sin introducir ineficiencias.

Propagación de constantes

- Desde que se asigna a una variable un valor constante hasta la siguiente asignación, se considera a la variable equivalente a la constante.
- Ejemplo: Propagación Ensamblamiento
 $\text{PI}=3.14 \rightarrow \text{PI}=3.14 \rightarrow \text{PI}=3.14$
 $\text{G2R}=\text{PI}/180 \rightarrow \text{G2R}=3.14/180 \rightarrow \text{G2R}=0.017$
PI y G2R se consideran constantes hasta la próxima asignación.
- Estas optimizaciones permiten que el programador utilice variables como constantes sin introducir ineficiencias. Por ejemplo en C no hay constantes y será lo mismo utilizar
 - Int a=10;
 - #define a 10Con la ventaja que la variable puede ser local.
- Actualmente en C se puede definir const int a=10;

Reducción de potencia(strength reduction)

- Se busca sustituir operaciones costosas por otras mas simples. Ejemplo:
 - sustituir productos por sumas.
 $a=2*a$
 $a=a+a$
 - Evitar la operación append (++)
 $A=length(s1 ++ s2)$
convertirlo en
 $A=length(s1)+length(s2)$

Eliminación de subexpresiones redundantes.

- Las subexpresiones que aparecen más de una vez se calculan una sola vez y se reutiliza el resultado.
- Idea: Detectar las subexpresiones iguales y que las compartan diversas ramas del árbol. Problema: Hay que trabajar con un **grafo acíclico**.
- Dos expresiones pueden ser equivalentes y no escribirse de la misma forma $A+B$ es equivalente a $B+A$. Para comparar dos expresiones se utiliza la **forma normal**
 - Se ordenan los operandos: Primero las constantes, después variables ordenadas alfabéticamente, las variables indexadas y las subexpresiones. Ejemplo
 - X=C+3+A+5 -> X= 3+5+A+C
 - Y=2+A+4+C -> Y=2+4+A+C
 - divisiones y restas se ponen como sumas y productos para poder commutar
 - $A-B \rightarrow A + (-B)$
 - $A/B \rightarrow A * (1/B)$

Optimizaciones Dentro de Bucles

- La optimización de bucles es muy importante por las mejoras en tiempo de ejecución que se obtienen
- Estrategias de optimización dentro de bucles
 - Expansión de bucles (loop unrolling)
 - Reducción de frecuencia (frequency reduction)
 - Reducción de potencia (strength reduction)

Loop unwinding

From Wikipedia, the free encyclopedia

Loop unwinding, also known as **loop unrolling**, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size (space-time tradeoff). The transformation can be undertaken manually by the programmer or by an optimizing compiler.

The goal of loop unrolling is to increase a program's speed by reducing (or eliminating) instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration;^[1] reducing branch penalties; as well as "hiding latencies, in particular, the delay in reading data from memory".^[2] To eliminate this overhead, loops can be re-written as a repeated sequence of similar independent statements.^[3]

Contents

- 1 Advantages
- 2 Disadvantages
- 3 Static/manual loop unrolling
 - 3.1 A simple manual example in C
 - 3.2 Early complexity
 - 3.3 Unrolling WHILE loops
- 4 Dynamic unrolling
 - 4.1 Assembler example (IBM/360 or Z/Architecture)
 - 4.2 C example
- 5 See also
- 6 References
- 7 Further reading
- 8 External links

Advantages

The overhead in "tight" loops often consists of instructions to increment a pointer or index to the next element in an array (pointer arithmetic), as well as "end of loop" tests. If an optimizing compiler or assembler is able to pre-calculate offsets to each *individually referenced* array variable, these can be built into the machine code instructions directly, therefore requiring no additional arithmetic operations at run time (note that in the example given below this is not the case).

- Significant gains can be realized if the reduction in executed instructions compensates for any performance reduction caused by any increase in the size of the program.
- branch penalty is minimised.^[4]
- If the statements in the loop are independent of each other (i.e. where statements that occur earlier in the loop do not affect statements that follow them), the statements can potentially be executed in parallel.
- Can be implemented dynamically if the number of array elements is unknown at compile time (as in Duff's device)

Optimizing compilers will sometimes perform the unrolling automatically, or upon request.

Disadvantages

- Increased program code size, which can be undesirable, particularly for embedded applications.
 - Can also cause an increase in instruction cache misses, which may adversely affect performance.
- Unless performed transparently by an optimizing compiler, the code may become less readable.
- If the code in the body of the loop involves function calls, it may not be possible to combine unrolling with inlining, since the increase in code size might be excessive. Thus there can be a trade-off between the two optimizations.
- Possible increased register usage in a single iteration to store temporary variables, which may reduce performance, though much will depend on possible optimizations.^[5]
- Apart from very small and simple codes, unrolled loops that contain branches are even slower than recursions^[6]

Static/manual loop unrolling

Manual (or static) loop unrolling involves the programmer analyzing the loop and interpreting the iterations into a sequence of instructions which will reduce the loop overhead. This is in contrast to dynamic unrolling which is accomplished by the compiler.

A simple manual example in C

A procedure in a computer program is to delete 100 items from a collection. This is normally accomplished by means of a *for*-loop which calls the function *delete(item_number)*. If this part of the program is to be optimized, and the overhead of the loop requires significant resources compared to those for the *delete(x)* loop, unwinding can be used to speed it up.

Normal loop	After loop unrolling
<pre>int x; for (x = 0; x < 100; x++) { delete(x); }</pre>	<pre>int x; for (x = 0; x < 100; x+=5) { delete(x); delete(x+1); delete(x+2); delete(x+3); delete(x+4); }</pre>

As a result of this modification, the new program has to make only 20 iterations, instead of 100. Afterwards, only 20% of the jumps and conditional branches need to be taken, and represents, over many iterations, a potentially significant decrease in the loop administration overhead. To produce the optimal benefit, no variables should be specified in the unrolled code that require pointer arithmetic. This usually requires "base plus offset" addressing, rather than indexed referencing.

On the other hand, this *manual* loop unrolling expands the source code size from 3 lines to 7, that have to be produced, checked, and debugged, and the compiler may have to allocate more registers to store variables in the expanded loop iteration. In addition, the loop control variables and number of operations inside the unrolled loop structure have to be chosen carefully so that the result is indeed the same as in the original code (assuming

Reducción de frecuencia. (frequency reduction)

- La reducción de frecuencia detecta las operaciones invariantes de bucle y las calcula una única vez delante del bucle. Ejemplo:

```
for i=1 to n do c=i*sin(a);
- sin(a) es una operación invariante del bucle que puede pasar
de calcularse n veces a una con la siguiente transformación
tmp=sin(a);
for i=1 to n do c=i*tmp;
- Esta transformación no tiene en cuenta que el bucle igual no
se ejecuta ninguna vez y esto supondría perder tiempo de
ejecución calculando la invariante de bucle
innecesariamente. Además el calculo de la invariante puede
producir un error de ejecución. Por ejemplo una división por
cero.
```

Reducción de potencia(strength reduction)

- Reducción de potencia aplicada a bucles
 - Sustituir productos entre variables inductivas e invariantes de bucle por sumas

```
for(i=1; i<10; ++i) a[i]=3*i;  
convertir en  
for(i=1,j=3;i<10;++i,j+=3) a[i]=j;
```
- Problemas a resolver
 - Detectar las invariantes de bucle
 - Ya esta solucionado
 - Detectar las variables inductivas

Optimización Global

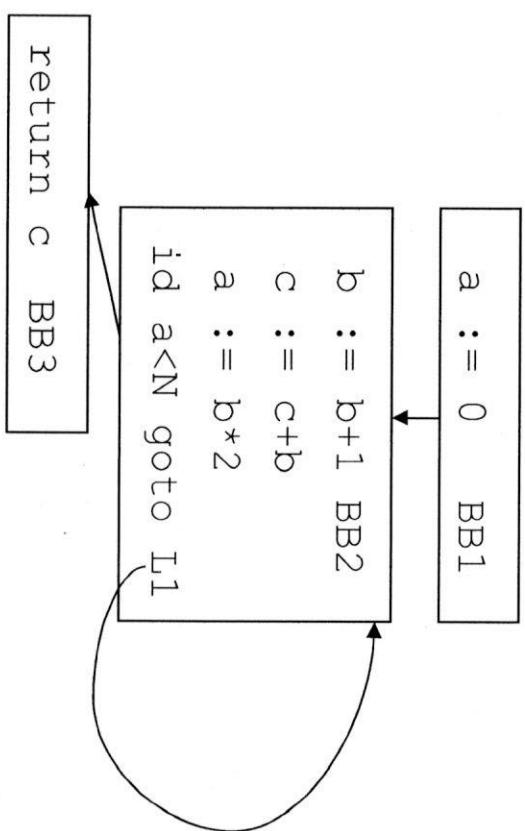
- **Grafo del flujo de ejecución**
 - Antes de realizar una optimización global es necesario crear el grafo de flujo de ejecución.
 - El grafo de flujo de ejecución representa todos los caminos posibles de ejecución del programa.
 - La información contenida en el grafo es útil para
 - el programador y
 - el optimizador
- La optimización global a partir del análisis del grafo del flujo de ejecución permite
 - Una propagación de constantes fuera del bloque básico.
 - Eliminación del código no utilizado
 - Una mejor asignación de los registros.
 - Etc.
- Problema: la optimización global es muy costosa en tiempo de compilación

Control Flow Graph (CFG)

A program's Control Flow Graph is a directed graph, whose nodes are Basic Blocks, and whose vertices are program-defined flows of control from Basic Blocks to others

Example

- (1) $a := 0$
L1:
(2) $b := b+1$
(3) $c := c+b$
(4) $a := b*2$
(5) if $a < N$ goto L1
(6) return c



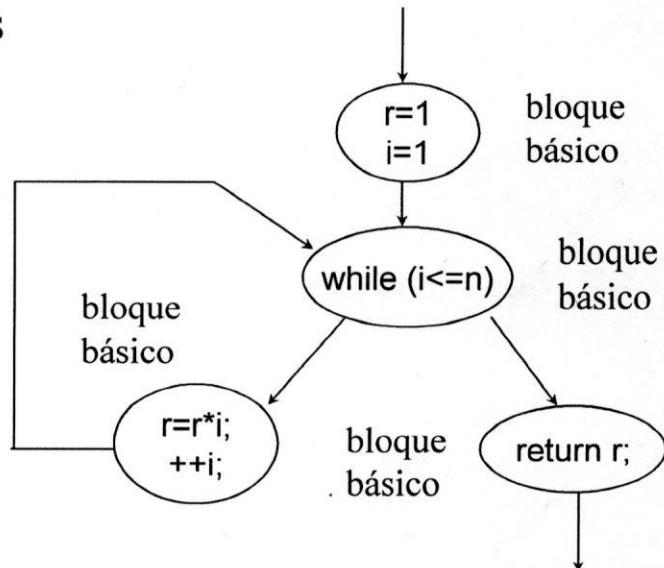
Construcción del Grafo del Flujo de Ejecución

- Tipos de grafo

- Orientado a procedimiento/función
 - Grafo de llamadas

- Ejemplo

```
int fact(int n) {  
    int r;  
    r=1;  
    i=1;  
    while (i<=n) {  
        r=r*i;  
        ++i;  
    }  
    return r;  
}
```



Construcción del Grafo del Flujo de Ejecución

- Pasos
 - Dividir el programa en bloques básicos
 - Se representa el programa en un código intermedio donde queden explícitamente representados los saltos condicionales e incondicionales.
 - Un bloque básico será cualquier trozo de código que no contenga saltos ni etiquetas en su interior (es posible tener etiquetas al inicio del bloque y saltos al final).
 - En el grafo, los vértices representan los bloques básicos y las aristas representan los saltos de un bloque básico a otro.

- (1) PROD = 0
- (2) I = 1
- (3) T2 = addr(A) - 4
- (4) T4 = addr(B) - 4
- (5) T1 = 4 x I
- (6) T3 = T2[T1]
- (7) T5 = T4[T1]
- (8) T6 = T3 x T5
- (9) PROD = PROD + T6
- (10) I = I + 1
- (11) IF I <=20 GOTO (5)

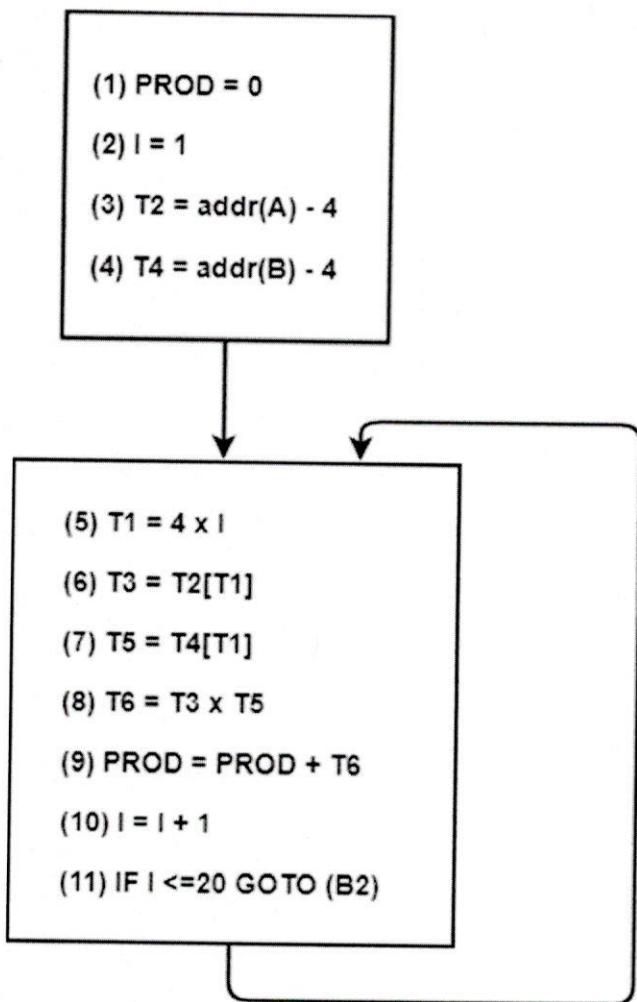
B1

- (1) PROD = 0
- (2) I = 1
- (3) T2 = addr(A) - 4
- (4) T4 = addr(B) - 4

B2

- (5) T1 = 4 x I
- (6) T3 = T2[T1]
- (7) T5 = T4[T1]
- (8) T6 = T3 x T5
- (9) PROD = PROD + T6
- (10) I = I + 1
- (11) IF I <=20 GOTO (5)

Flow Graphs



Detección de Código no Utilizado (Código Muerto)

- Código muerto: es código que nunca se ejecutará.
- Detección de código no utilizado
 - El código no utilizado son los bloques básicos donde no llega ninguna arista.
 - Se quita el bloque y las aristas que salen de él.
 - Repetir hasta no eliminar ningún bloque básico.