# Optimisation du code

Section 9.1

# * Contenu *

- Transformations à l'intérieur d'un bloc
  - Élimination de sous-expressions communes
  - Élimination de code mort
  - Réordonnancement
  - Arithmétique
- Transformations sur plusieurs blocs
  - Élimination d'expressions redondantes
  - Propagation de copie et de valeur
- Transformations sur les boucles
  - Déplacement de code
  - Changement de variables d'induction
  - Affaiblissement
- Exemple complet

# Introduction

Ce que nous entendons par *optimisation du code* consiste en fait à améliorer le code plutôt qu'à produire un code véritablement optimal.

On peut améliorer la qualité d'un programme compilé en apportant des modifications à différents niveaux:

- **Le code source.** L'*usager* peut effectuer le profilage du programme, modifier les algorithmes et effectuer des transformations du code dans les parties principales du programme (boucles).
- **Le code intermédiaire.** Le *compilateur* peut améliorer le code des boucles et des appels de fonctions.
- **Le code cible.** Le *compilateur* peut choisir d'utiliser les registres de la machine, sélectionner les instructions avantageusement et effectuer des optimisations *peephole*.

# Introduction

Dans cette section, nous nous intéressons à l'optimisation du code intermédiaire.

Normalement, un compilateur devrait fournir un effort d'optimisation particulier dans les parties du programme impliquant du calcul intensif. (Les boucles sont visées.)

Au minimum, une optimisation devrait améliorer le programme *la plupart du temps*.

L'implanteur du compilateur doit évaluer la *rentabilité* de toute nouvelle optimisation, i.e. coût d'implantation versus amélioration espérée des programmes.

Une optimisation ne doit jamais modifier le code de façon à ce que l'exécution du programme cible ne soit plus compatible avec l'exécution du programme source.

# Fragment de programme source

```
void merge(int m, int n)
{
  i := 0;  j := 0;  k :=0;
  while (i < m and j < n)
    {
      if (a[i] < b[j])
        { c[k] := a[i];  i++;  k++; }
      else
        { c[k] := b[j];  j++;  k++; }
    }
  while (i < m)
    { c[k] := a[i];  i++;  k++; }
  while (j < n)
    { c[k] := b[j];  j++;  k++; }
}
```

Les tableaux ‘a’ et ‘b’ contiennent respectivement ‘m’ et ‘n’ éléments déjà triés. Le tableau ‘c’ reçoit la fusion de ces deux séquences croissantes.

# Code à trois adresses correspondant

```
(1)     i := 0
(2)     j := 0
(3)     k := 0
(4)   if i >= m goto (25)
(5)   if j >= n goto (25)
(6)     t1 := 4 * i
(7)     t2 := a[t1]
(8)     t3 := 4 * j
(9)     t4 := b[t3]
(10)  if t2 >= t4 goto (18)
(11)    t5 := 4 * i
(12)    t6 := a[t5]
(13)    t7 := 4 * k
(14)  c[t7] := t6
(15)    i := i + 1
(16)    k := k + 1
(17)  goto (4)
(18)    t8 := 4 * j
(19)    t9 := b[t8]
(20)   t10 := 4 * k
(21)  c[t10] := t9
(22)    j := j + 1
(23)    k := k + 1
(24)  goto (4)
(25)  if i >= m goto (33)
(26)    t11 := 4 * i
(27)    t12 := a[t11]
(28)    t13 := 4 * k
(29)  c[t13] := t12
(30)    i := i + 1
(31)    k := k + 1
(32)  goto (25)
(33)  if j >= n goto (41)
(34)    t14 := 4 * j
(35)    t15 := b[t14]
(36)    t16 := 4 * k
(37)  c[t16] := t15
(38)    j := j + 1
(39)    k := k + 1
(40)  goto (33)
(41)        ...
```

# Transformations sur les blocs de base

On peut voir un bloc de base comme étant le calcul de plusieurs expressions. Les résultats sont les valeurs laissées dans les variables qui sont vivantes à la sortie du bloc.

On dit que deux blocs de base sont *équivalents* s'ils produisent les mêmes résultats dans les variables vivantes en sortie.

On peut optimiser un bloc de base en le transformant en un "meilleur" bloc, lequel doit être équivalent. L'optimisation de programmes bloc par bloc est très courante.

On retrouve deux classes importantes de transformations sur les blocs de base: les *transformations qui préservent la structure* et les *transformations arithmétiques*.

# Transformations sur les blocs de base

Voici quatre transformations qui préservent la structure des blocs de base:

- Élimination des sous-expressions communes (présentée à la page 11).
- Élimination du code mort (présentée aux pages 17–18).
- Renommage des variables temporaires. Il faut modifier le nom de la variable en sa définition et en toutes ses utilisations.
- Réordonnancement d'instructions indépendantes. Exemple:

$$\begin{array}{l} \texttt{t}_1 \texttt{ := b + c} \\ \texttt{t}_2 \texttt{ := x + y} \end{array} \quad \Rightarrow \quad \begin{array}{l} \texttt{t}_2 \texttt{ := x + y} \\ \texttt{t}_1 \texttt{ := b + c} \end{array}$$

# Transformations sur les blocs de base

Les transformations arithmétiques consistent à remplacer les instructions coûteuses par des instructions équivalentes moins coûteuses ou à éliminer des instructions sans effet.

```
x := x + 0      ⇒      —
x := x * 1      ⇒      —
x := y ** 2     ⇒      x := y * y
x := 16 * x     ⇒      x := x << 4
```

(en nombres entiers sur certaines machines)

```
                       t := x << 3
                       t := t - x
t := 57 * x     ⇒      t := t << 3
                       t := t + x
```

(en nombres entiers sur certaines machines)

# Optimisations préservant le comportement

On dit d'une optimisation qu'elle est:

- *locale* lorsqu'on l'applique uniquement à l'échelle d'un bloc de base à la fois;
- *globale* lorsqu'on l'applique à travers plusieurs ou tous les blocs de base à la fois.

Parmi les optimisations préservant le comportement, nous retrouvons:

- l'élimination des expressions redondantes,
- la propagation de copies,
- la propagation de constantes et
- l'élimination du code mort.

# Élimination des expressions redondantes

Exemple:

```
a := b + c          a := b + c
b := a - d    ⇒     b := a - d
c := b + c          c := b + c
d := a - d          d := b
```

Considérons l'extrait de programme suivant:

$$\begin{array}{l} \texttt{t}_i \texttt{ := } E \\ \texttt{...} \\ \texttt{t}_j \texttt{ := } E \end{array}$$

L'expression $E$ dans la deuxième instruction est considérée comme *redondante* si le contrôle doit toujours passer par la première instruction avant de se rendre à la seconde et si ni les variables dans $E$ ni $\texttt{t}_i$ ne sont modifiées sur le chemin entre les deux.

On peut alors éliminer un des deux calculs de $E$ en remplaçant la seconde instruction par:

$$\texttt{t}_j \texttt{ := t}_i$$

# Élimination des expressions redondantes

**Exemples 9.1**. Soit le code intermédiaire suivant:

```
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
```

Après l'élimination de sous-expressions communes (et quelques autres optimisations), on obtient:

```
t6 = 4 * i
x = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x
```

# Propagation des copies (et des constantes)

Considérons l'extrait de programme suivant:

$$
\begin{aligned}
&\texttt{t}_i \texttt{ := } \ldots \\
&\ldots \\
&\texttt{t}_j \texttt{ := t}_i \\
&\ldots \\
&\texttt{t}_k \texttt{ := } E(\texttt{t}_j)
\end{aligned}
$$

Si la variable $\texttt{t}_i$ originalement calculée par la première instruction demeure disponible et que $\texttt{t}_j$ demeure inchangée, alors on peut récrire la dernière instruction ainsi:

$$\texttt{t}_k \texttt{ := } E(\texttt{t}_i)$$

Ultimement, grâce à ce type de transformation, on pourrait rendre l'instruction de copie inutile.

# Propagation (des copies et) des constantes

Considérons l'extrait de programme suivant:

$$\begin{aligned} &\texttt{t}_j \texttt{ := } 18 \\ &\ldots \\ &\texttt{t}_k \texttt{ := } E(\texttt{t}_j) \end{aligned}$$

Si la variable $\texttt{t}_j$ demeure disponible, alors on peut récrire la dernière instruction ainsi:

$$\texttt{t}_k \texttt{ := } E(18)$$

# Propagation des copies et des constantes

## Exemple 9.1 (plus détaillé)

1. Code original

```
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
```

2. Sous-expressions communes

```
t6 = 4 * i
x = a[t6]
t7 = t6
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = t8
a[t10] = x
```

3. Propagation des copies

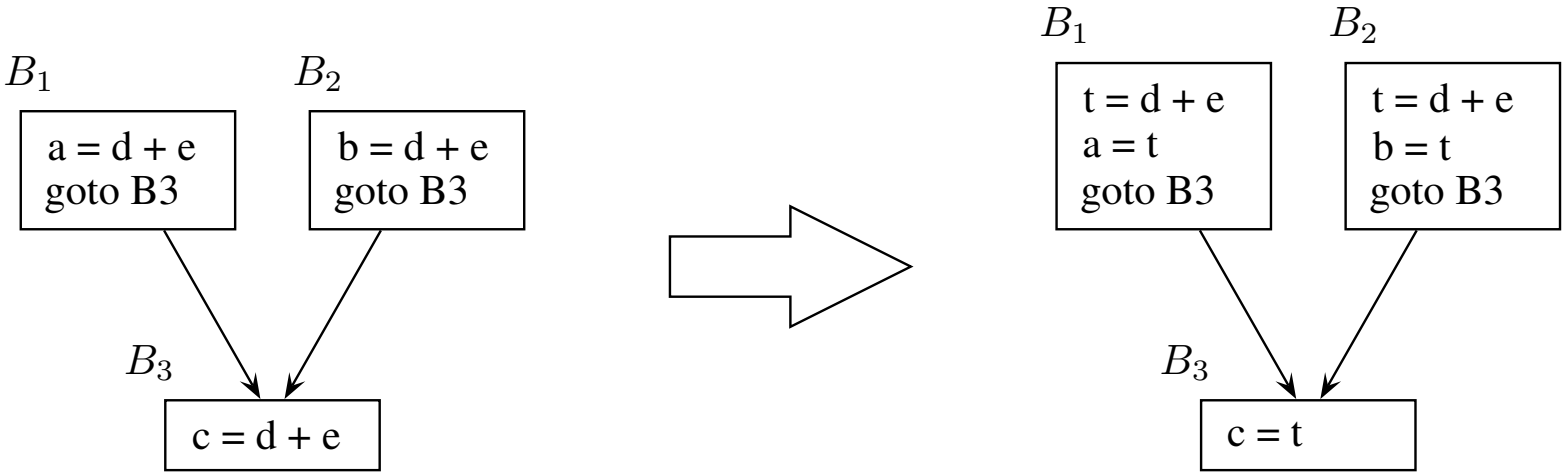
```
t6 = 4 * i
x = a[t6]
t7 = t6
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
t10 = t8
a[t8] = x
```

4. Élimination des variables mortes (t7 et t10)

```
t6 = 4 * i
x = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x
```

# Propagation des copies et des constantes

**Exemple 9.3**. Propagation de copies sur plusieurs blocs




Lorsqu'on arrive à $B_3$, l'expression $d+e$ est déjà calculée, mais est soit dans $a$, soit dans $b$. Il faut donc introduire une variable supplémentaire $(t)$ pour pouvoir récupérer la valeur de $d + e$ peu importe par quel bloc l'exécution a passé pour atteindre $B_3$.

# Élimination du code mort / variables mortes

Exemple: `x := y + z` $\Rightarrow$ — où `x` est morte

Une instruction `x := y` *op* `z` peut être éliminée si:

- la variable `x` est morte et
- l'opération *op* ne produit pas d'effet de bord.

# Élimination du code mort / code inaccessible

Des instructions qui sont inaccessibles à cause d'instructions de saut sont considérées comme du code mort et peuvent être éliminées:

```
if 1 <> 0 goto label
i := a[k]
k := k + 1
label:
```

**Exemple 9.4**. Considérons le code suivant:

```
if (debug) printf("...");
```

Si une optimisation de propagation de valeur transforme le code ainsi:

```
if (false) printf("...");
```

alors cette ligne peut être complètement retirée puisque l'appel à `printf` ne s'effectuera jamais.

# Optimisations de boucle

L'optimisation des boucles est particulièrement intéressante car l'intérieur des boucles est typiquement exécuté intensivement.

Les transformations que l'on peut appliquer aux boucles incluent:

- le déplacement de code hors des boucles et
- l'identification et l'élimination des variables d'induction,
- l'affaiblissement d'instructions.

# Déplacement de code

Un calcul interne à une boucle et dont les opérandes sont constants dans la boucle est appelé *invariant de boucle* et peut être déplacé en dehors de la boucle.

Supposons que $E$ soit un calcul invariant dans la boucle suivante:

```
while (...)
  ...
  x := E
  ...
```

Alors, on peut déplacer le calcul de $E$ en dehors de la boucle:

```
t123 := E
while (...)
  ...
  x := t123
  ...
```

# Déplacement de code

**Exemple 9.5**. Le code suivant:

```
while (i <= limit - 2)
  { /* le corps de la boucle ne modifie pas ‘limit’ */ }
```

peut être réécrit ainsi:

```
t = limit - 2
while (i <= t) { /* ... */ }
```

# Variables d'induction

Une variable dont la valeur augmente ou diminue de façon régulière à chaque itération d'une boucle est appelée *variable d'induction*.

Parfois, deux variables d'induction '`i`' et '`j`' sont synchronisées. C'est-à-dire qu'il existe une fonction linéaire liant les deux, i.e. telle que $\texttt{j} = a \times \texttt{i} + b$.

Dans de tels cas, on peut parfois éliminer une des deux variables d'induction (disons '`j`') en réinterprétant les opérations sur '`j`' en fonction de '`i`'.

**Exemple:**

*Avant*:

```
i := 2
j := 1
...
while (j <= 17)
  ...
  i := i + 1
  j := j + 2
  ...
```

*Après*:

```
i := 2
j := 1
...
while (i <= 10)
  ...
  i := i + 1
  j := j + 2
  ...
```

# Affaiblissement d'instruction

Lorsque l'on a deux variables d'induction qui sont liées par un calcul considéré coûteux comme dans:

```
while (...)
  ...
  i := i + 1
  x := 4 * i
  ...
```

alors on affaiblir l'instruction utilisée pour faire le calcul et espérer accélérer le code:

```
x := 4 * i
while (...)
  ...
  i := i + 1
  x := x + 4
  ...
```

# Affaiblissement d'instruction

**Exemple**. Soit le code suivant: `for (i = 0; i < limit; i++) a[i] = 0;`

**Nous générons le code intermédiaire suivant:**

```
    i := 0
    goto Lverification

Lboucle:
    t0 := a.type.elem.width * i
    a[t0] := 0
    i := i + 1

Lverification:
    if (i < limit) goto Lboucle
```

**Affaiblissement:**

```
    i := 0
    t0 := 0 // i * a.type.elem.width
    goto Lverification
Lboucle:

    a[t0] := 0
    i := i + 1
    t0 := t0 + a.type.elem.width
Lverification:
    if (i < limit) goto Lboucle
```

**Changement de variable d'induction:**

```
    i := 0
    t0 := 0

    goto Lverification
Lboucle:
    a[t0] := 0
    i := i + 1
    t0 := t0 + a.type.elem.width
Lverification:
    t1 := limit * a.type.elem.width
    if (t0 < t1) goto Lboucle
```

**Variable morte et invariant:**

```
    t0 := 0
    t1 := limit * a.type.elem.width
    goto Lverification
Lboucle:
    a[t0] := 0

    t0 := t0 + a.type.elem.width
Lverification:

    if (t0 < t1) goto Lboucle
```

Optimisation:

# Exemple complet

# Exemple

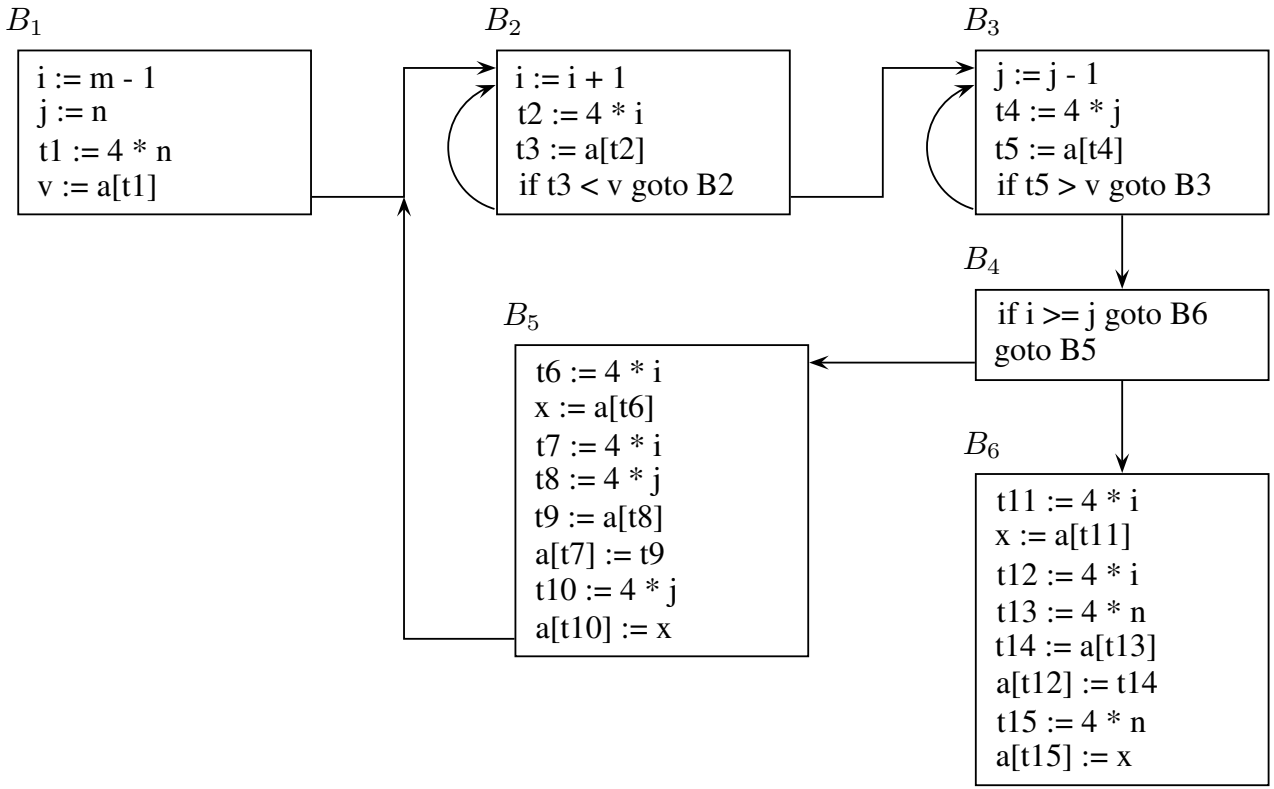
Soit le code suivant:

```
void quicksort(int m, int n)
{
    int i, j;
    int v, x;
    if (n <= m) return;

    /* Début du fragment */
    i = m - 1;
    j = n;
    v = a[n];
    while (true)
    {
        do { i = i + 1; }
            while (a[i] < v);
        do { j = j - 1; }
            while (a[j] > v);

        if (i >= j) break;
        x    = a[i];   // interchanger a[i] et a[j]
        a[i] = a[j];
        a[j] = x;
    }

    x    = a[i];   // interchanger a[i] et a[n]
    a[i] = a[n];
    a[n] = x;
    /* Fin du fragment */

    quicksort(m, j);
    quicksort(i + 1, n);
}
```

Voici le code intermédiaire correspondant:

```
 (1) i := m - 1
 (2) j := n
 (3) t1 := 4*n
 (4) v := a[t1]
 (5) i := i + 1
 (6) t2 := 4*i
 (7) t3 := a[t2]
 (8) if t3 < v goto (5)
 (9) j := j - 1
(10) t4 := 4*j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*i
(15) x := a[t6]        (23) t11 := 4*i
(16) t7 := 4*i         (24) x := a[t11]
(17) t8 := 4*j         (25) t12 := 4*i
(18) t9 := a[t8]       (26) t13 := 4*n
(19) a[t7] := t9       (27) t14 := a[t13]
(20) t10 := 4*j        (28) a[t12] := t14
(21) a[t10] := x       (29) t15 := 4*n
(22) goto (5)          (30) a[t15] := x
```

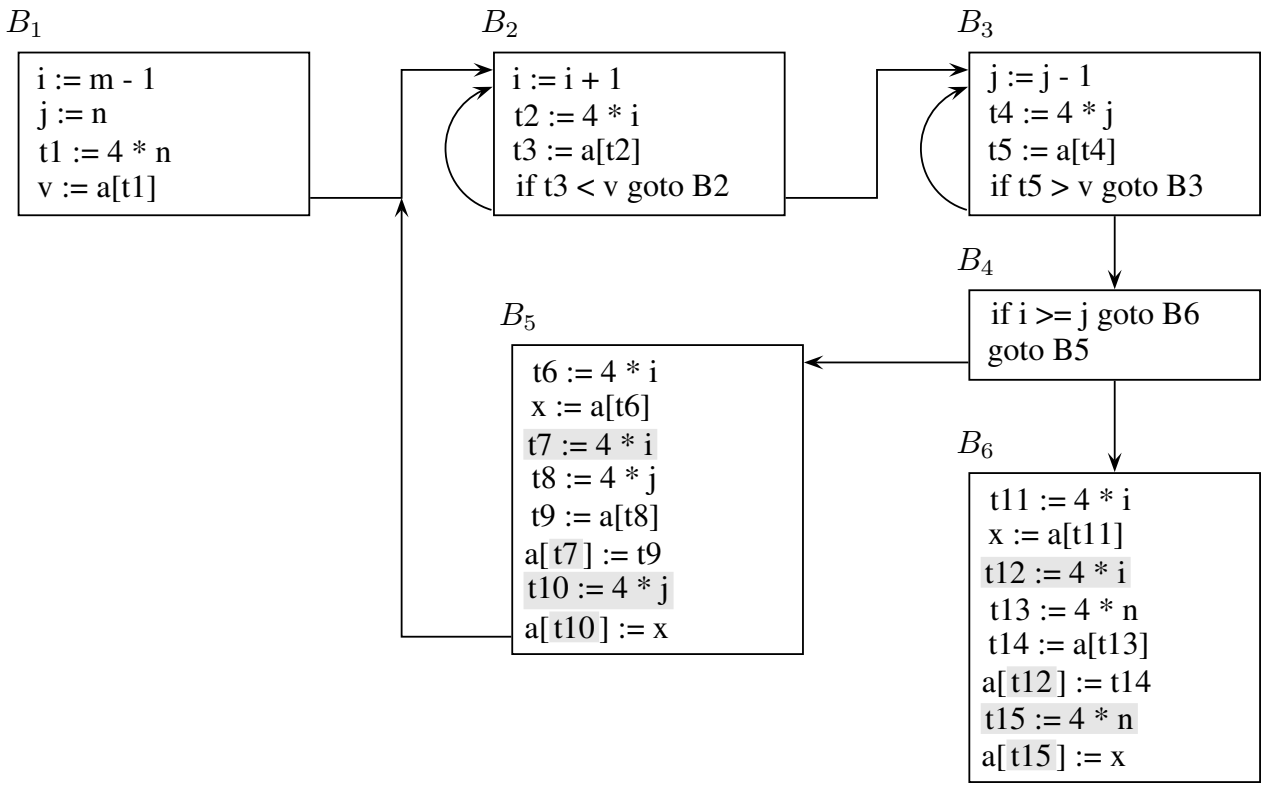
# Exemple: graphe de flot




Boucles:

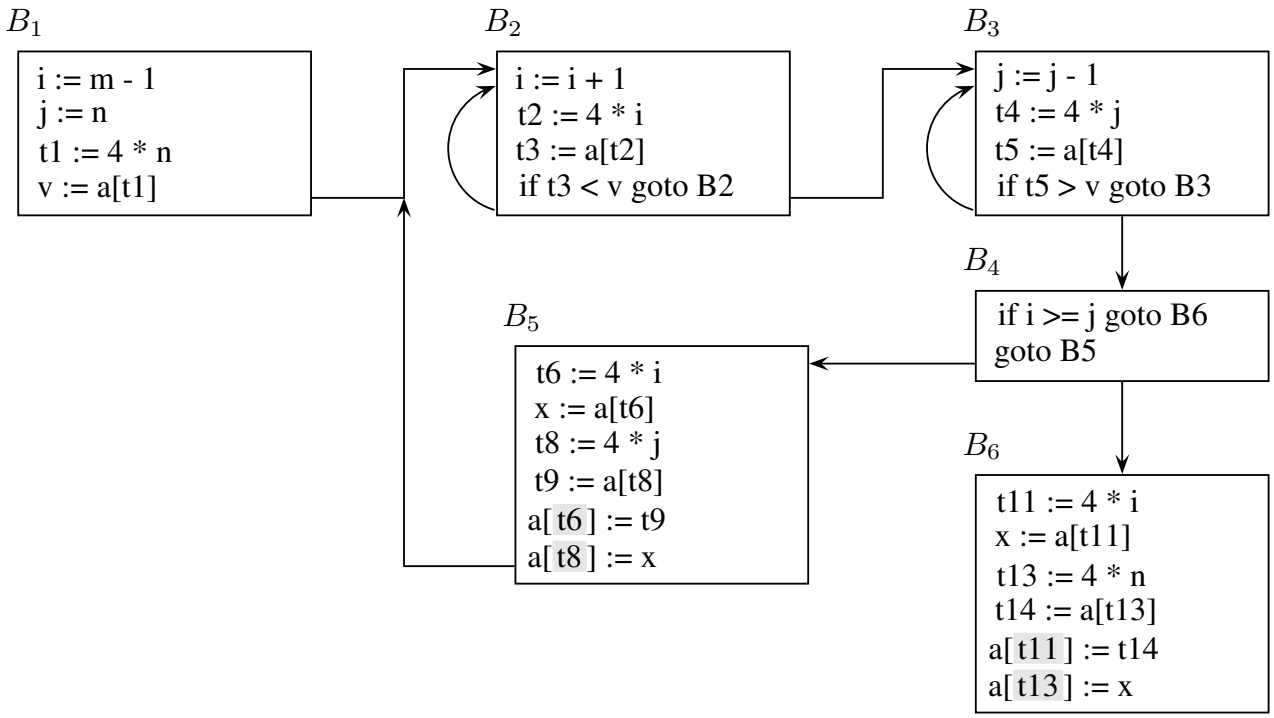
- $B_2$
- $B_3$
- $B_2$, $B_3$, $B_4$, $B_5$

# Exemple: Identification de sous-expressions locales communes

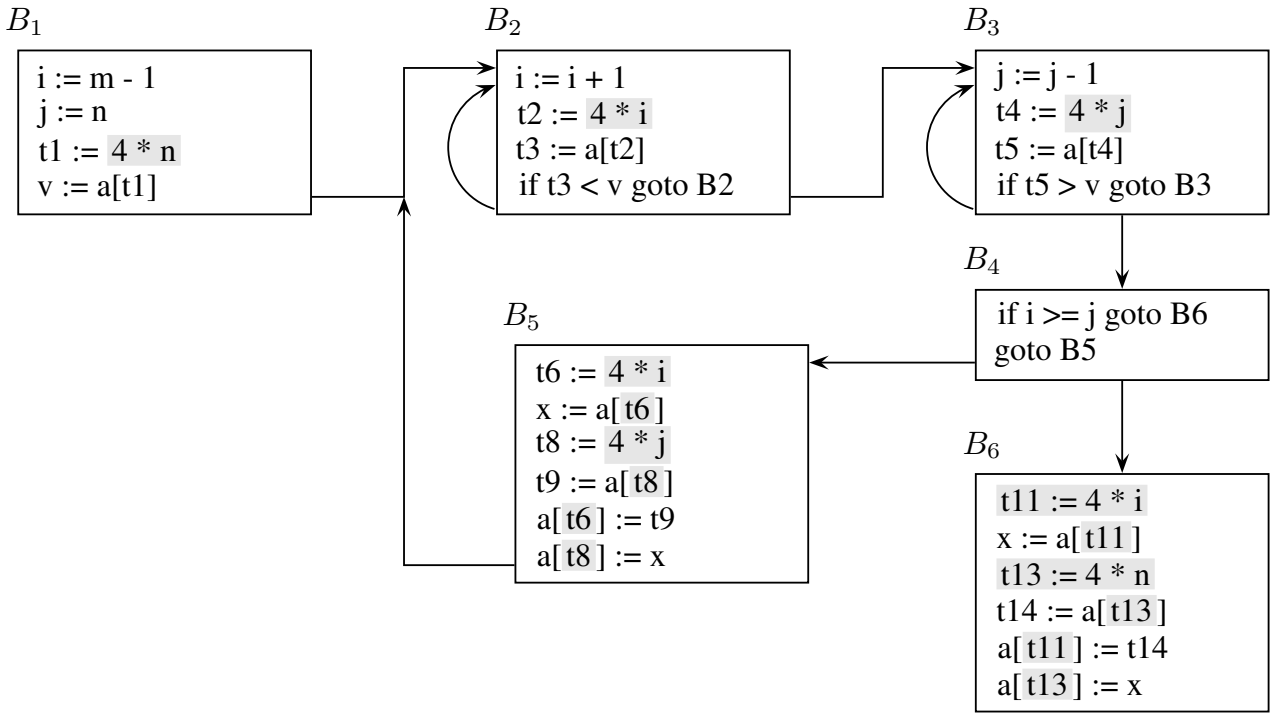



- $t_7$ a la même valeur que $t_6$, et $t_6$ conserve sa valeur, on pourra donc remplacer les occurrences de $t_7$ par $t_6$;
  - $t_7$ devient alors une variable morte;
- $t_{10}$ a la même valeur que $t_8$, et $t_8$ conserve sa valeur, on pourra donc remplacer les occurrences de $t_{10}$ par $t_8$;
  - $t_{10}$ devient alors une variable morte;
- $t_{12}$ a la même valeur que $t_{11}$, et $t_{11}$ conserve sa valeur, on pourra donc remplacer les occurrences de $t_{12}$ par $t_{11}$;
  - $t_{12}$ devient alors une variable morte;
- $t_{15}$ a la même valeur que $t_{13}$, et $t_{13}$ conserve sa valeur, on pourra donc remplacer les occurrences de $t_{15}$ par $t_{13}$;
  - $t_{15}$ devient alors une variable morte;

# Exemple: Propagation de copie + élimination de code mort

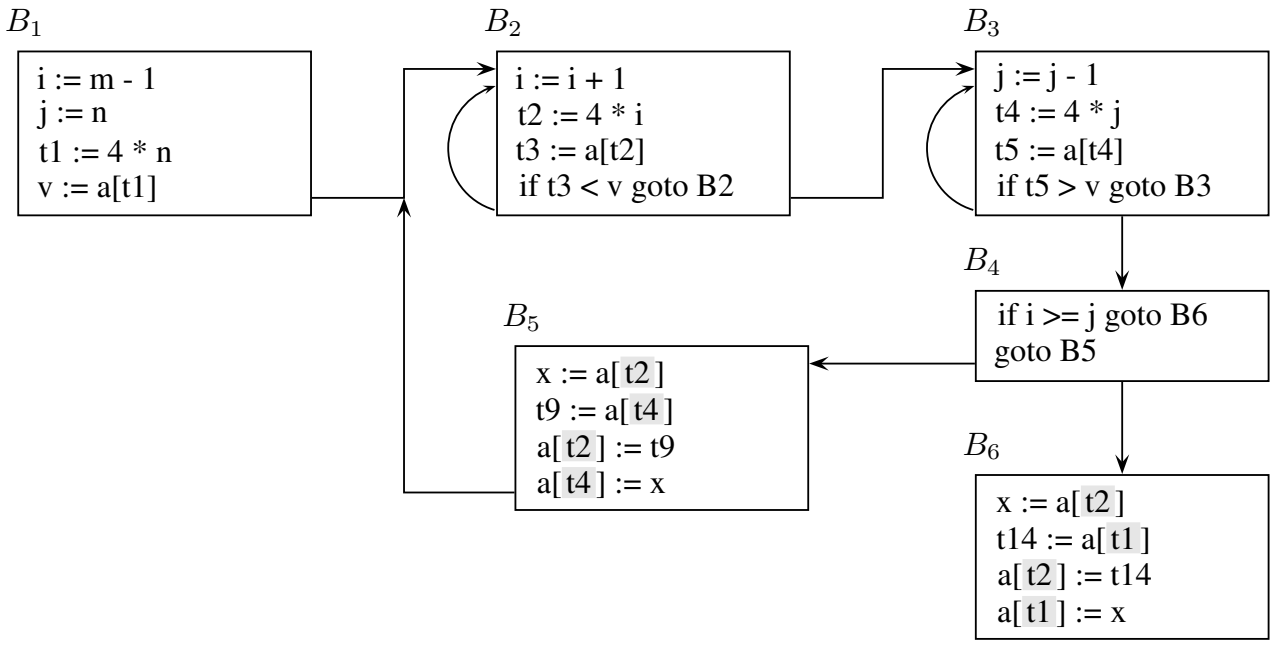


# Exemple: Identification de sous-expressions globales communes

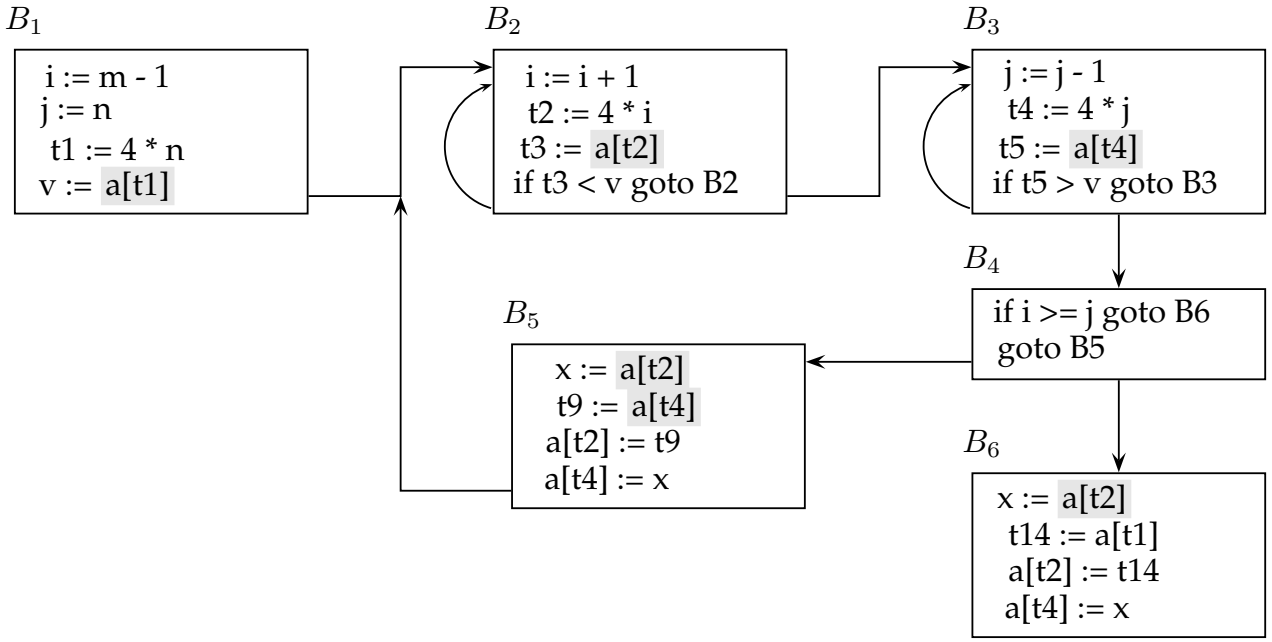



- $t_6$ a la même valeur que $t_2$, et $t_2$ conserve la bonne valeur de $B_2$ à $B_5$, on pourra donc remplacer $t_6$ par $t_2$;
  - $t_6$ devient alors une variable morte;
- $t_8$ a la même valeur que $t_4$, et $t_4$ conserve la bonne valeur de $B_3$ à $B_5$, on pourra donc remplacer $t_8$ par $t_4$;
  - $t_{13}$ devient alors une variable morte;
- $t_{11}$ a la même valeur que $t_2$, et $t_2$ conserve la bonne valeur de $B_2$ à $B_6$, on pourra donc remplacer $t_{11}$ par $t_2$;
  - $t_{11}$ devient alors une variable morte;
- $t_{13}$ a la même valeur que $t_1$, et $t_1$ conserve la bonne valeur de $B_1$ à $B_6$, on pourra donc remplacer $t_{13}$ par $t_1$;
  - $t_{13}$ devient alors une variable morte;

# Exemple: Propagation de copie + élimination de code mort

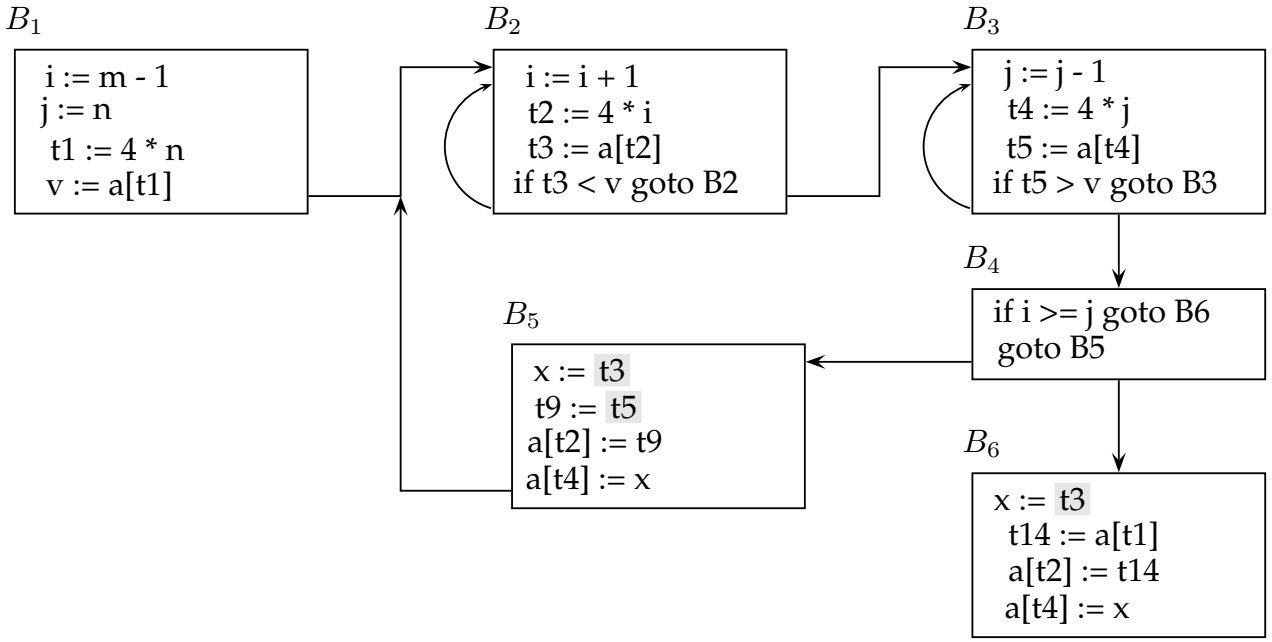


# Exemple: Identification de sous-expressions globales communes




- $t_3$ et $x$ dans $B_5$ sont calculés identiquement, $a[t_2]$ et $t_3$ demeurent inchangés de $B_2$ à $B_5$: on pourra donc remplacer $a[t_2]$ par $t_3$;
- $t_3$ et $x$ dans $B_6$ sont calculés identiquement, $a[t_2]$ et $t_3$ demeurent inchangés de $B_2$ à $B_6$: on pourra donc remplacer $a[t_2]$ par $t_3$;
- $t_5$ et $t_9$ sont calculés identiquement, $a[t_4]$ et $t_5$ demeurent inchangés de $B_3$ à $B_5$: on pourra donc remplacer $a[t_4]$ par $t_5$;
- même si $v$ et $t_{14}$ sont calculés identiquement, comme les éléments de $a$ ont été modifiés entre $B_1$ et $B_6$: on **ne** peut **pas** remplacer $a[t_1]$ par $v$ dans $B_6$.

# Exemple: Propagation de copie




On peut effectuer trois propagations de copie supplémentaires:

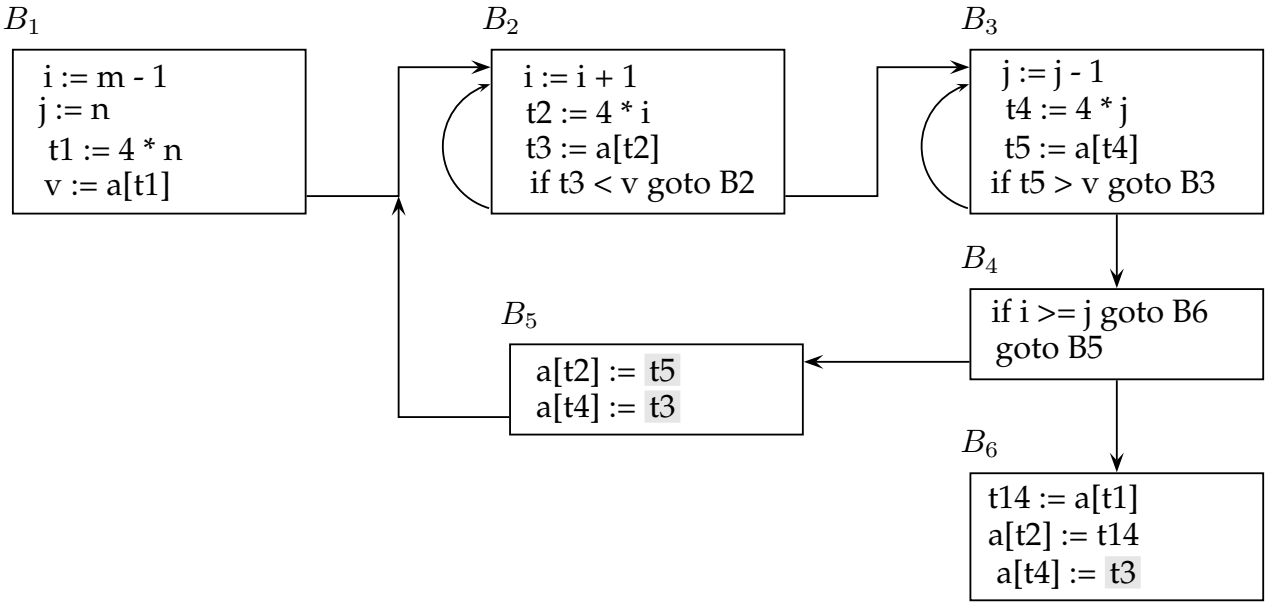
```
// Bloc B5          // Bloc B5          // Bloc B6
t9 := t5            x := t3             x := t3
a[t2] := t9         a[t4] := x          a[t4] := x
```

deviennent:

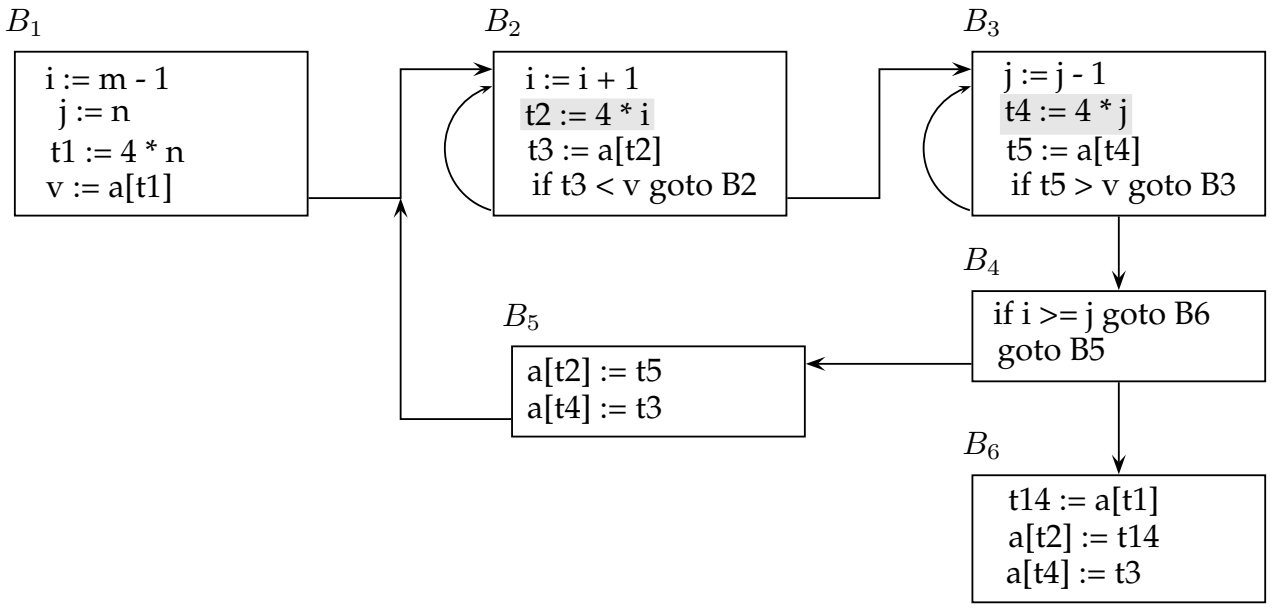
```
t9 := t5            x := t3             x := t3
a[t2] := t5         a[t4] := t3         a[t4] := t3
```

puis $x$ et $t_9$ deviennent des variables mortes et sont retirées.

# Exemple: Propagation de copie + élimination de code mort

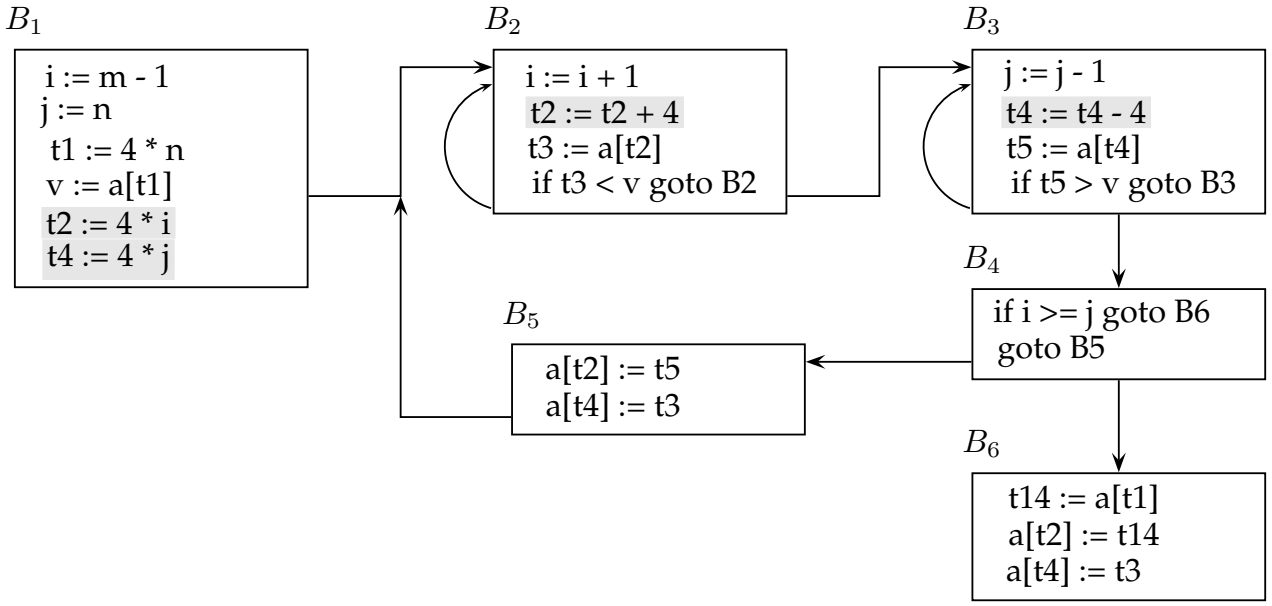


# Exemple: Opportunités d'affaiblissement




- $t_2$ est une variable d'induction "synchronisée" avec i (en tout temps, $t_2 = 4i$);
  - $t_2$ est calculée avec une multiplication, il serait moins couteux de la calculer avec une addition;
- $t_4$ est une variable d'induction "synchronisée" avec j (en tout temps, $t_4 = 4j$);
  - $t_4$ est calculée avec une multiplication, il serait moins couteux de la calculer avec une soustraction.

# Exemple: Affaiblissement



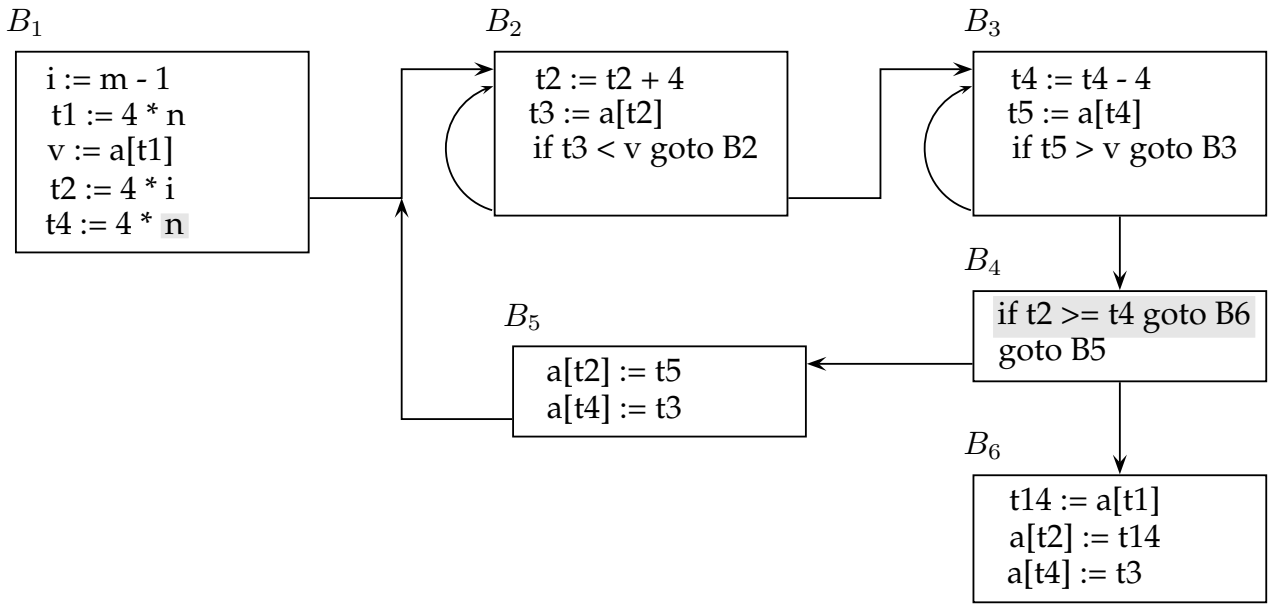

**Opportunité:** il est maintenant possible d'effectuer un changement de variable d'induction. $i$ et $j$ ne sont utilisés que dans l'expression `if i >= j`.

Or, $(i >= j) \equiv (4i >= 4j) \equiv (t_2 >= t_4)$.

On peut donc remplacer l'expression par `if t2 >= t4` et complètement retirer $i$ et $j$.

**De plus**, on peut propager la copie `j := n` dans `t4 := 4 * j`.

## Exemple: Changement de variable d'induction + Propagation de copie




**Opportunité:** l'expression $4 * n$ est calculée deux fois dans $B_1$.

# Exemple: Élimination d'expression redondante