AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Optymalizacja kodu pośredniego

Dr inż. Janusz Majewski
Języki formalne i automaty

Analiza przepływu, grafy obliczeń

- Graf obliczeń jest to skierowany graf tworzony na podstawie kodu trójadresowego. Krawędzie (ścieżki) w grafie wskazują możliwą kolejność wykonywania obliczeń. Wierzchołkami grafu są bloki podstawowe. Blok podstawowy jest ciągiem instrukcji trójadresowych, takich że jeśli sterowanie zostanie przekazane do pierwszej instrukcji tego bloku, to opuści blok po wykonaniu ostatniej instrukcji (nie będzie wewnątrz bloku skoków ani rozkazu stopu).

Analiza przepływu, grafy obliczeń

- Dzielenie kodu trójadresowego na bloki podstawowe :

(1) wyodrębniamy pierwsze instrukcje (liderów):

(a) pierwsza instrukcja kodu trójadresowego jest liderem

(b) każda instrukcja, do której prowadzi skok warunkowy lub bezwarunkowy jest liderem

(c) każda instrukcja, która następuje bezpośrednio po skoku warunkowym lub bezwarunkowym jest liderem

(2) każdy blok rozpoczyna się swoim liderem i zawiera wszystkie instrukcje, aż do napotkania kolejnego lidera lub końca kodu.

Analiza przepływu, grafy obliczeń

- Mając podział na bloki podstawowe tworzymy graf obliczeń uwzględniający możliwą kolejność wykonywania obliczeń. Pierwszy blok jest wyróżnionym wierzchołkiem początkowym w grafie. W grafie biegnie krawędź od wierzchołka B_i do B_j, jeśli:

  (a) istnieje skok warunkowy lub bezwarunkowy z ostatniej instrukcji B_i do pierwszej instrukcji B_j

  (b) B_j bezpośrednio następuje po B_i w kodzie trójadresowym, przy czym B_i nie kończy się skokiem bezwarunkowym.

Przykład: program źródłowy

Założenie: stała „s” zawiera *sizeof(int)*

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

Początek

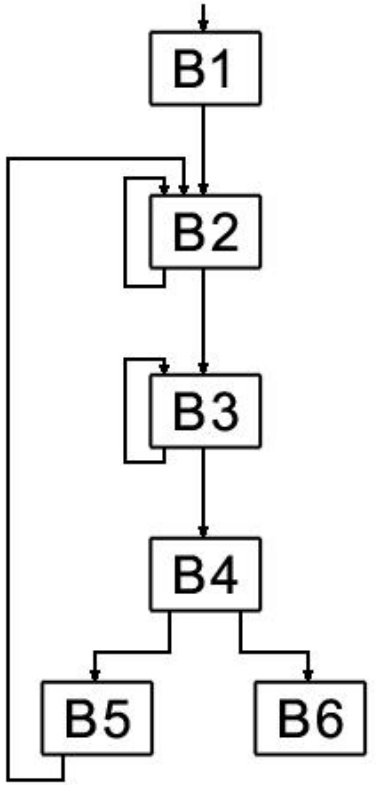
```
        i = m – 1;  j = n;   v = a[n];
        while(1)   {
                do i = i + 1;  while (a[i] < v );
                do j = j – 1;  while (a[j] > v );
                if ( i >= j ) break;
                x = a[i];  a[i] = a[j];  a[j] = x;
                }
        x = a[i];  a[i] = a[n];  a[n] = x;
```

Koniec

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

Przykład: tłumaczenie

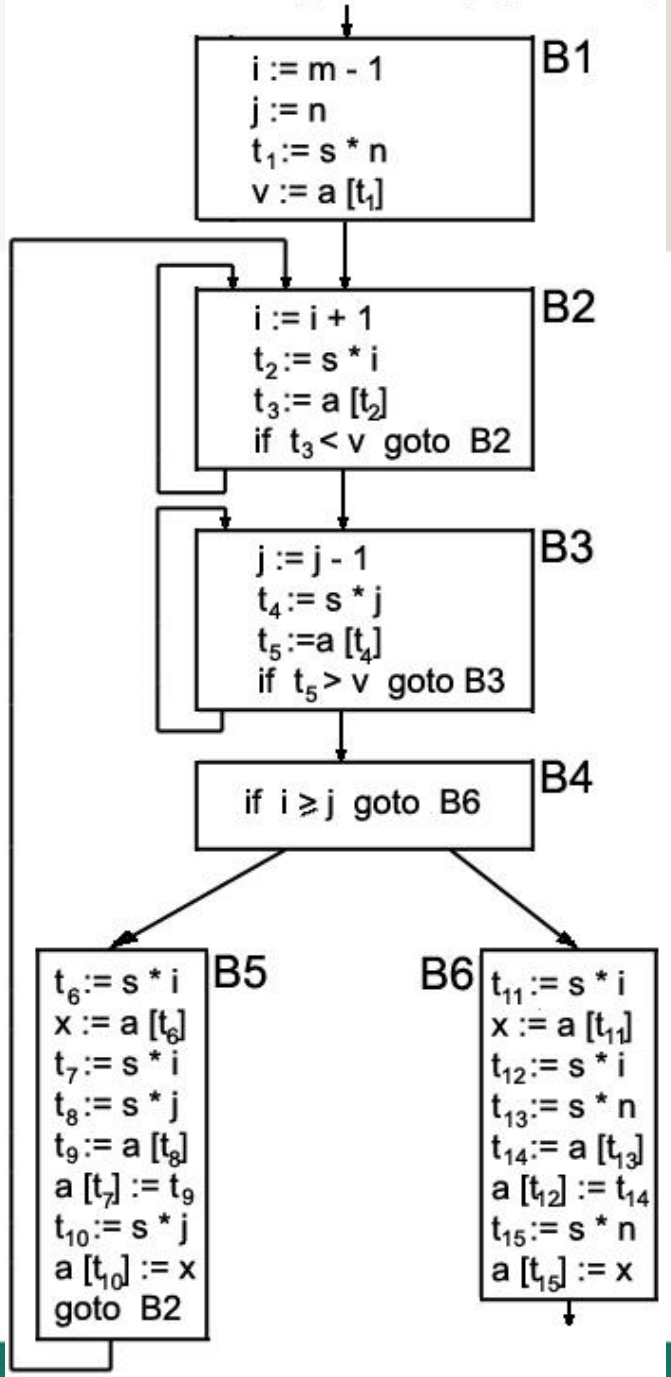
```
(1)  i   := m – 1
(2)  j   := n
(3)  t1  := s * n
(4)  v   := a[t1]                 B1

(5)  i   := i + 1
(6)  t2  := s * i
(7)  t3  := a[t2]
(8)  if t3 < v goto(5)            B2

(9)  j   := j – 1
(10) t4  := s * j
(11) t5  := a[t4]
(12) if t5 > v goto(9)            B3

(13) if i ≥ j goto(23)            B4

(14) t6  := s * i
(15) x   := a[t6]
(16) t7  := s * i
(17) t8  := s * j
(18) t9  := a[t8]
(19) a[t7] := t9
(20) t10 := s * j
(21) a[t10] := x
(22) goto(5)                      B5

(23) t11 := s * i
(24) x   := a[t11]
(25) t12 := s * i
(26) t13 := s * n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := s * n
(30) a[t15] := x                  B6
```



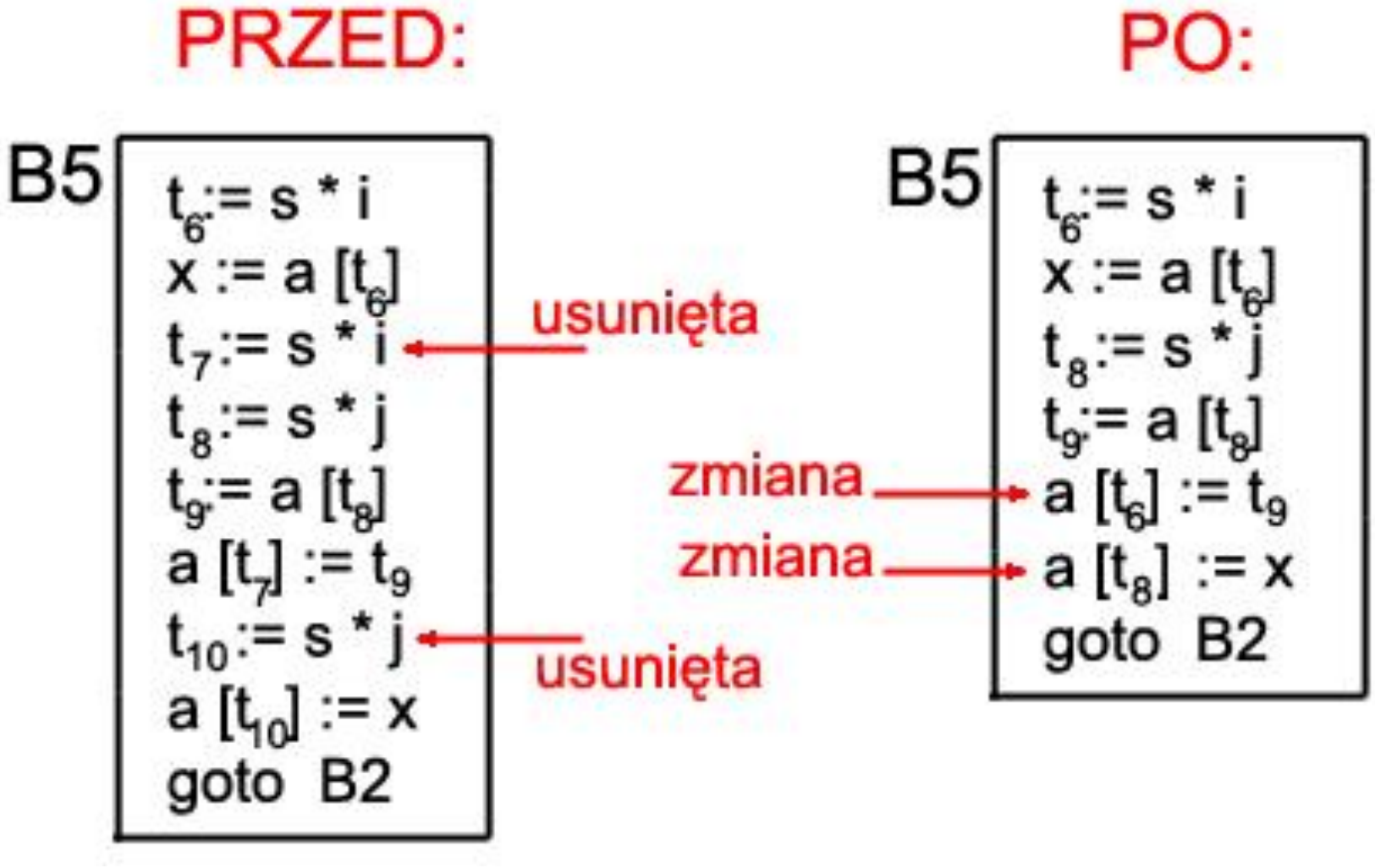
Graf obliczeń

AGH

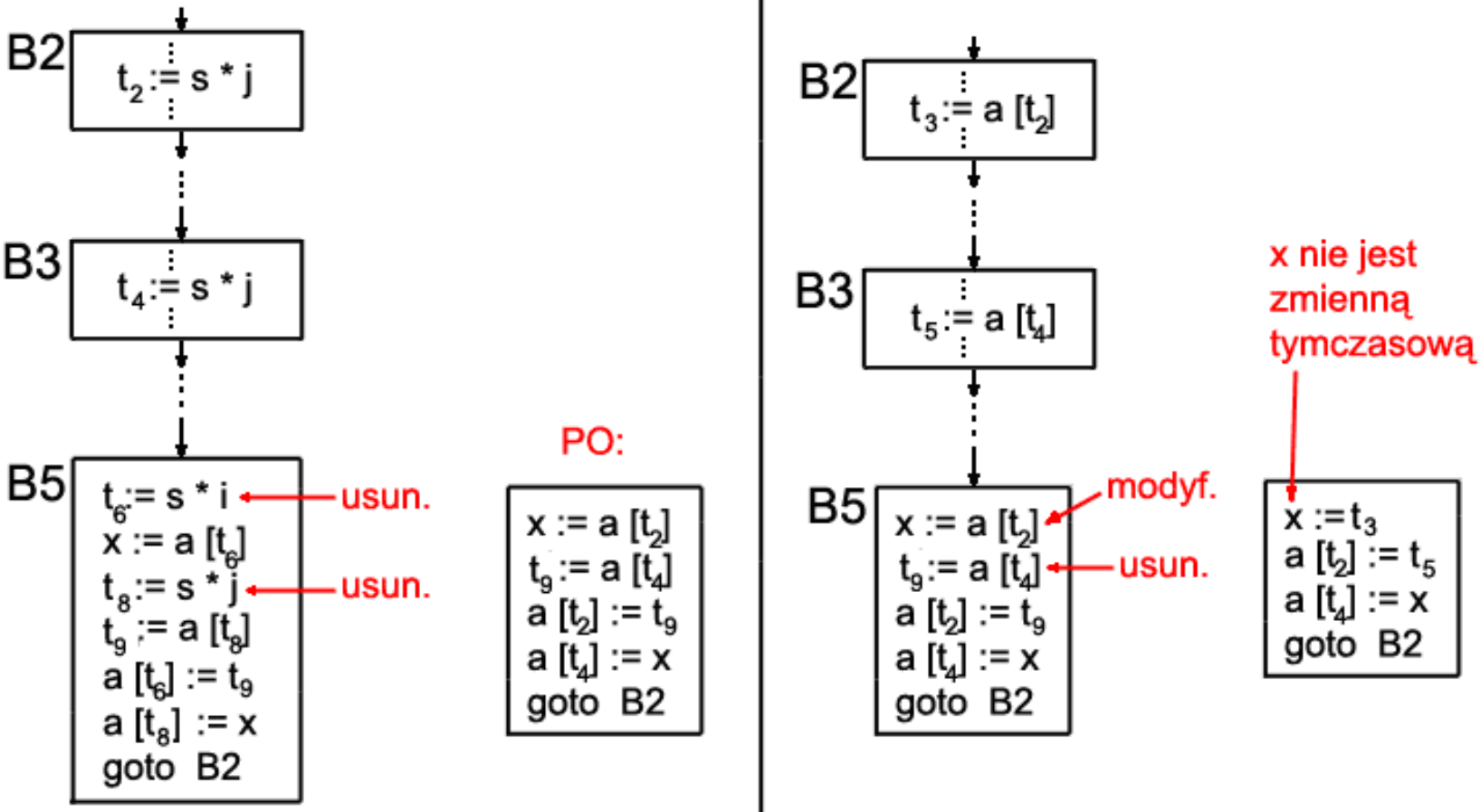
Graf obliczeń (przed optymaliz.)

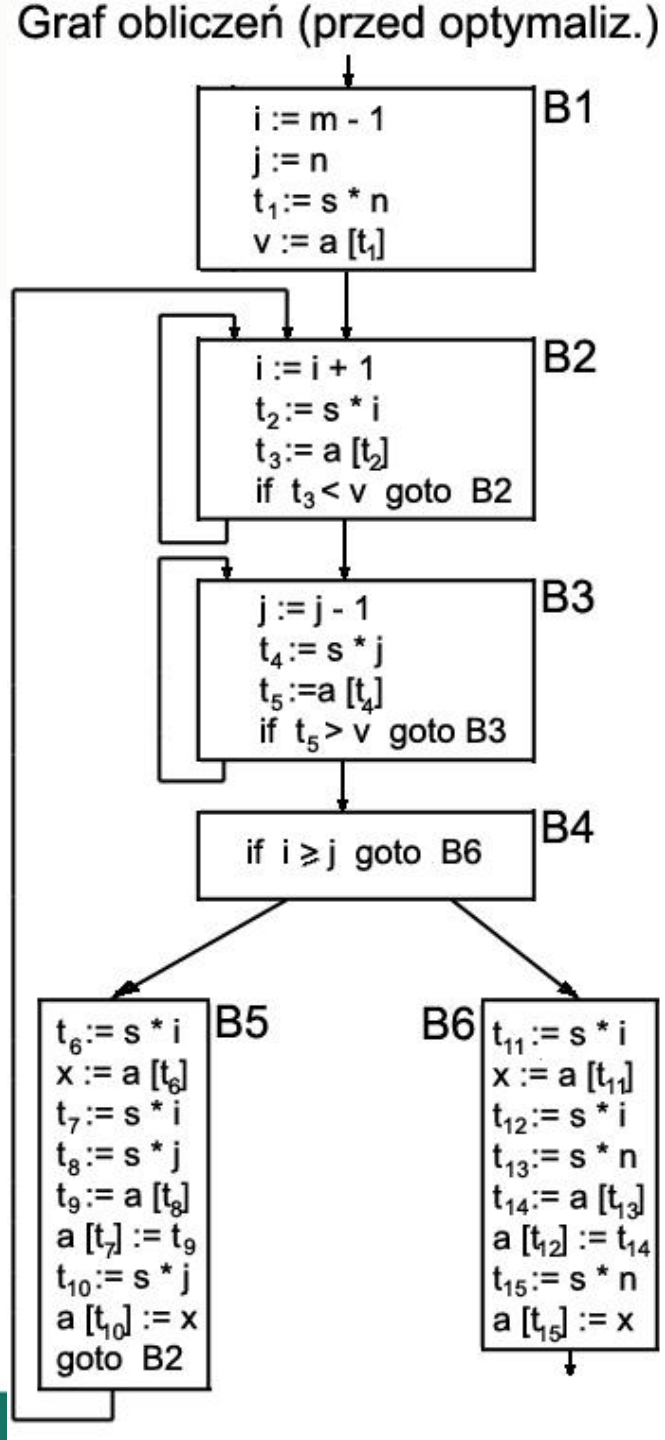


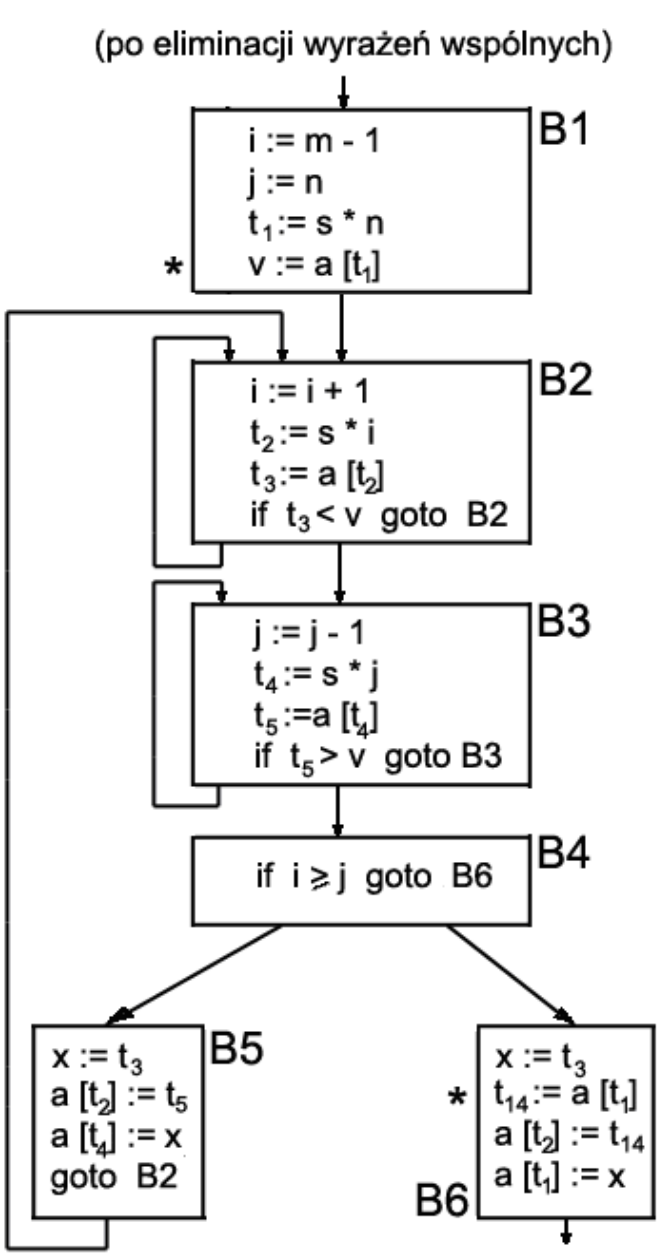
Lokalna eliminacja wspólnych podwyrażeń



Globalna eliminacja wspólnych podwyrażeń







Dlaczego pary instrukcji oznaczonych * nie uznano za wyrażenia wspólne?

Propagacja kopiowania

Można zastąpić:

```
b := a;  c := b
```

Przez:

```
b := a; c := a
```

W ten sposób „przerywamy” łańcuch propagacji kopiowania

PRZED:

B5:

```
x := t3
a[t2] := t5
a[t4] := x
goto B2
```

PO:

B5:

```
x := t3
a[t2] := t5
a[t4] := t3
goto B2
```

Pozornie nie przynosi to efektów, ale…

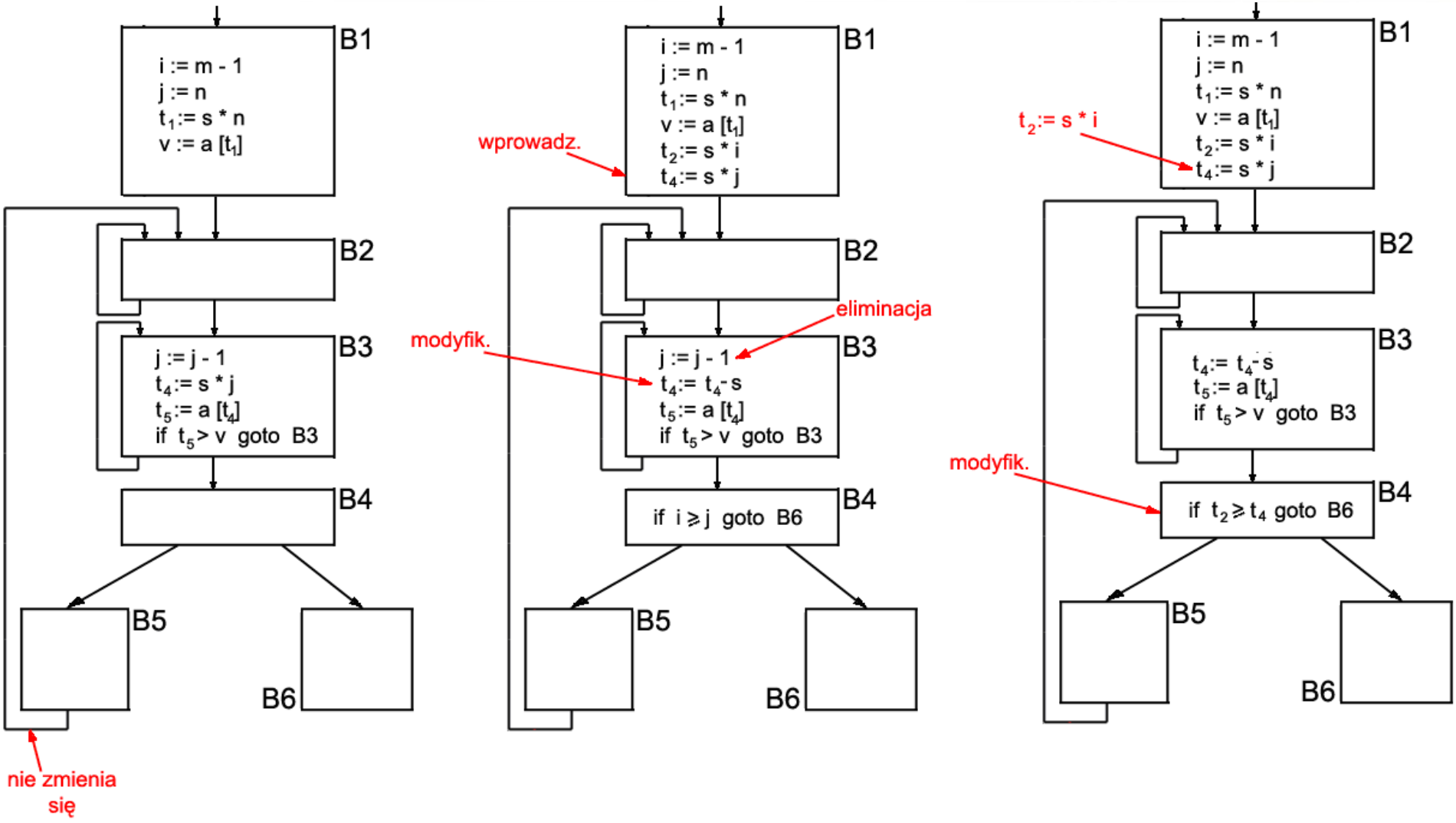
Eliminacja martwego kodu

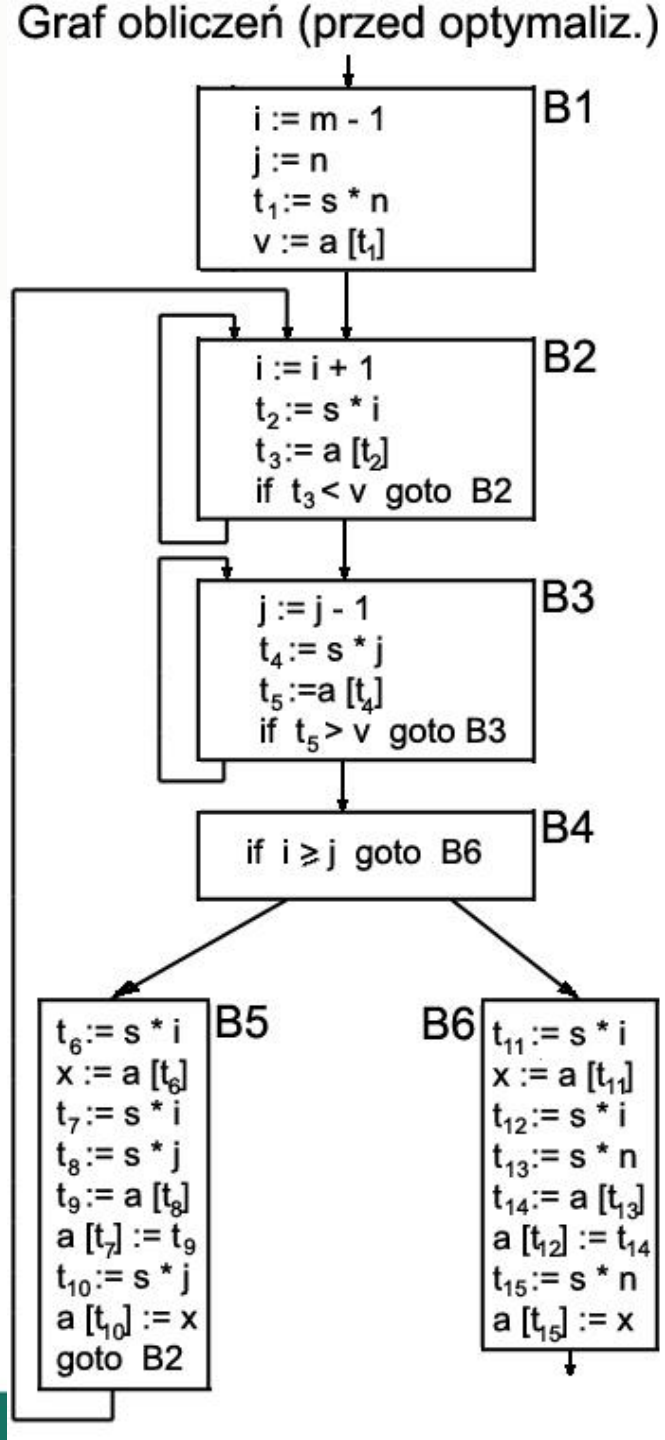
Eliminuje się te instrukcje trójadresowe, które nadają wartość takim zmiennym, które nie będą potem używane.

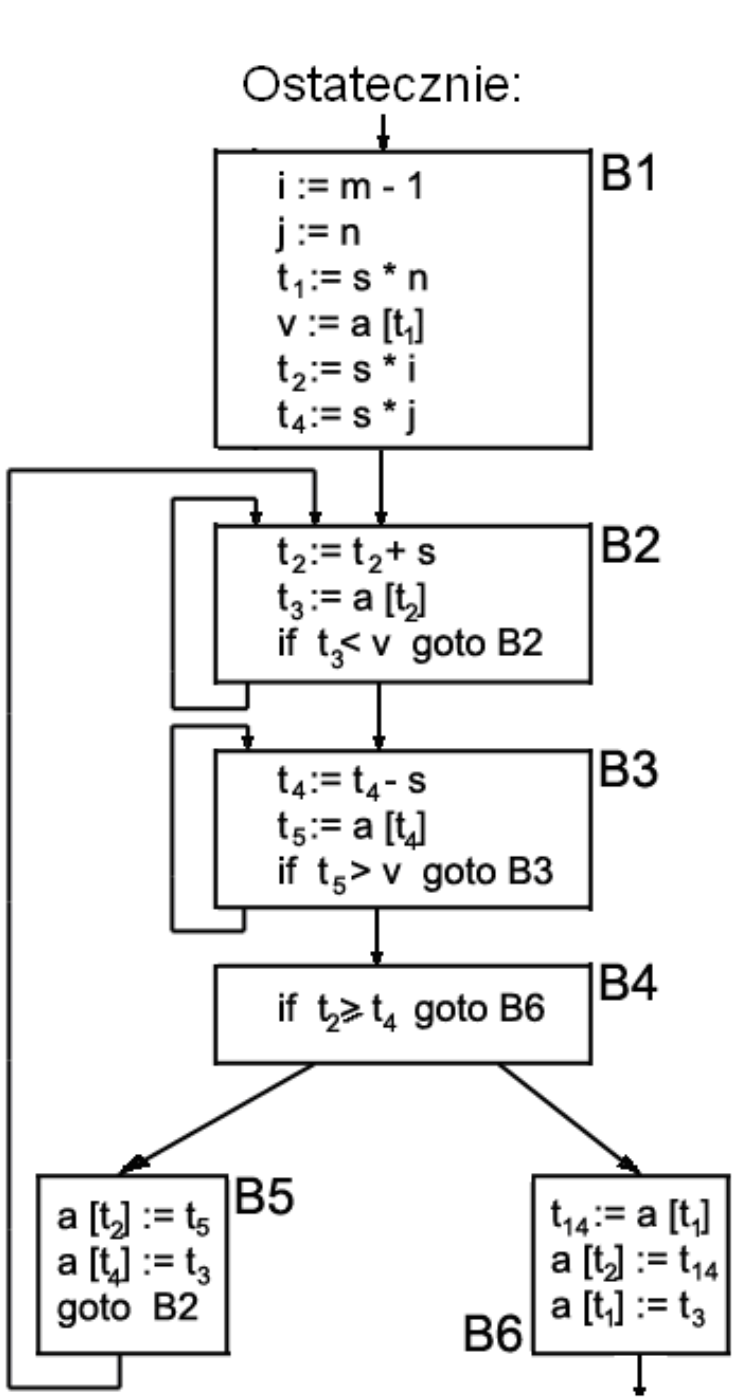
Zmienne indukcyjne w pętlach

- Zmienne indukcyjne: zmienne ściśle ze sobą związane w pętli; zmiana jednej z nich powoduje synchroniczną zmianę drugiej.
- Redukcja mocy: zastąpienie operacji dłużej wykonującej się operacją szybszą; przykład: zamiana mnożenia na dodawanie, zamiana mnożenia całkowitoliczbowego bez znaku na przesunięcie bitowe, zamiana potęgowania na mnożenie.
- Wykrycie zmiennych indukcyjnych pozwala na redukcję mocy kodu.
- Istnieje także możliwość eliminacji niektórych zmiennych indukcyjnych.

Zmienne indukcyjne - przykład




Graf obliczeń (przed optymaliz.)
B1
i := m - 1
j := n
t1 := s * n
v := a [t1]
B2
i := i + 1
t2 := s * i
t3 := a [t2]
if t3 < v goto B2
B3
j := j - 1
t4 := s * j
t5 := a [t4]
if t5 > v goto B3
B4
if i ≥ j goto B6
B5
t6 := s * i
x := a [t6]
t7 := s * i
t8 := s * j
t9 := a [t8]
a [t7] := t9
t10 := s * j
a [t10] := x
goto B2
B6
t11 := s * i
x := a [t11]
t12 := s * i
t13 := s * n
t14 := a [t13]
a [t12] := t14
t15 := s * n
a [t15] := x


Ostatecznie:
B1
i := m - 1
j := n
t1 := s * n
v := a [t1]
t2 := s * i
t4 := s * j
B2
t2 := t2 + s
t3 := a [t2]
if t3 < v goto B2
B3
t4 := t4 - s
t5 := a [t4]
if t5 > v goto B3
B4
if t2 ≥ t4 goto B6
B5
a [t2] := t5
a [t4] := t3
goto B2
B6
t14 := a [t1]
a [t2] := t14
a [t1] := t3

Przemieszczanie kodu niezmienniczego

Przykład :

```
while ( i <= limit - 2) { ...
   /* instrukcje nie zmieniające
    wartości zmiennej ”limit” */
}
```

może być przekształcone do postaci równoważnej:

```
t = limit - 2;
while ( i <= t ) { ...
   /* instrukcje nie zmieniające
    wartości zmiennych ”limit” i ”t” */
}
```

Optymalizacja „przez szparkę”

Peephole optimization – dosłownie: ”optymalizacja przez judasza”

- Eliminacja zbędnych skoków

PRZED:

```
        if a < b   goto L3
        goto L1
L1:     if c < d   goto L2
        goto L4
L2:     if e < f   goto L3
        goto L4
```

można wyeliminować

PO:

```
        if a < b   goto L3
L1:     if c < d   goto L2
        goto L4
L2:     if e < f   goto L3
        goto L4
```

Eliminacja zbędnych skoków

PRZED:

```
        if a < b   goto L3
L1:     if c < d   goto L2
        goto L4
L2:     if e < f   goto L3
        goto L4
```

można zmodyfikować „skok przez skok” zmieniając warunek na przeciwny

```
        if a < b   goto L3
L1:     if c >= d  goto L4
        goto L2
L2:     if e < f   goto L3
        goto L4
```

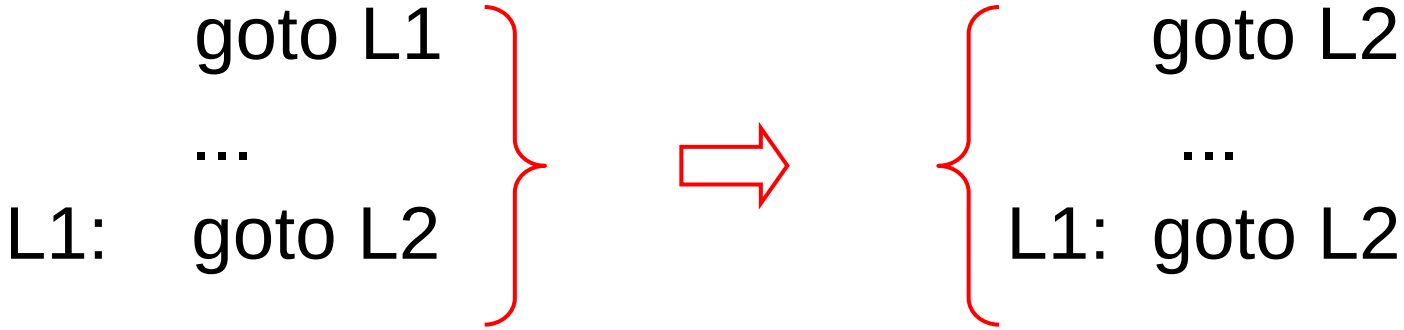
można wyeliminować

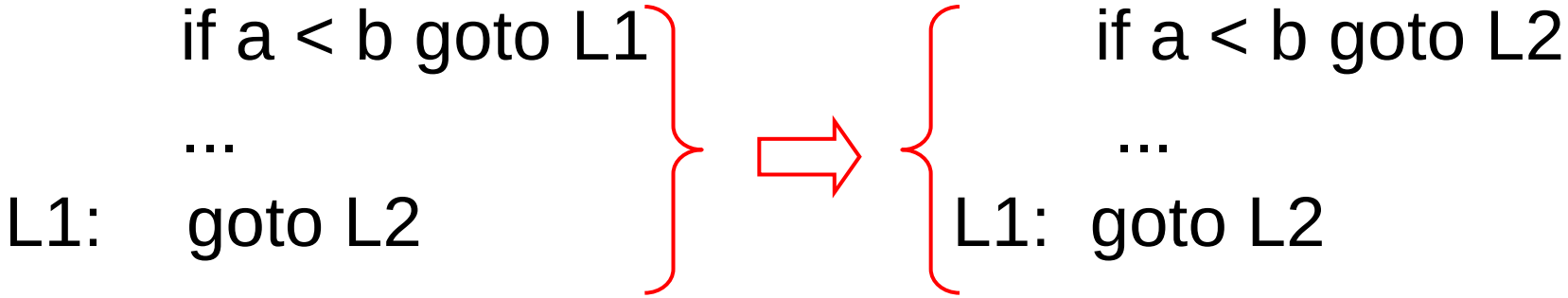
PO:

```
        if a < b   goto L3
L1:     if c >= d  goto L4
L2:     if e < f   goto L3
        goto L4
```

optymalizacja tego fragmentu uzależniona jest od położenia etykiet: L3 i L4

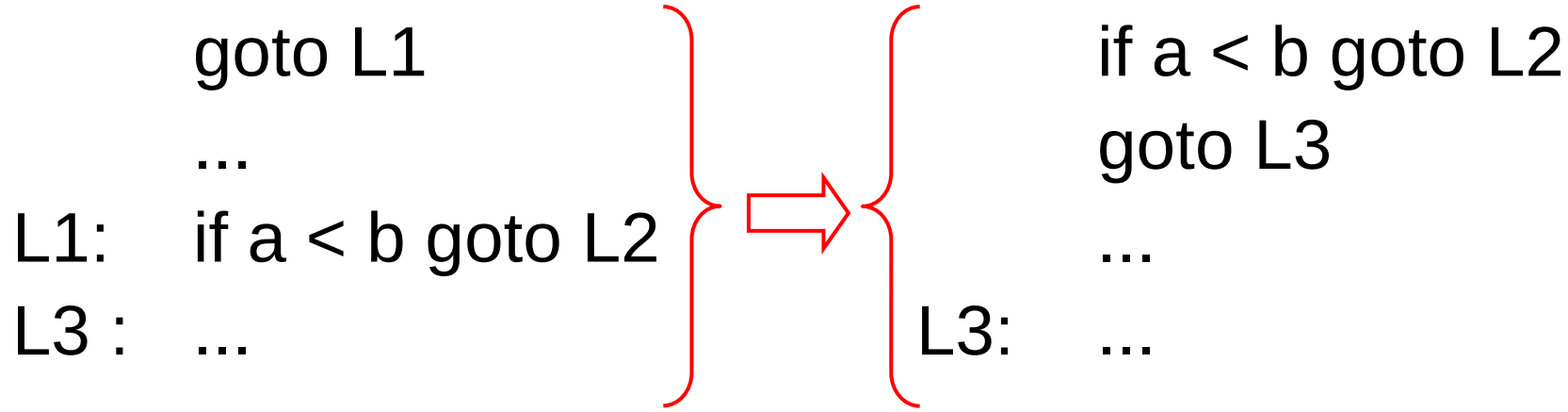
Optymalizacja przebiegu sterowania





Optymalizacja tego typu nie zmniejsza liczby instrukcji ale zmniejsza liczbę realizowanych skoków w czasie wykonania programu.

Reorganizacja kodu



Zakładamy, że do L1 jest tylko jeden skok (liczba skoków do każdej etykiety może być pamiętana w tablicy symboli).

Optymalizacja tego typu nie zmniejsza liczby instrukcji ale zmniejsza liczbę realizowanych skoków w czasie wykonania programu.

Eliminacja nieosiągalnego kodu

Przykład:

```
    debug := 0
    if debug = 1  goto L1
    goto L2                                   eliminacja ”skoku przez skok”
L1: ..... /* print debug information */
L2:
```

```
    if debug ≠ 1 goto L2                      uwzględnianie tego, że debug = 0
    ..... /* print debug information */
L2:
```

```
    if 0 ≠ 1 goto L2                          0 ≠ 1 → zawsze prawdziwe
    ...../* print debug information */        kod nieosiągalny można usunąć
L2:
```

```
goto L2
L2:
```

Eliminacja zbędnego skoku powoduje całkowite wyeliminowanie rozważanego fragmentu kodu

Eliminacja tożsamości algebraicznych

Przykład:

x := x + 0

x := x * 1

mogą być wyeliminowane gdyż faktycznie nie zmieniają wartości zmiennej x

Redukcja mocy kodu

Przykład:

y := 2 * x

może być zastąpione przez

y := x + x

gdyż mnożenie trwa dłużej niż dodawanie