



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).

- 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.

- 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.



AGH

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) $t_1 := s * n$
 (4) $v := a[t_1]$

B1

(5) $i := i + 1$
 (6) $t_2 := s * i$
 (7) $t_3 := a[t_2]$
 (8) if $t_3 < v$ goto(5)

B2

(9) $j := j - 1$
 (10) $t_4 := s * j$
 (11) $t_5 := a[t_4]$
 (12) if $t_5 > v$ goto(9)

B3

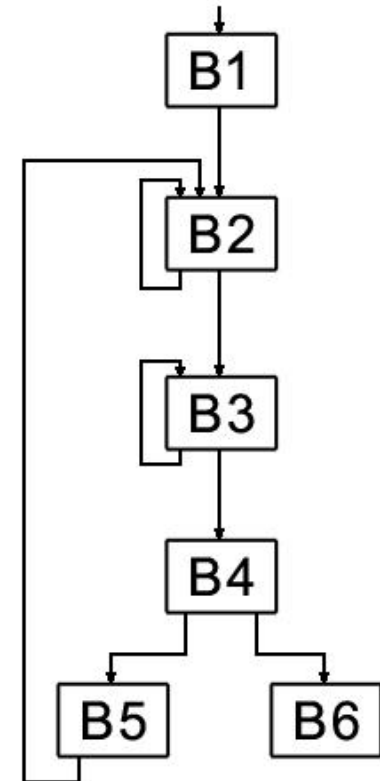
(13) if $i \geq j$ goto(23) **B4**

(14) $t_6 := s * i$
 (15) $x := a[t_6]$
 (16) $t_7 := s * i$
 (17) $t_8 := s * j$
 (18) $t_9 := a[t_8]$
 (19) $a[t_7] := t_9$
 (20) $t_{10} := s * j$
 (21) $a[t_{10}] := x$
 (22) goto(5)

B5

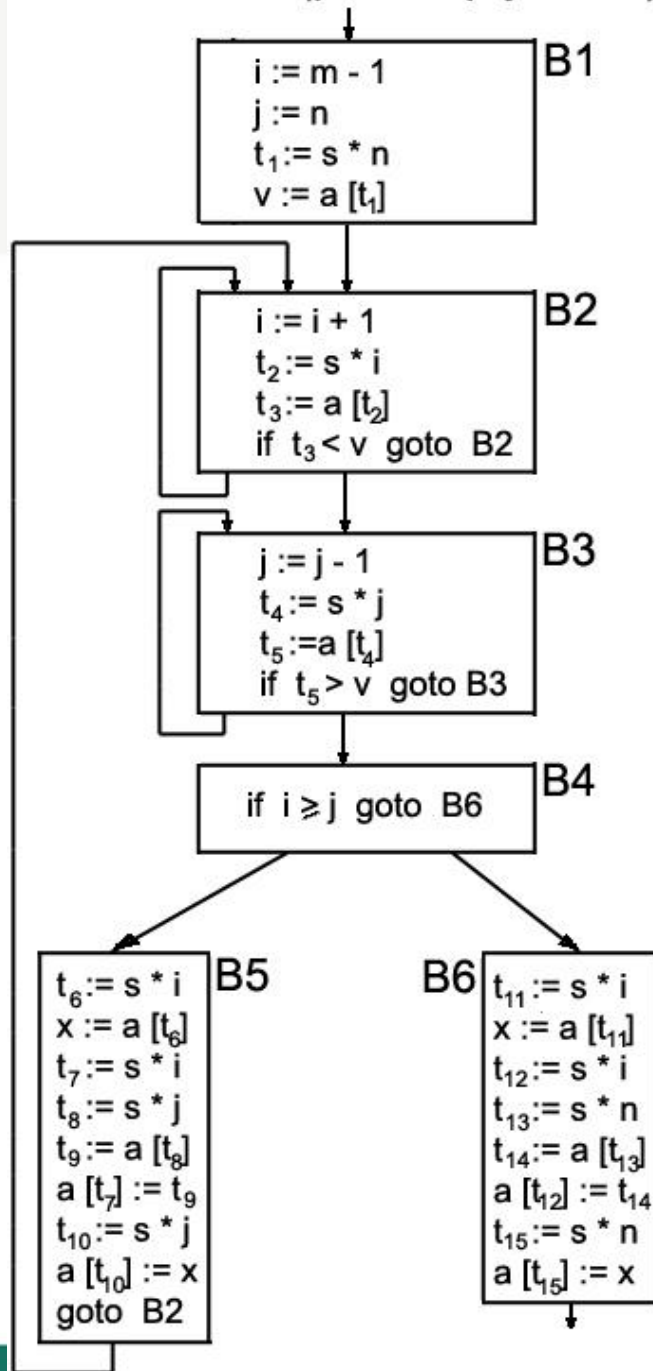
(23) $t_{11} := s * i$
 (24) $x := a[t_{11}]$
 (25) $t_{12} := s * i$
 (26) $t_{13} := s * n$
 (27) $t_{14} := a[t_{13}]$
 (28) $a[t_{12}] := t_{14}$
 (29) $t_{15} := s * n$
 (30) $a[t_{15}] := x$

B6



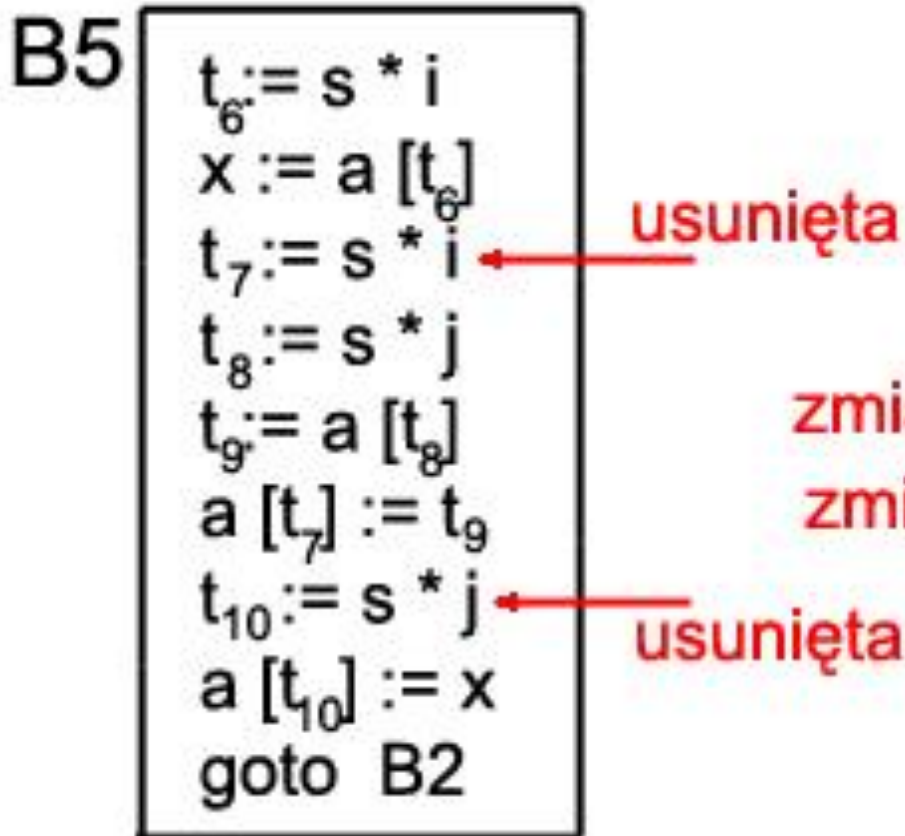
Graf obliczeń

Graf obliczeń (przed optymaliz.)

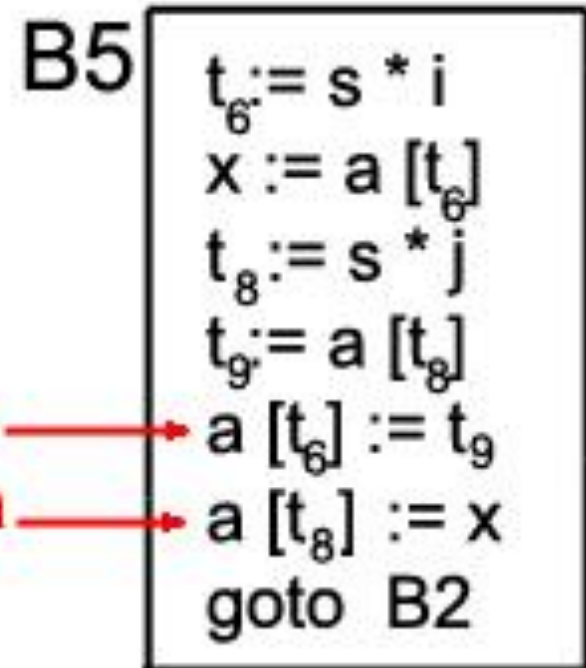


Lokalna eliminacja wspólnych podwyrażeń

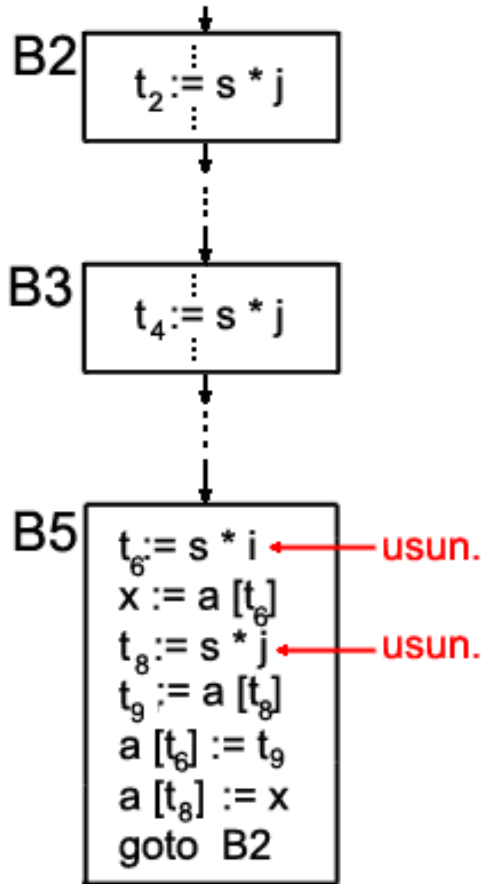
PRZED:



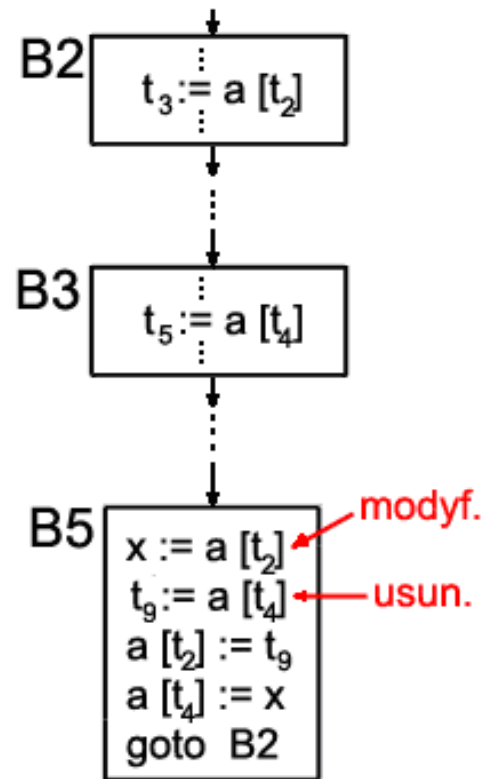
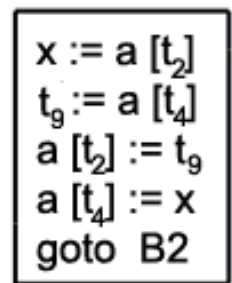
PO:



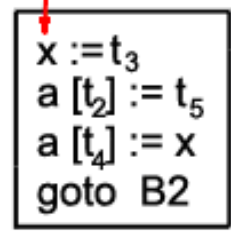
Globalna eliminacja wspólnych podwyrażeń



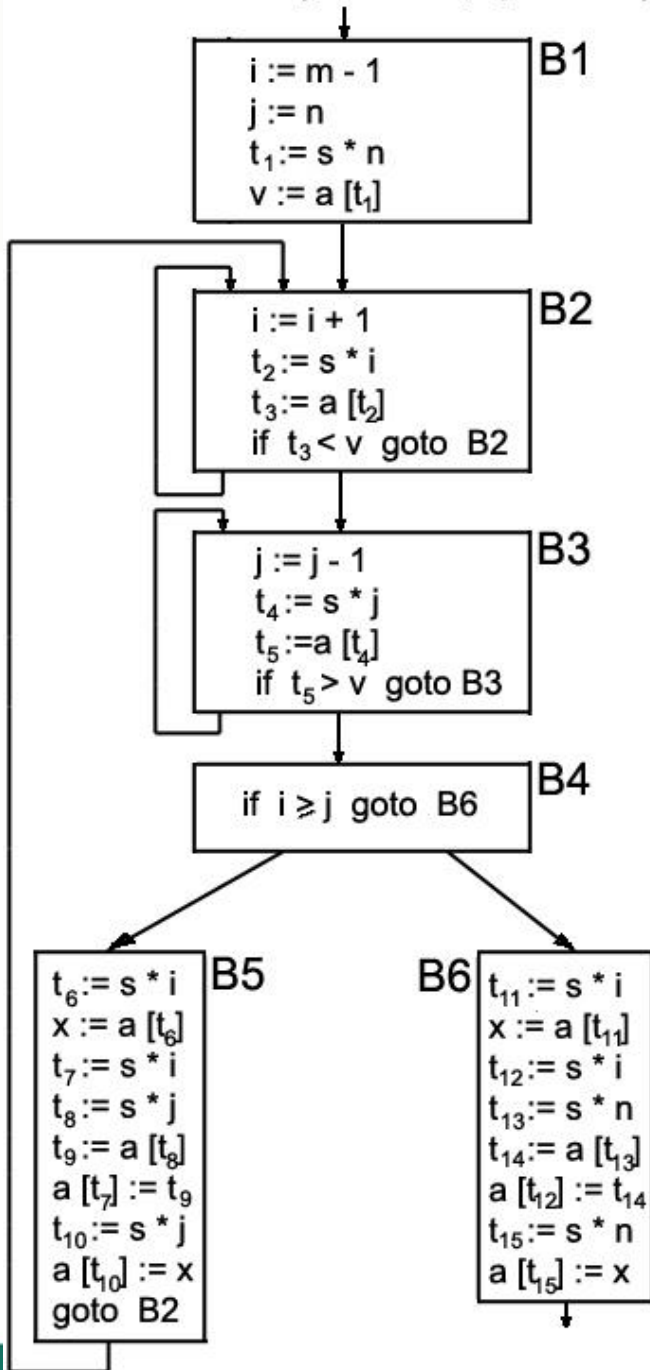
PO:



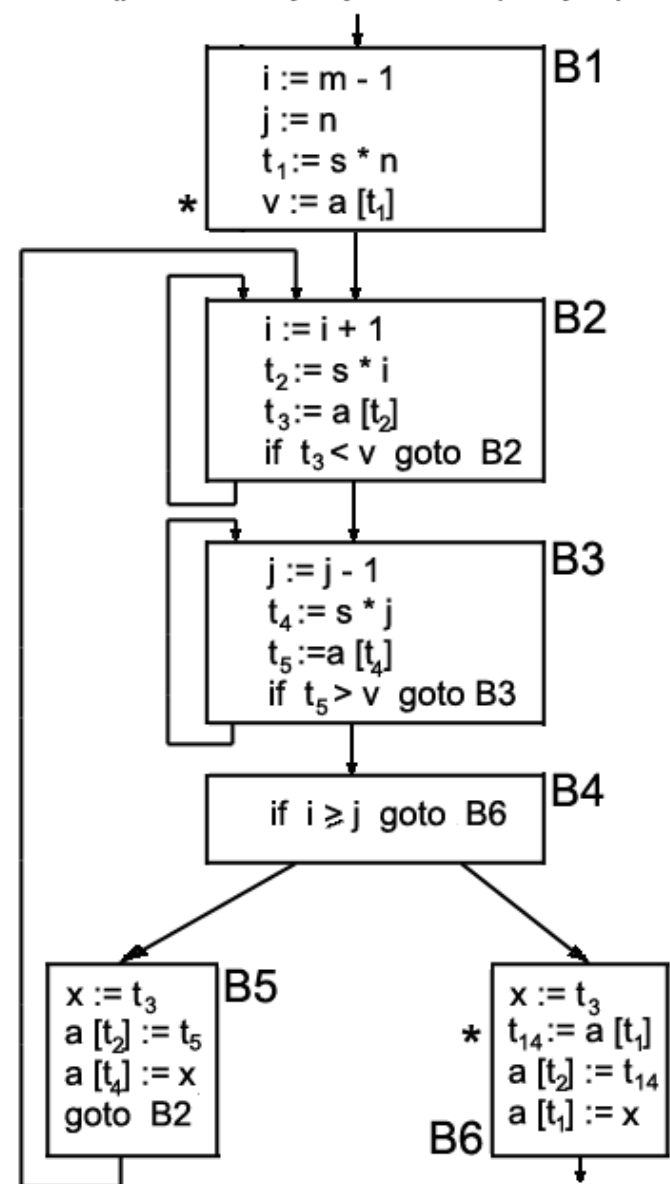
x nie jest
zmienną
tymczasową



Graf obliczeń (przed optymaliz.)



(po eliminacji wyrażeń wspólnych)



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.

PRZED:

B5: `x := t3`
`a[t2] := t5`
`a[t4] := t3`
`goto B2`

wyeliminowane

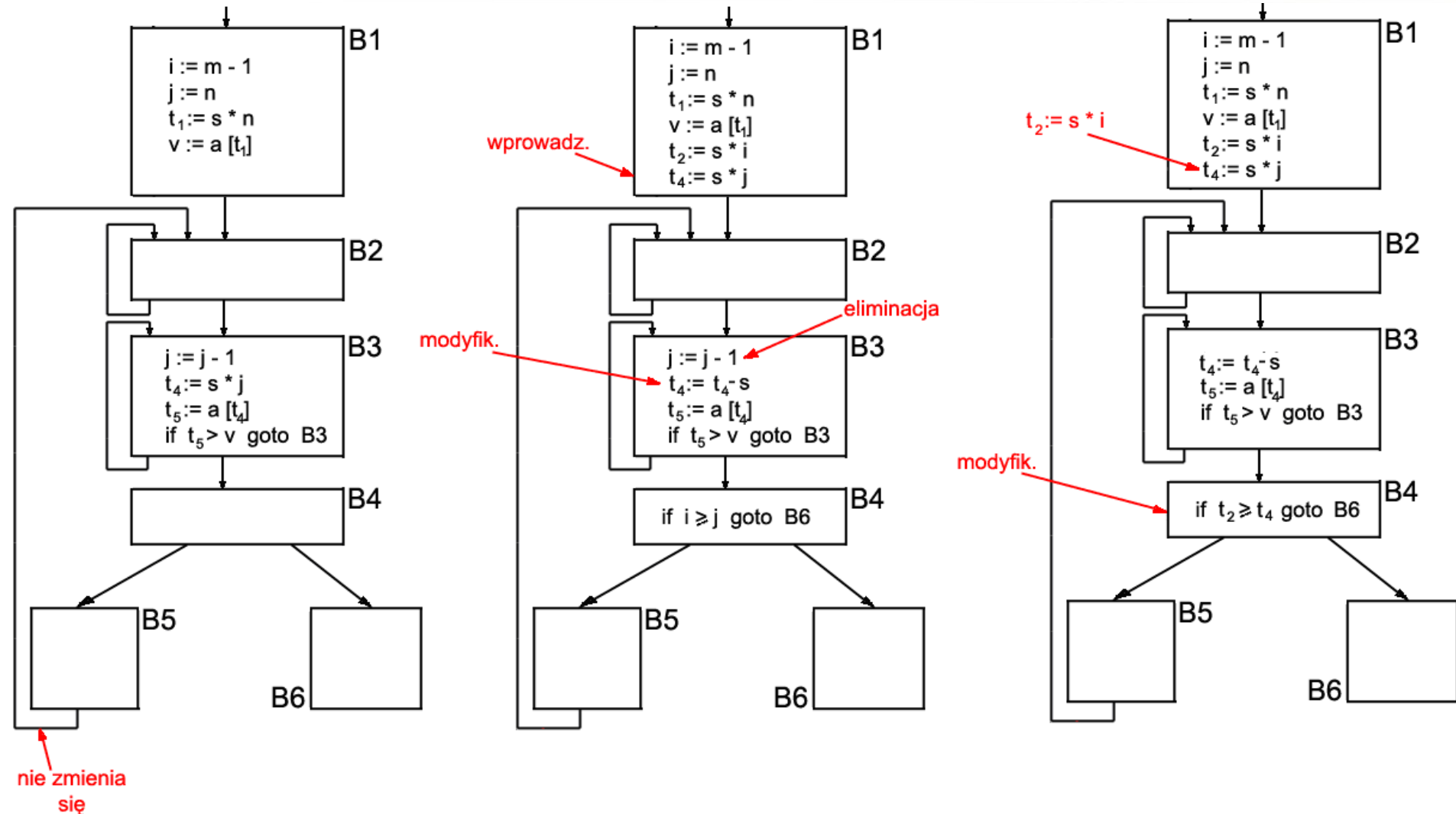
PO:

B5: `a[t2] := t5`
`a[t4] := t3`
`goto B2`

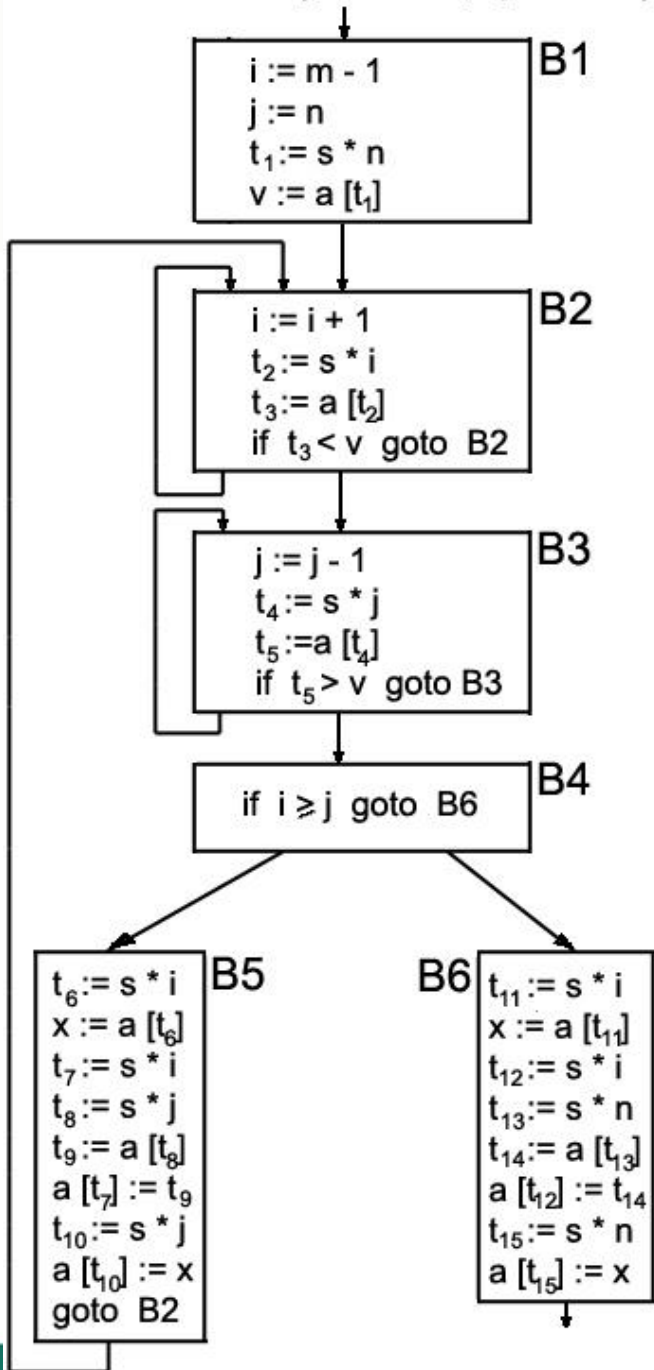
Zmienne indukcyjne w pętłach

- 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.

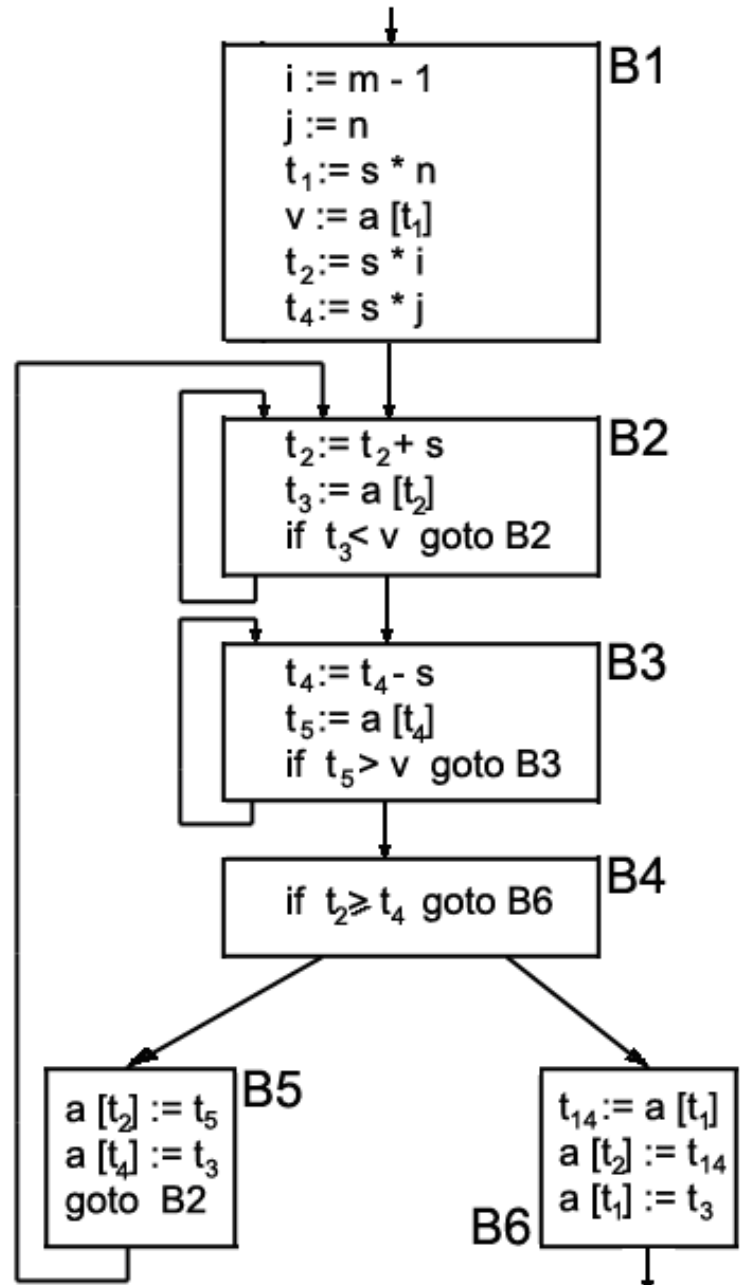
Zmienne indukcyjne - przykład



Graf obliczeń (przed optymaliz.)



Ostatecznie:



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" */  
}
```




AGH

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
```

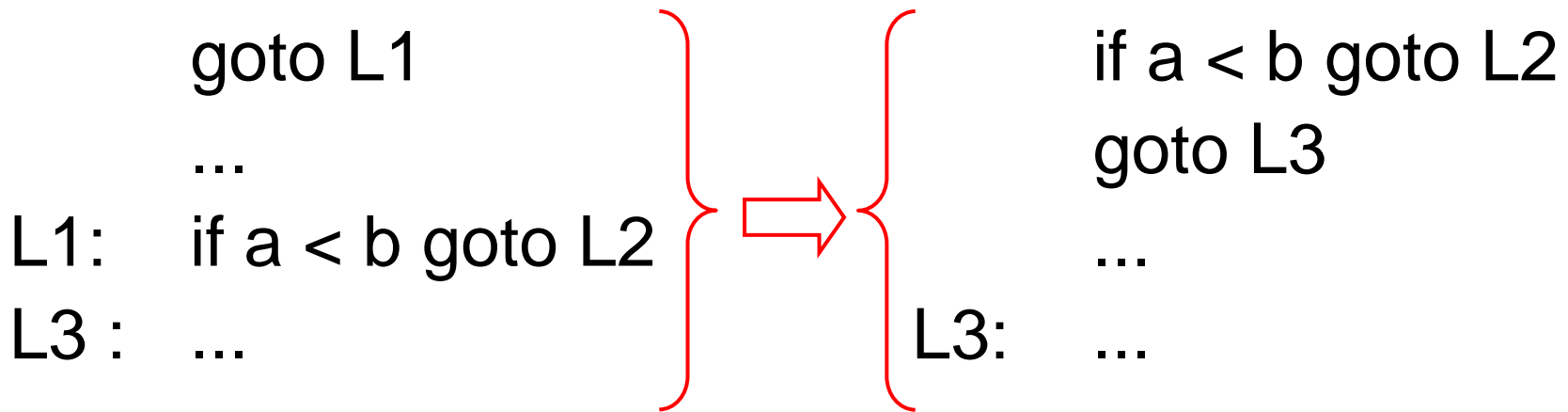
można wyeliminować

PO:

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

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

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
```

```
L1: ..... /* print debug information */
L2:
```

eliminacja "skoku przez skok"

```
if debug ≠ 1 goto L2
..... /* print debug information */
```

```
L2:
```

uwzględnianie tego, że debug = 0

```
if 0 ≠ 1 goto L2
...../* print debug information */
```

```
L2:
```

0 ≠ 1 → zawsze prawdziwe
kod nieosiągalny można usunąć

```
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