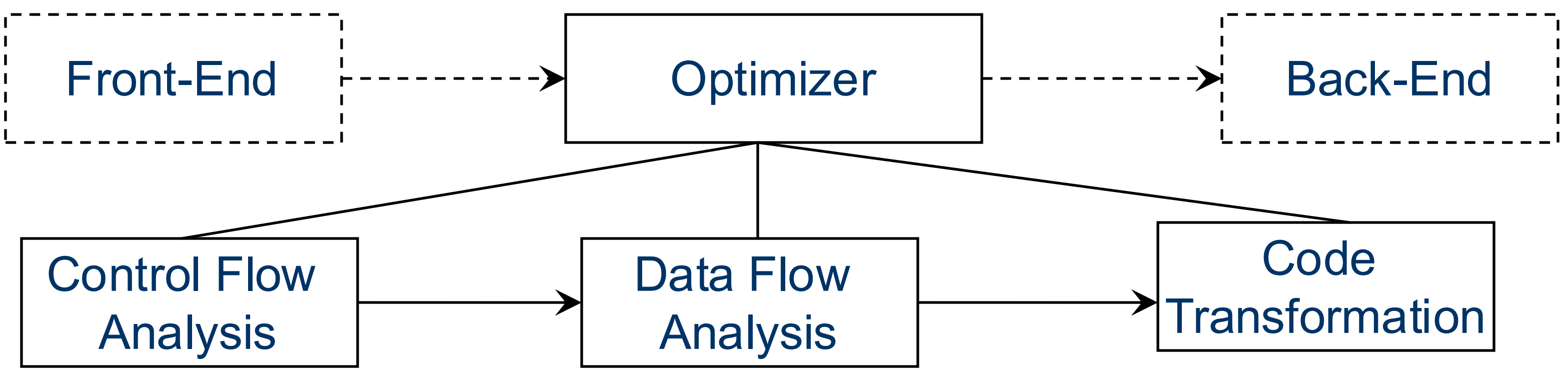
Chapter 10: Optimization

Zhen Gao
gaozhen@tongji.edu.cn

Optimization

Perform various **equivalent** transformations on a program so that the transformed program can generate **more efficient** target code.



Outline

- Overview
- Local Optimization
- Loop Optimization

Overview

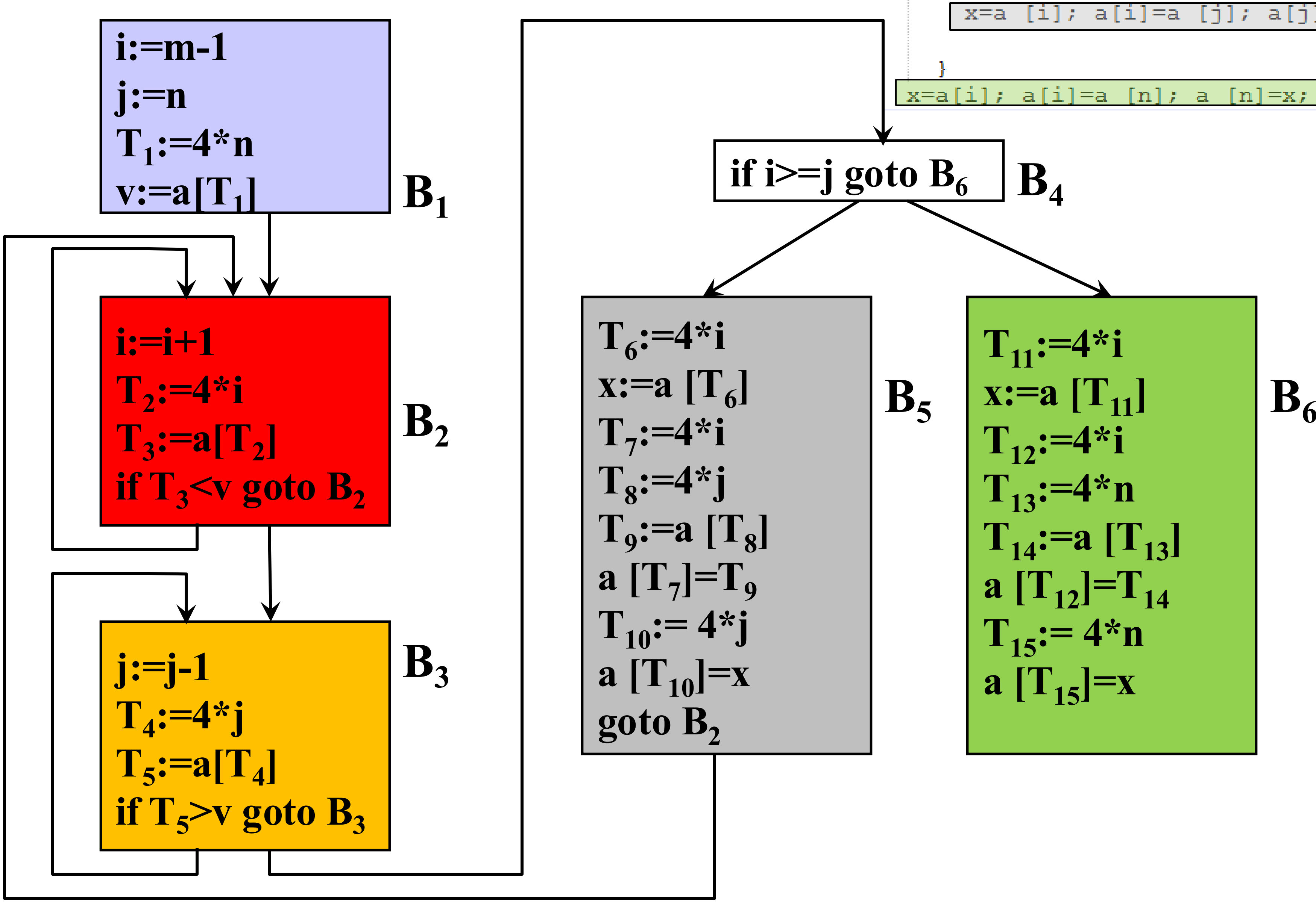
- **Principles for Code Transformations in an Optimizing Compiler:**
  - **Equivalence Principle:** Optimizations must not change the program's output
  - **Effectiveness Principle:** Optimized code should have shorter runtime and use less memory
  - **Profitability Principle:** Achieve good optimization results at minimal cost

Overview

- Three Levels of Optimization:
  - Local Optimization
  - Loop Optimization
  - Global Optimization
- Types of Optimization:
  - Eliminate Redundant Computations (or Common Subexpression Elimination)
  - Copy Propagation
  - Dead-Code Elimination
  - Code Motion
  - Strength Reduction
  - Transform Loop Control Conditions
  - Constant Folding

```
void quicksort (m, n);
int m, n;
   {
         int i, j;
         int v, x;
         if (n<=m) return;
         /* fragment begins here*/
         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;
        /*fragment ends here*/
       quicksort (m, j); quicksort (i+1, n);
}
```

```
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;
```



Examples of Optimization Techniques

- Types of Optimization :
  - **Eliminate Redundant Computations (or Common Subexpression Elimination)**
  - Copy Propagation
  - Dead-Code Elimination
  - Strength Reduction
  - Eliminate Induction Variables

Common Subexpression Elimination

```
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;
```

B_5:

T_6:=4*i

x:=a[T_6]

T_7:=4*i

T_8:=4*j

T_9:=a[T_8]

a[T_7]=T_9

T_{10}:= 4*j

a[T_{10}]=x

goto B_2

B_5:

T_6:=4*i

x:=a[T_6]

T_7:= T_6

T_8:=4*j

T_9:=a[T_8]

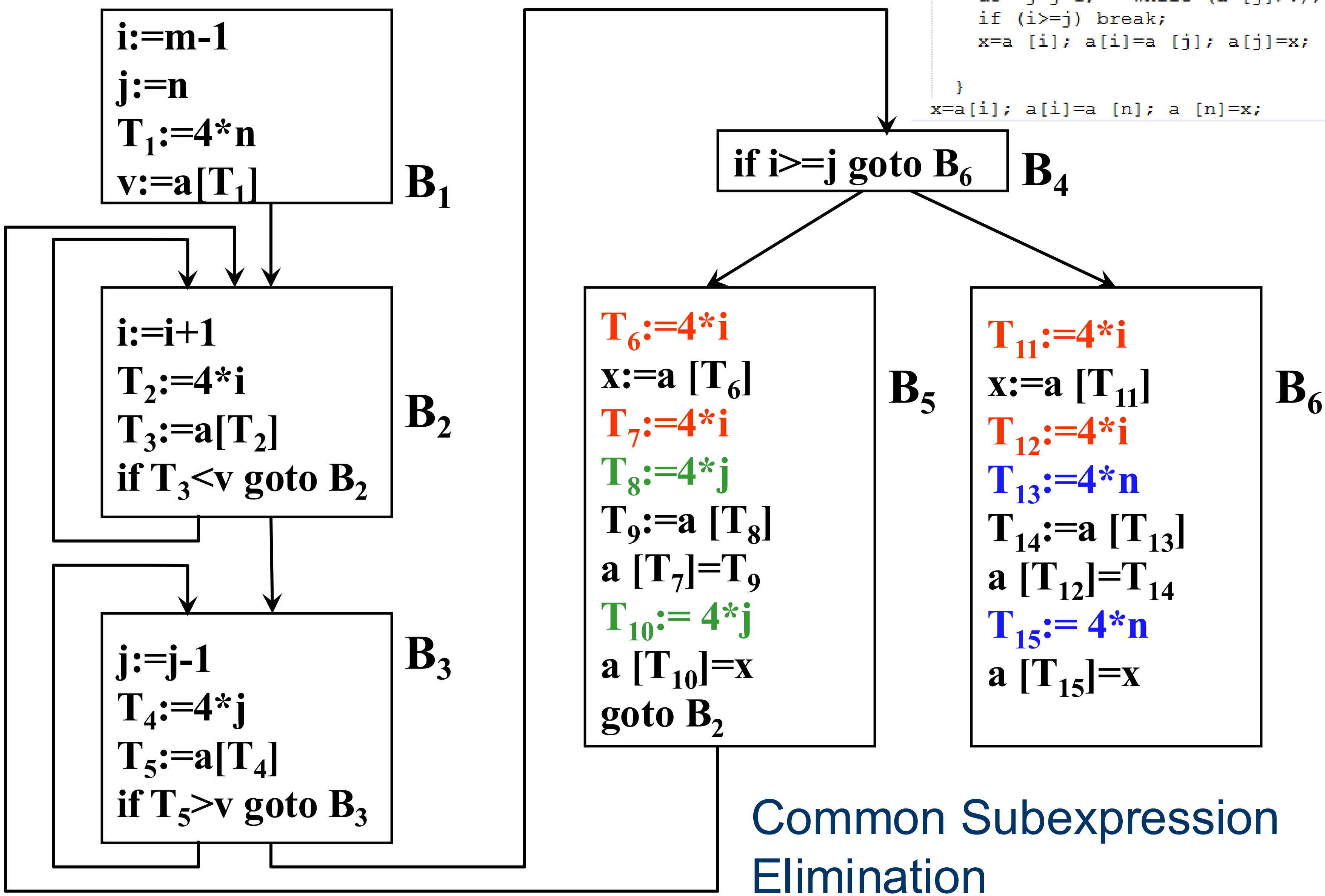
a[T_7]=T_9

T_{10}:= T_8

a[T_{10}]=x

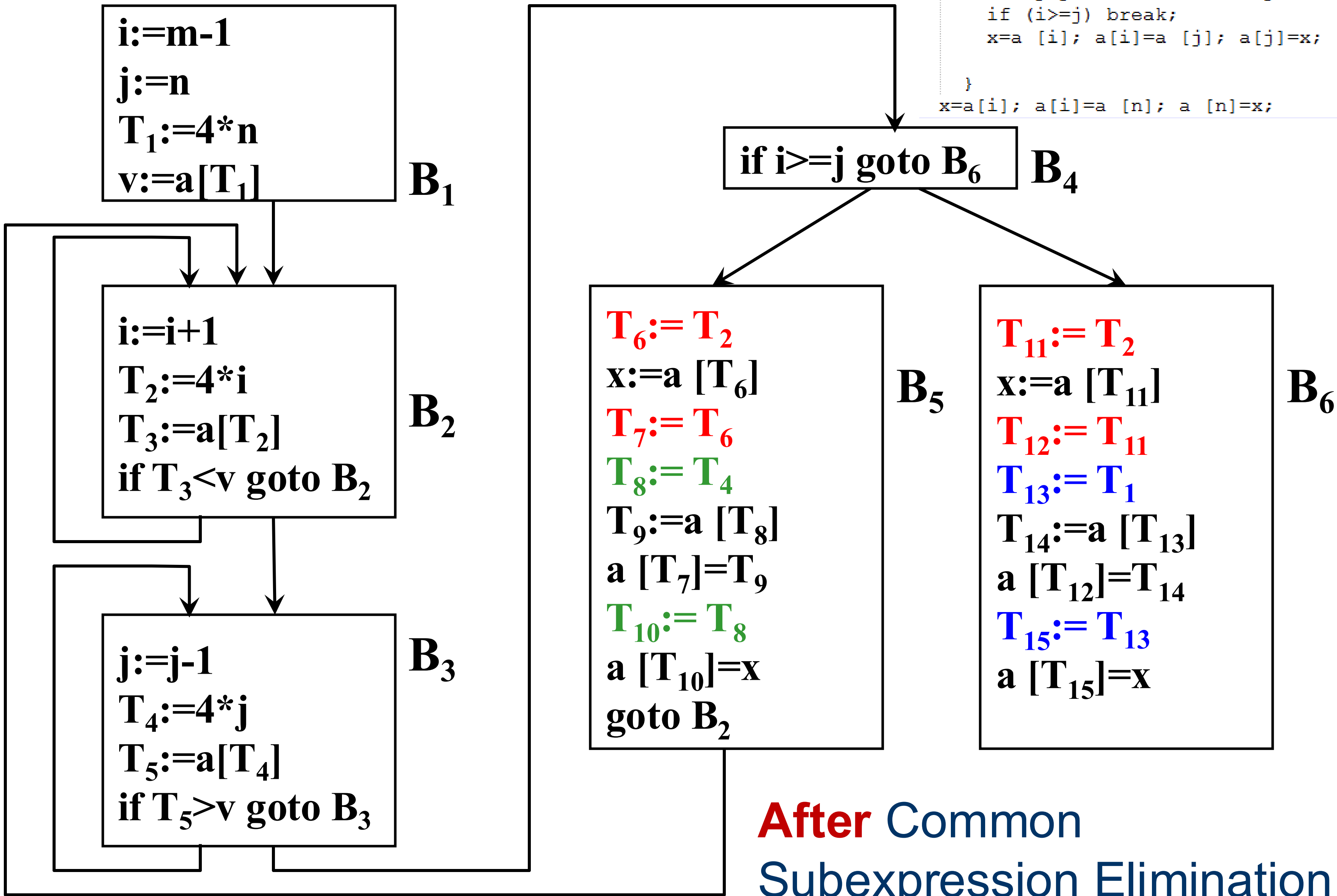
goto B_2

```
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;
```



Common Subexpression Elimination

```
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;
```



After Common Subexpression Elimination

Examples of Optimization Techniques

- Types of Optimization:
  - Eliminate Redundant Computations (or Common Subexpression Elimination)
  - **Copy Propagation**
  - Dead-Code Elimination
  - Strength Reduction
  - Eliminate Induction Variables

```
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;
```

Copy Propagation

B₅:

$T_6 := T_2$

$x := a[T_6]$

$T_7 := T_6$

$T_8 := T_4$

$T_9 := a[T_8]$

$a[T_7] = T_9$

$T_{10} := T_8$

$a[T_{10}] = x$

goto B_2

$T_6 := T_2$

$x := a[T_6]$

T6's value has not changed.

B₅:

$T_6 := T_2$

$x := a[T_2]$

$T_7 := T_2$

$T_8 := T_4$

$T_9 := a[T_4]$

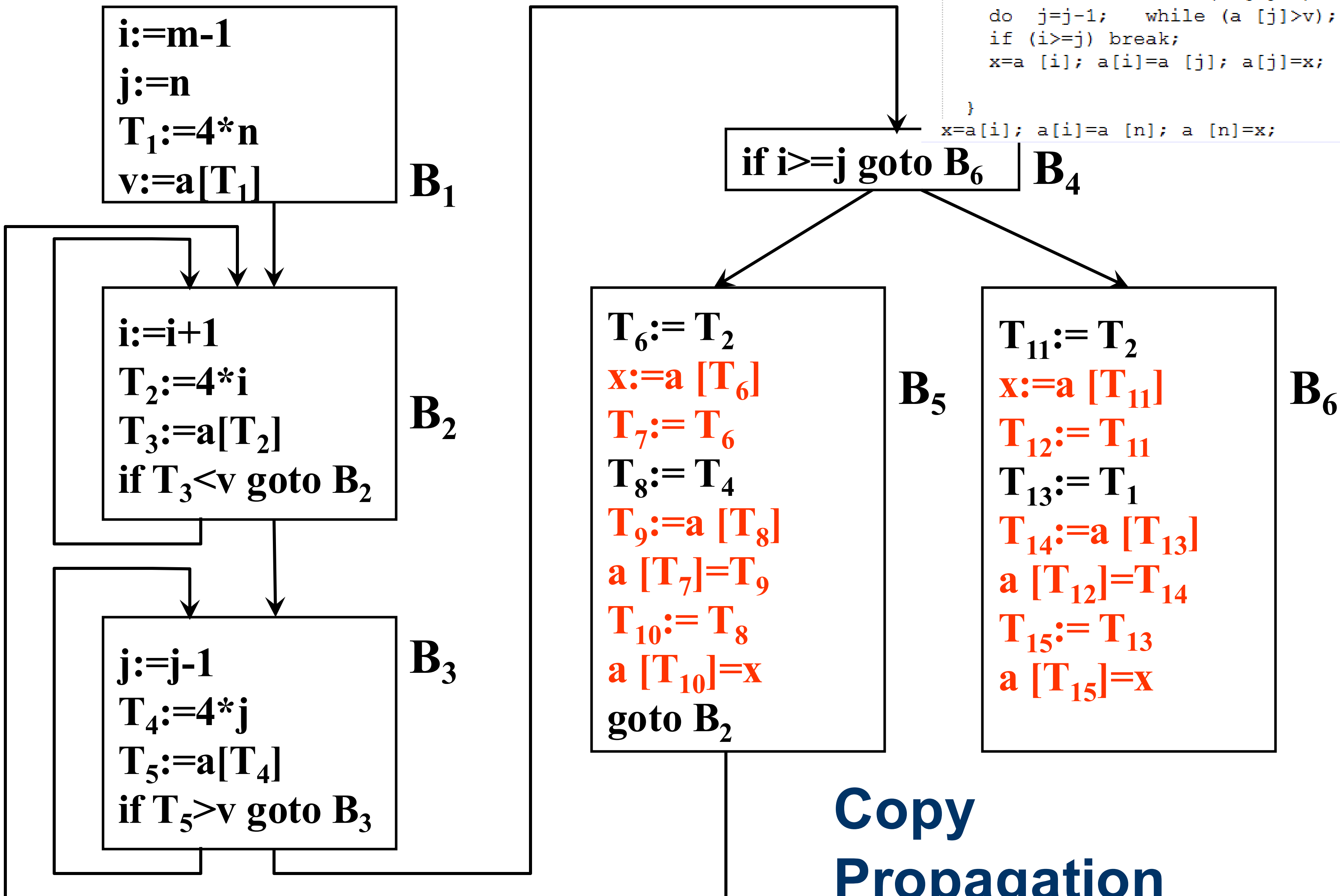
$a[T_2] = T_9$

$T_{10} := T_4$

$a[T_4] = x$

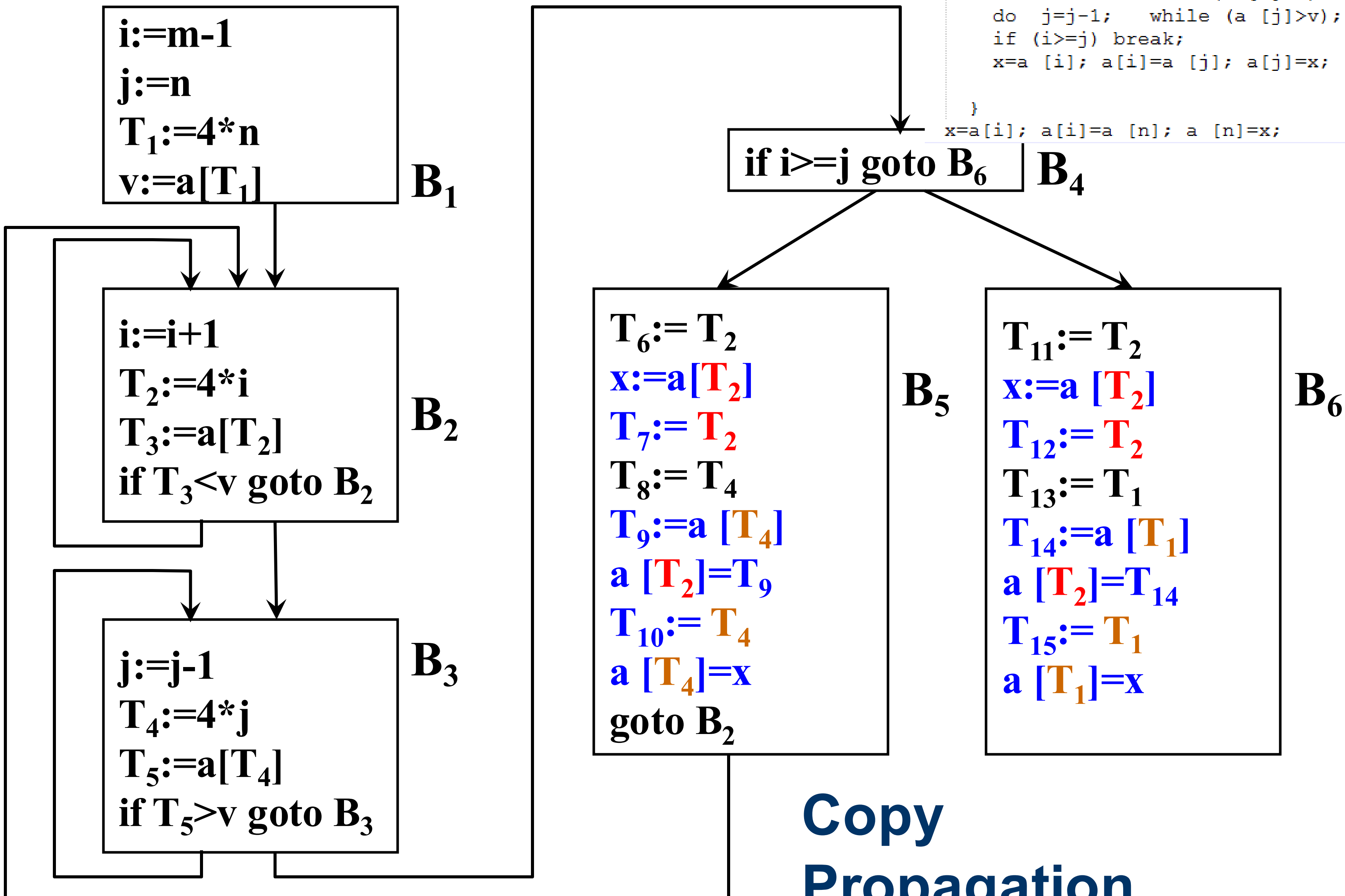
goto B_2

```
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;
```



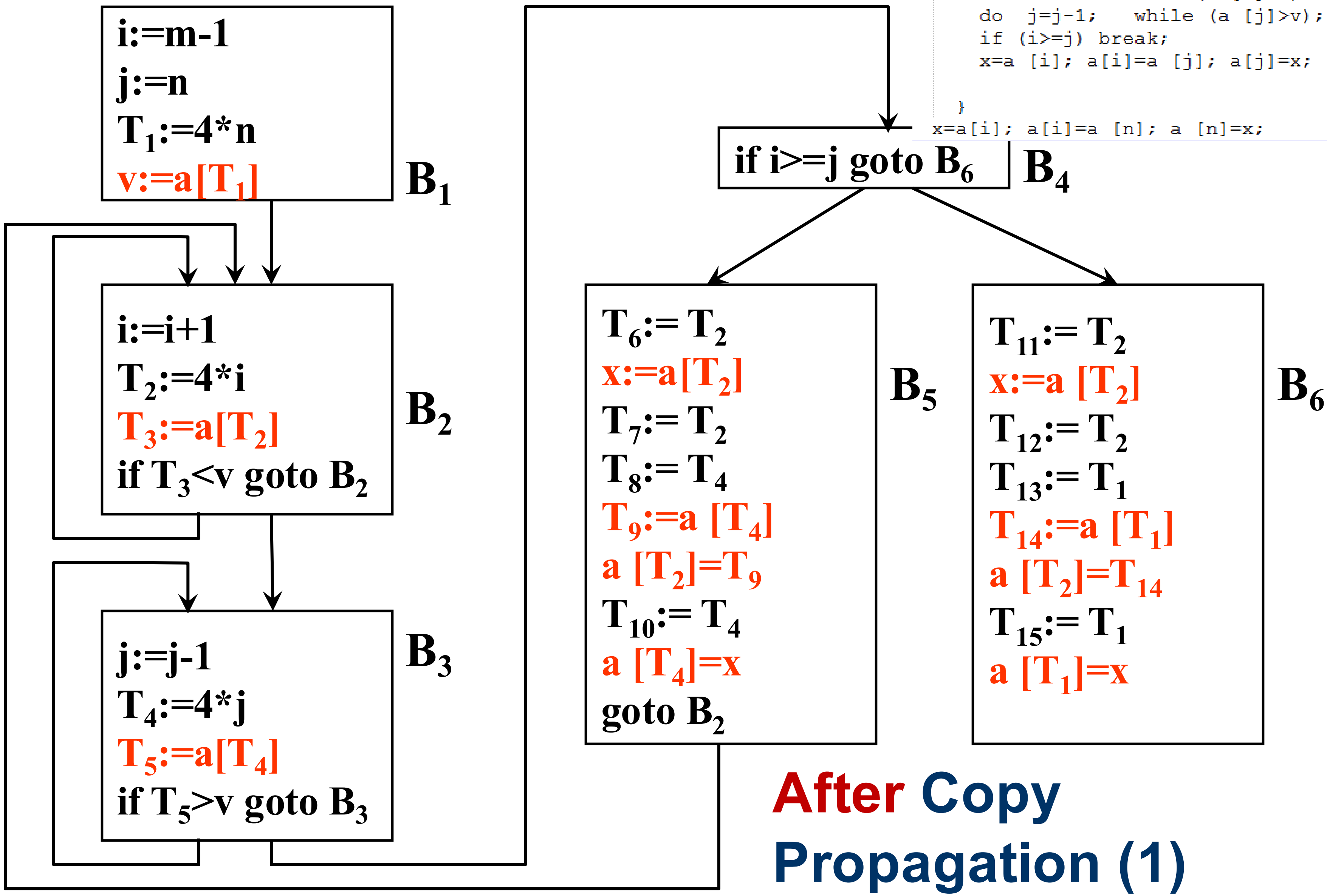
Copy Propagation

```
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;
```



Copy Propagation

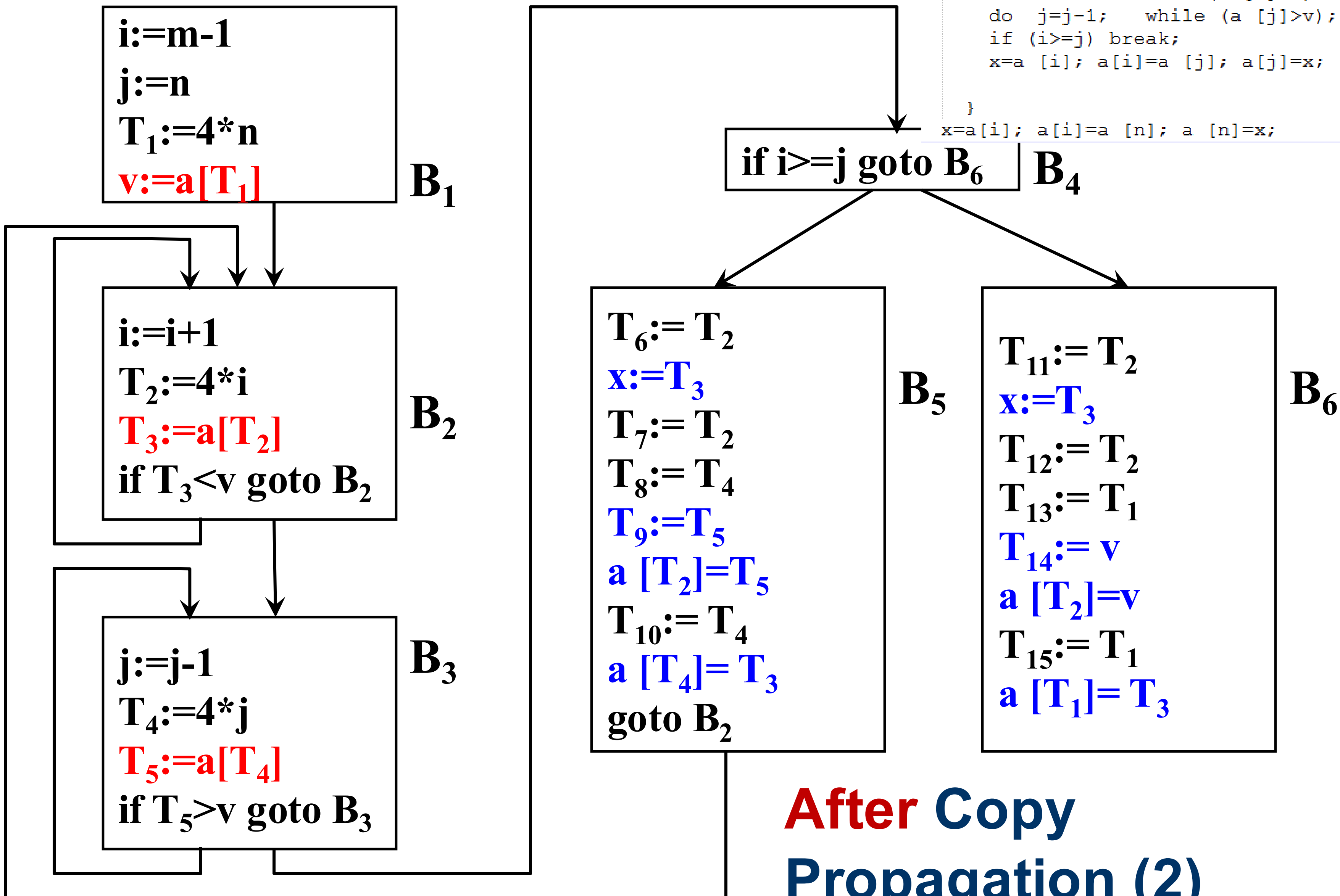
```
i=m-1; j=n; v=a [n];
while (1) {
    do  i=i+1;   while (a [i]<v);
    do  j=j-1;   while (a [j]>v);
    if (i>=j) break;
    x=a [i]; a[i]=a [j]; a[j]=x;

  }
x=a[i]; a[i]=a [n]; a [n]=x;
```



After Copy Propagation (1)

Observe $a[T_1]$ $a[T_2]$ $a[T_4]$

```
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;
```



After Copy Propagation (2)

Examples of Optimization Techniques

- Types of Optimization :
  - Eliminate Redundant Computations (or Common Subexpression Elimination)
  - Copy Propagation
  - **Dead-Code Elimination**
  - Strength Reduction
  - Eliminate Induction Variables

```
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;
```

Dead-Code Elimination

B5

```
T6:=4*i
x:=a [T6]
T7:=4*i
T8:=4*j
T9:=a [T8]
a [T7]=T9
T10:= 4*j
a [T10]=x
goto B2
```

B5:

$T_6 := T_2$

$x := T_3$

$T_7 := T_2$

$T_8 := T_4$

$T_9 := T_5$

$a\ [T_2] = T_5$

$T_{10} := T_4$

$a\ [T_4] = T_3$

goto B_2

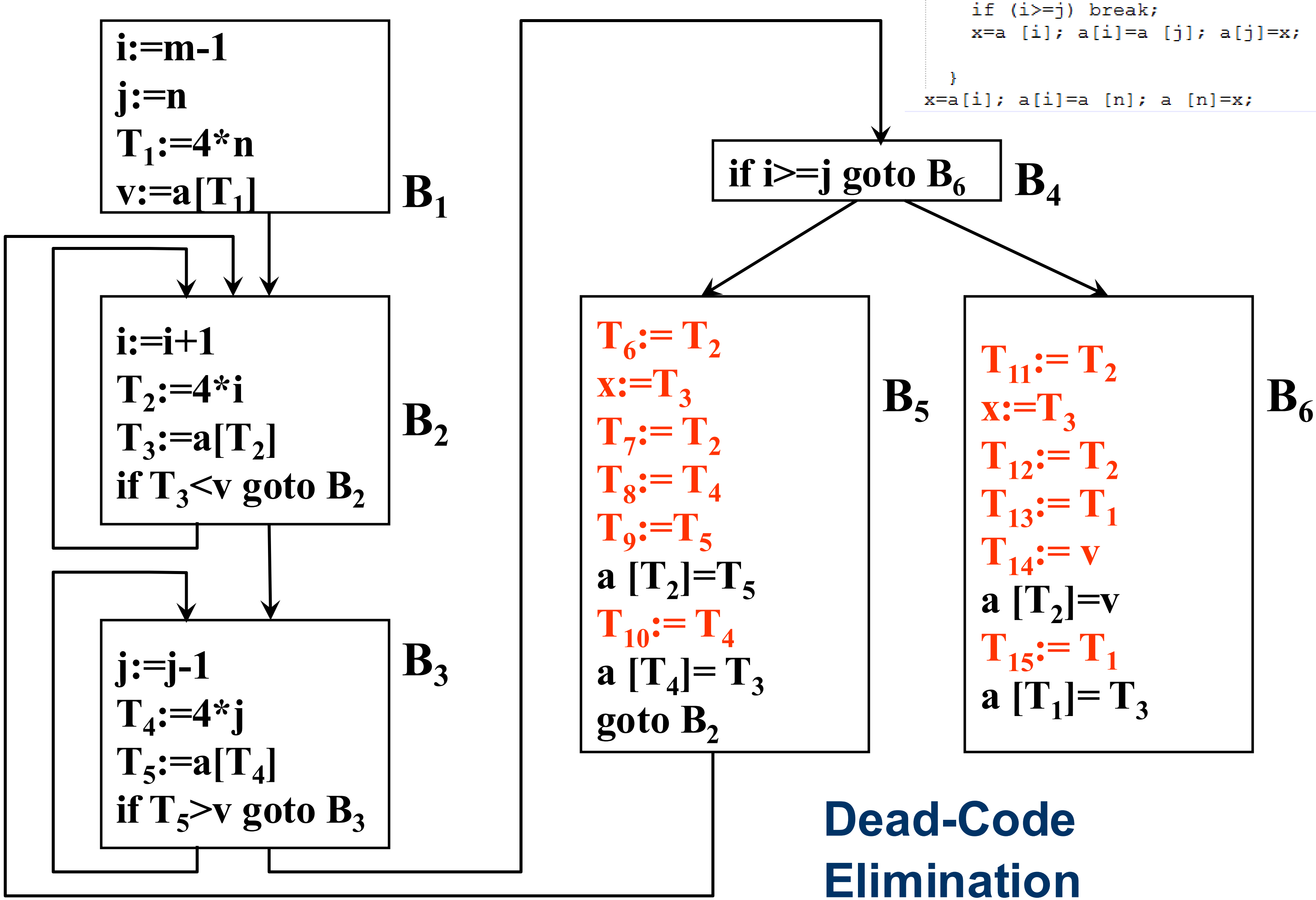
B5:

$a\ [T_2] = T_5$

$a\ [T_4] = T_3$

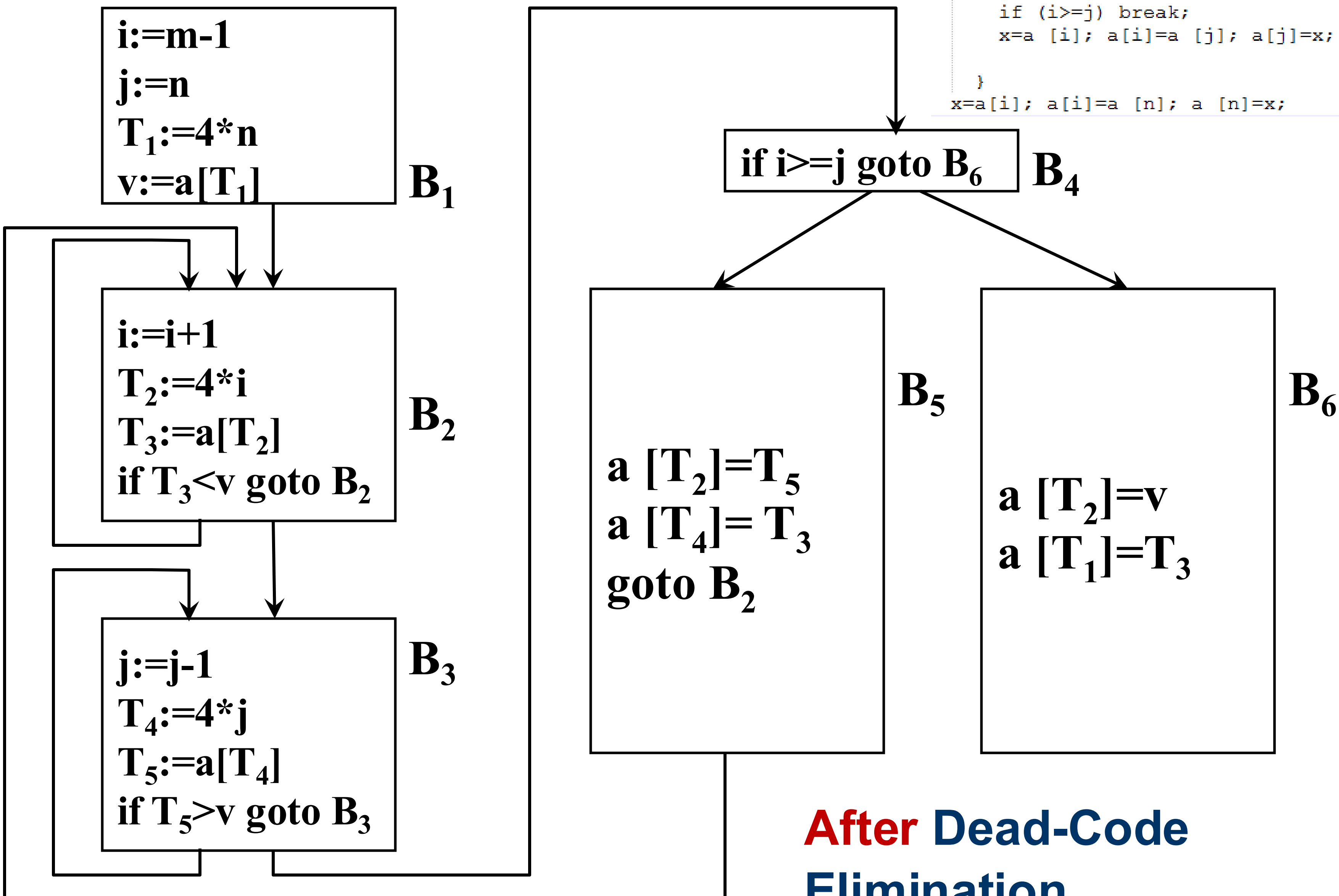
goto B_2

```
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;
```



Dead-Code Elimination

```
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;
```



After Dead-Code Elimination

Examples of Optimization Techniques

- Types of Optimization :
  - Eliminate Redundant Computations (or Common Subexpression Elimination)
  - Copy Propagation
  - Dead-Code Elimination
  - **Strength Reduction**
  - Eliminate Induction Variables

do j=j-1; while (a [j]>v);

Strength Reduction

B_3

$j := j-1$

$T_4 := 4 * j$

$T_5 := a[T_4]$

if $T_5 > v$ goto B_3

...

$T_4 := 4 * j$

...

B_3

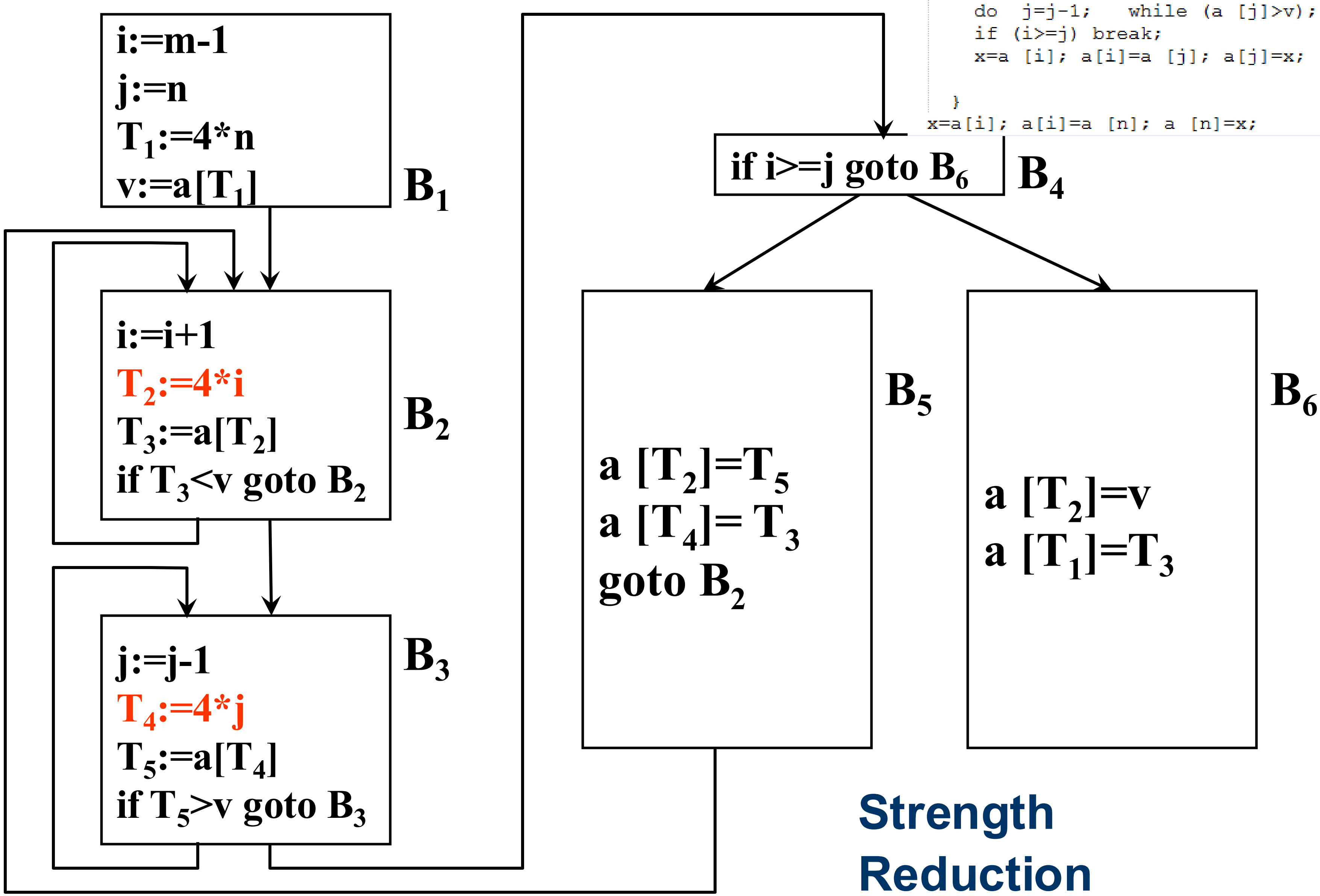
$j := j-1$

$T_4 := T_4 - 4$

$T_5 := a[T_4]$

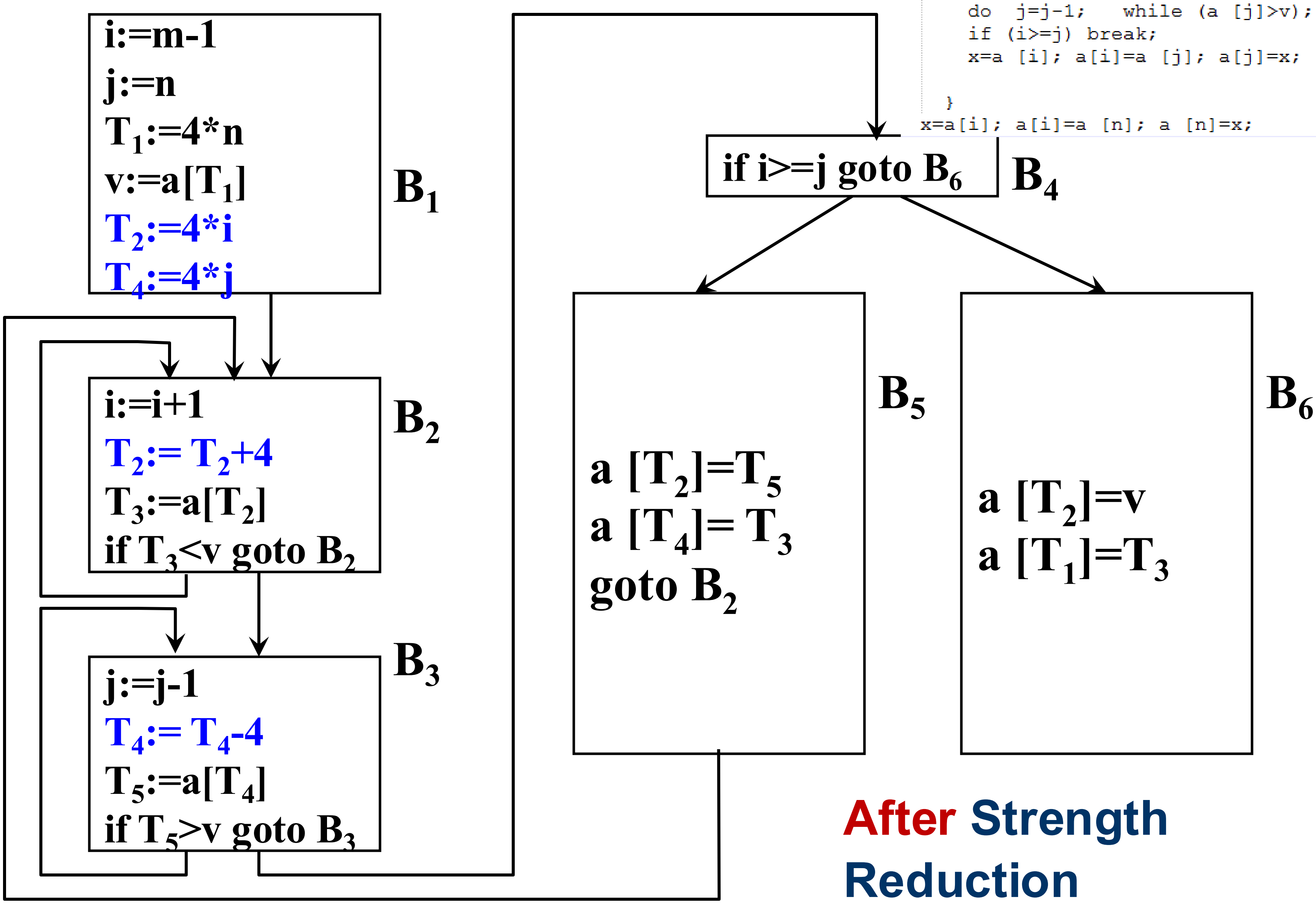
if $T_5 > v$ goto B_3

```
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;
```



Strength Reduction

```
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;
```



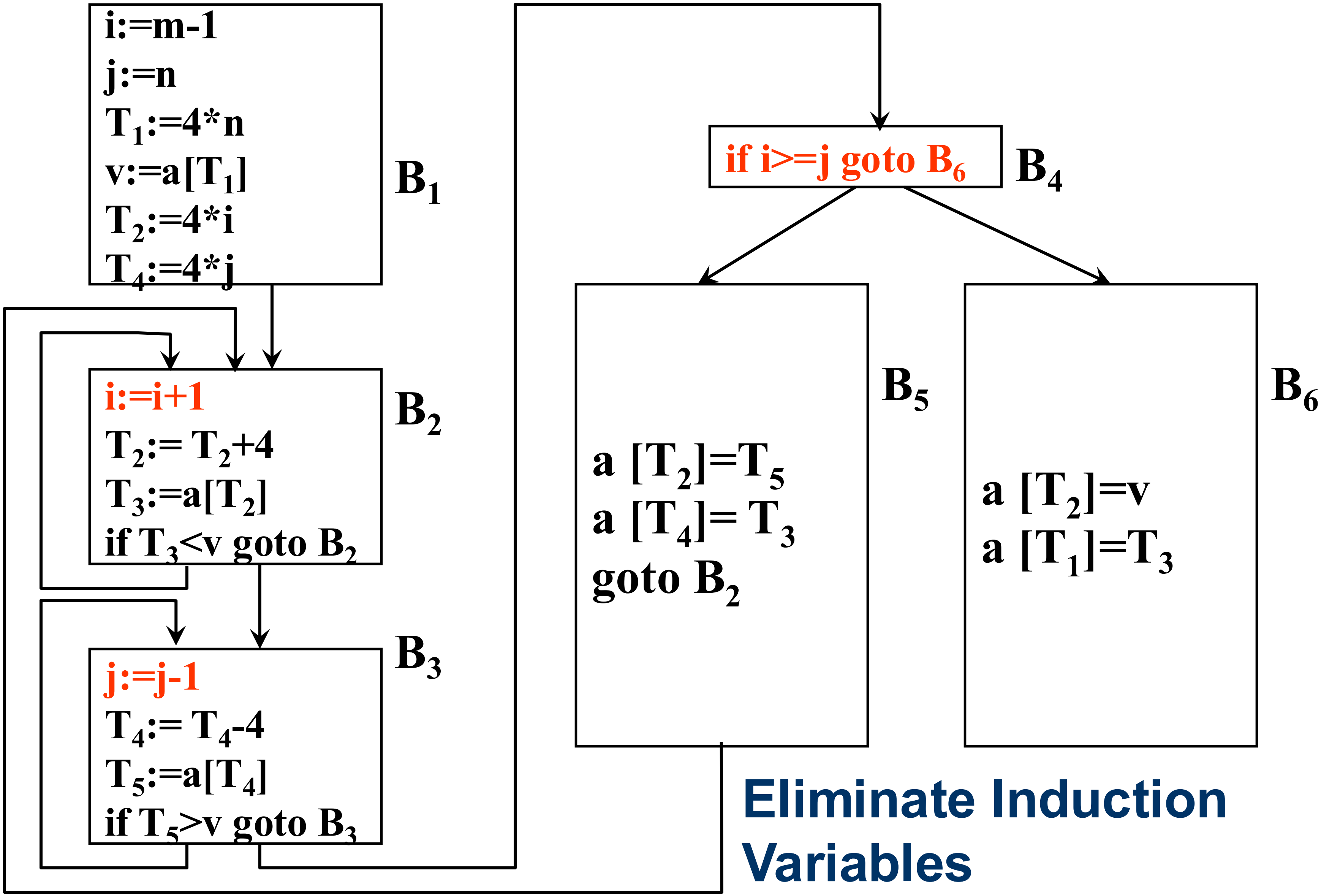
After Strength Reduction

Examples of Optimization Techniques

- Types of Optimization :
  - Eliminate Redundant Computations (or Common Subexpression Elimination)
  - Copy Propagation
  - Dead-Code Elimination
  - Strength Reduction
  - **Eliminate Induction Variables**

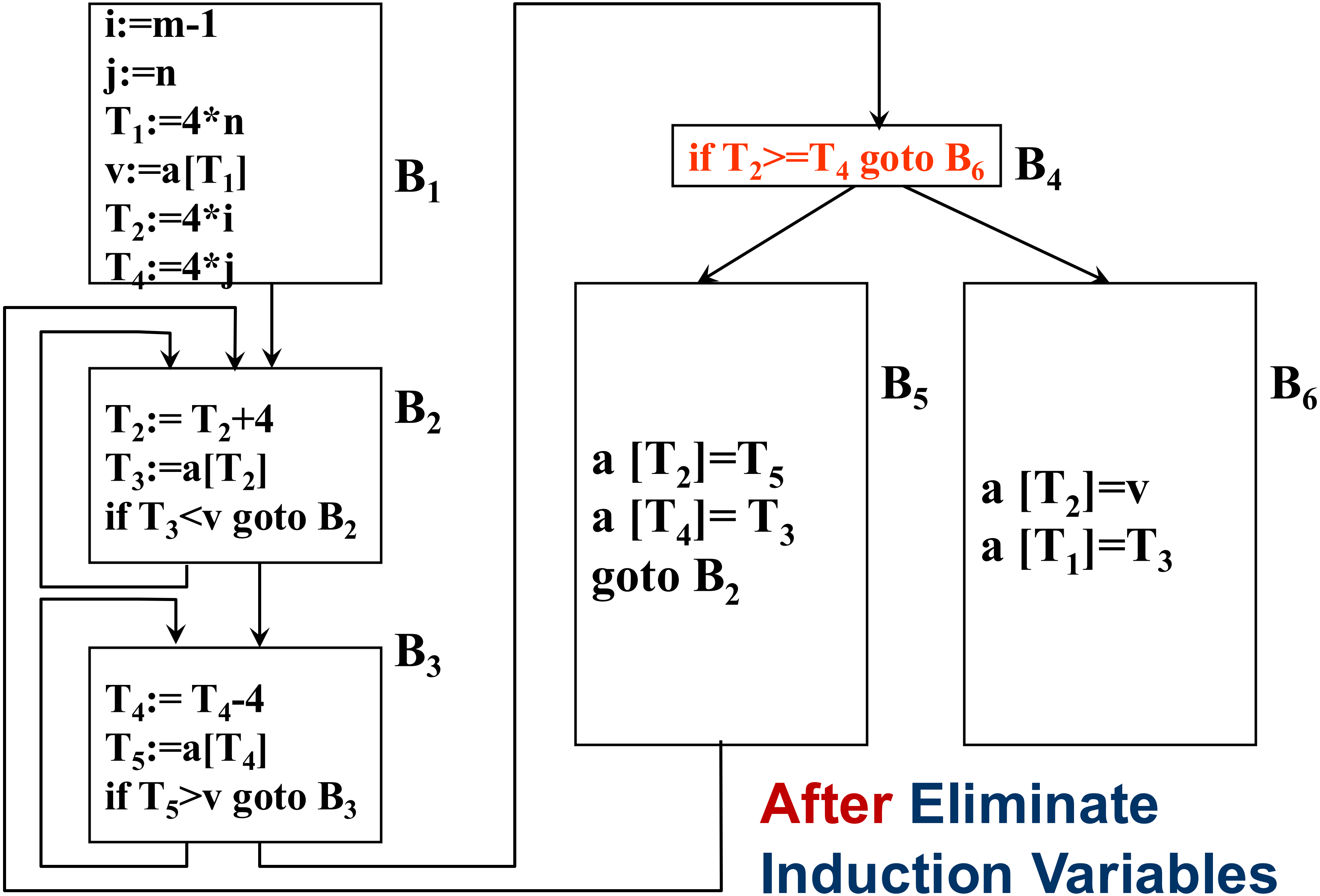
Eliminate Induction Variables

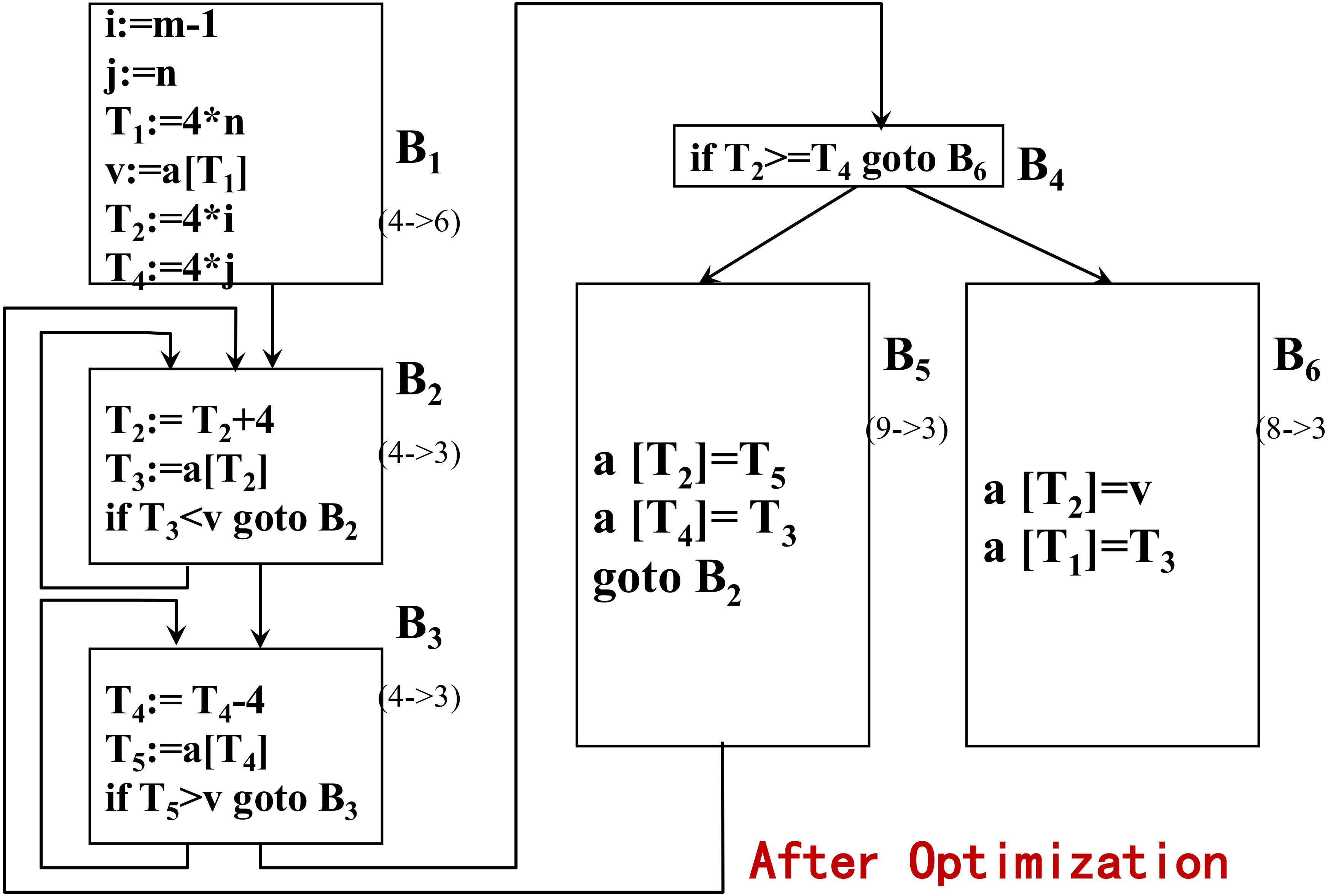
- The value of i in B2 maintains a linear relationship with T2.
- The value of j in B3 maintains a linear relationship with T4.
- Such variables are called induction variables.
- This type of variable can also be optimized.



Eliminate Induction Variables

After Eliminate Induction Variables





After Optimization

Outline

- Overview
- Local Optimization
- Loop Optimization

Questions

- What is a basic block?
- How to divide a basic block?
- What optimizations can be performed within a basic block?
- How to perform local optimization?

Basic Block

- **Basic block**: A sequence of statements with a single entry (first statement) and a single exit (last statement).
- Optimizations limited to the scope of a basic block are called intra-block optimizations or **local optimizations**.
- Within a basic block, the following **optimizations** can usually be applied:
  - Common subexpression elimination
  - Dead code elimination
  - Constant folding
  - Temporary variable renaming
  - Statement reordering
  - Algebraic transformations

Partitioning Basic Blocks

Dividing a Quadruple Program into Basic Blocks

(1) Identify the **entry statements** of each basic block:

- The **first statement** of the program, or
- A statement that can be the **target of a conditional or unconditional jump**, or
- A statement immediately **following a conditional jump.**

Partitioning Basic Blocks

(2) For each entry statement identified above, determine its **basic block**.

- It consists of the sequence of statements from this entry statement up to the **next entry statement** (excluding the next entry), or up to a **jump statement** (including the jump), or a **halt statement** (including the halt).

Entry STMT n	**Entry STMT n**	**Entry STMT n**
…	**…**	**…**
…	**…**	**…**
Entry STMT m	**Jump STMT m**	**Halt STMT m**

(3) Any statements not included in a basic block can be **removed from the program**.

Partitioning Basic Blocks

(1) read X
(2) read Y
(3) R:=X mod Y
(4) if R=0 goto (8)
(5) X:=Y
(6) Y:=R
(7) goto (3)
(8) write Y
(9) halt

Partitioning Basic Blocks

(1) read X
(2) read Y
(3) R:=X mod Y
(4) if R=0 goto (8)
(5) X:=Y
(6) Y:=R
(7) goto (3)
(8) write Y
(9) halt

- The **first statement** of the program, or
- A statement that can be the **target of a conditional or unconditional jump**, or
- A statement immediately **following a conditional jump.**

Partitioning Basic Blocks

(1) read X
(2) read Y
(3) R:=X mod Y
(4) if R=0 goto (8)
(5) X:=Y
(6) Y:=R
(7) goto (3)
(8) write Y
(9) halt

- The **first statement** of the program, or
- A statement that can be the **target of a conditional or unconditional jump**, or
- A statement immediately **following a conditional jump.**

Partitioning Basic Blocks

(1) **read X**
(2) **read Y**
(3) **R:=X mod Y**
(4) **if R=0 goto (8)**
(5) **X:=Y**
(6) **Y:=R**
(7) **goto (3)**
(8) **write Y**
(9) **halt**

- The **first statement** of the program, or
- A statement that can be the **target of a conditional or unconditional jump**, or
- A statement immediately **following a conditional jump**.

Partitioning Basic Blocks

(1)	read X
(2)	read Y
(3)	R:=X mod Y
(4)	if R=0 goto (8)
(5)	X:=Y
(6)	Y:=R
(7)	goto (3)
(8)	write Y
(9)	halt

It consists of the sequence of statements from this entry statement up to the

- **next entry statement** (excluding the next entry),
- or up to a **jump statement** (including the jump),
- or a **halt statement** (including the halt).

Question: Will statements (8) and (9) be executed?

Partitioning Basic Blocks

(1) read X
(2) read Y

(3) R:=X mod Y
(4) if R=0 goto (10)

(5) X:=Y
(6) Y:=R
(7) goto (3)

(8) X:=Y+1
(9) Y:=X-1

(10) write Y
(11) halt

Exercise P306 2

Partitioning Basic Blocks

```
(1)         read A, B
(2)         F:=1
(3)         C:=A*A
(4)         D:=B*B
(5)         if  C<D goto L1
(6)         E:=A*A
(7)         F:=F+1
(8)         E:=E+F
(9)         write E
(10)        halt
(11) L1:    E:=B*B
(12)        F:=F+2
(13)        E:=E+F
(14)        write E
(15)        if  E>100 goto L2
(16)        halt
(17) L2:    F:=F-1
(18)        goto L1
```

Partitioning Basic Blocks

```
(1)        read A, B
(2)        F:=1
(3)        C:=A*A
(4)        D:=B*B
(5)        if  C<D goto L1
(6)        E:=A*A
(7)        F:=F+1
(8)        E:=E+F
(9)        write E
(10)       halt
(11) L1:E:=B*B
(12)       F:=F+2
(13)       E:=E+F
(14)       write E
(15)       if  E>100 goto L2
(16)       halt
(17) L2:F:=F-1
(18)       goto L1
```

- The **first statement** of the program, or
- A statement that can be the **target of a conditional or unconditional jump**, or
- A statement immediately **following a conditional jump.**

Partitioning Basic Blocks

```
(1)      read A, B
(2)      F:=1
(3)      C:=A*A
(4)      D:=B*B
(5)      if  C<D goto L1
(6)      E:=A*A
(7)      F:=F+1
(8)      E:=E+F
(9)      write E
(10)     halt
(11) L1: E:=B*B
(12)     F:=F+2
(13)     E:=E+F
(14)     write E
(15)     if  E>100 goto L2
(16)     halt
(17) L2: F:=F-1
(18)     goto L1
```

- The **first statement** of the program, or
- A statement that can be the **target of a conditional or unconditional jump**, or
- A statement immediately **following a conditional jump.**

Partitioning Basic Blocks

```
(1)        read A, B
(2)        F:=1
(3)        C:=A*A
(4)        D:=B*B
(5)        if  C<D goto L1
(6)        E:=A*A
(7)        F:=F+1
(8)        E:=E+F
(9)        write E
(10)       halt
(11) L1:E:=B*B
(12)       F:=F+2
(13)       E:=E+F
(14)       write E
(15)       if  E>100 goto L2
(16)       halt
(17) L2:F:=F-1
(18)       goto L1
```

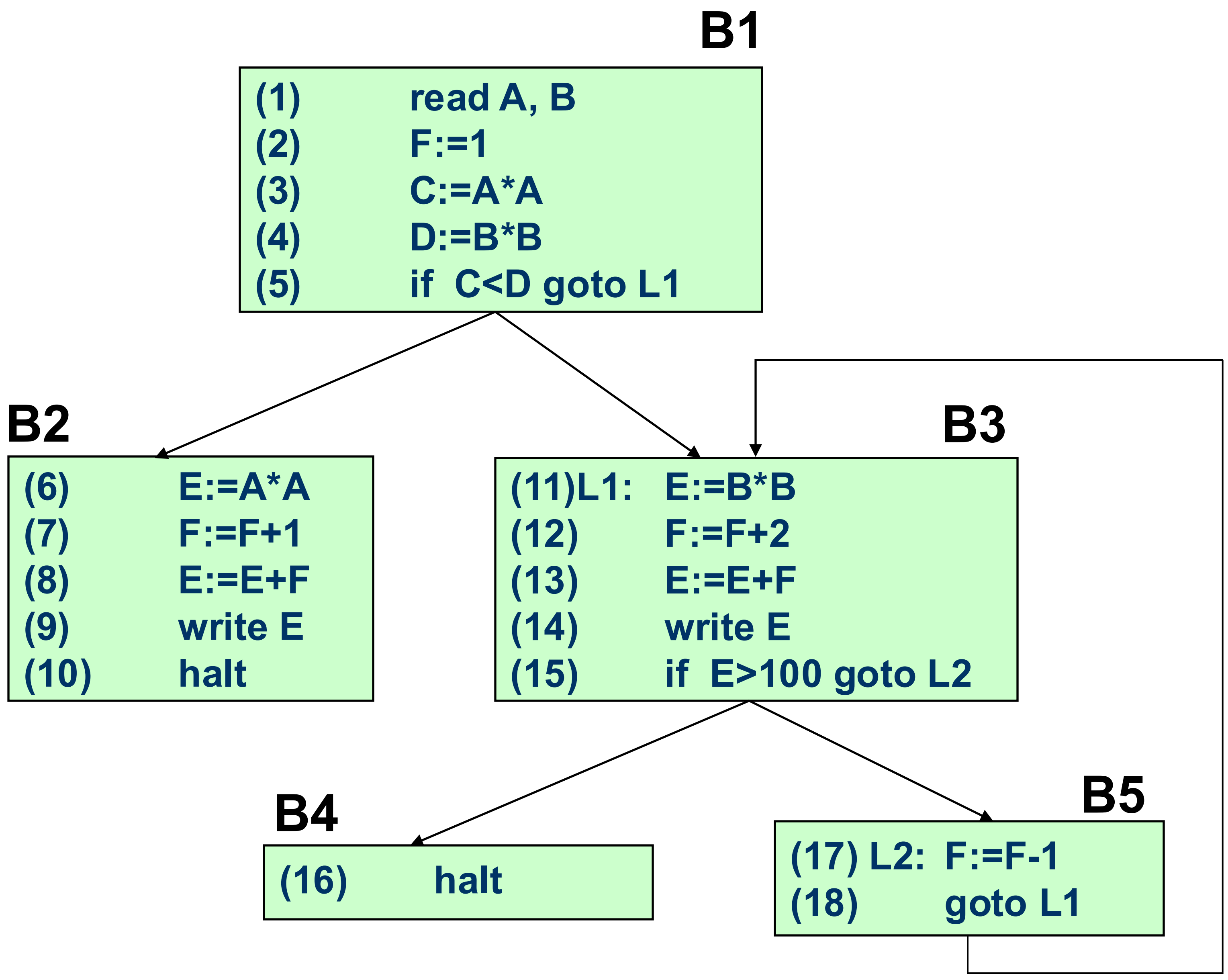
- The **first statement** of the program, or
- A statement that can be the **target of a conditional or unconditional jump**, or
- A statement immediately **following a conditional jump.**

Partitioning Basic Blocks

```
(1)       read A, B                 B1
(2)       F:=1
(3)       C:=A*A
(4)       D:=B*B
(5)       if  C<D goto L1
(6)       E:=A*A                    B2
(7)       F:=F+1
(8)       E:=E+F
(9)       write E
(10)      halt
(11) L1:E:=B*B                      B3
(12)      F:=F+2
(13)      E:=E+F
(14)      write E
(15)      if  E>100 goto L2
(16)      halt                      B4
(17) L2:F:=F-1                      B5
(18)      goto L1
```

It consists of the sequence of statements from this entry statement up to the

- **next entry statement** (excluding the next entry),
- or up to a **jump statement** (including the jump),
- or a **halt statement** (including the halt).


B1
(1) read A, B
(2) F:=1
(3) C:=A*A
(4) D:=B*B
(5) if C<D goto L1
B2
(6) E:=A*A
(7) F:=F+1
(8) E:=E+F
(9) write E
(10) halt
B3
(11)L1: E:=B*B
(12) F:=F+2
(13) E:=E+F
(14) write E
(15) if E>100 goto L2
B4
(16) halt
B5
(17) L2: F:=F-1
(18) goto L1

Optimization Methods

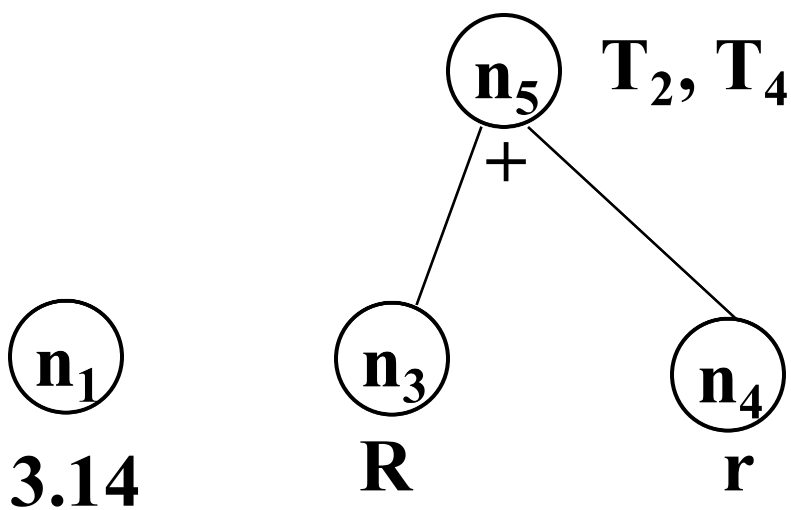
- Within a basic block, the following optimizations can usually be applied:
  - Eliminate Common Subexpressions
  - Remove Dead Assignments
  - Constant Folding (Combine Known Values)
  - Rename Temporary Variables
  - Reorder Statements
  - Algebraic Transformations

Optimization Methods

- combine known values
  - T1 := 2 … T2 := 4 * T1 → T2 := 8
- Algebraic transformations
  - Delete x := x + 0 or x := x * 1
  - x := y ** 2 (via function) → x := y * y
- Swap statement positions
  - T1 := b + c
  - T2 := x + y
- Rename temporary variables
  - T := b + c (T is temporary) → S := b + c

DAG representation of a basic block

- **Leaf nodes**: marked with an **identifier** or **constant**, representing the value of that variable or constant.
- **Internal nodes**: marked with an operator, representing the result of applying that **operator** to the values of its successor nodes.
- **Additional identifiers**: may be attached to nodes, indicating variables that **hold the value** represented by that node.



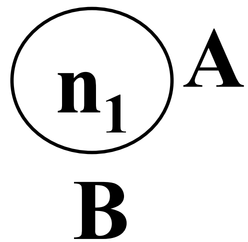
DAG representation of a basic block

- DAG nodes corresponding to each quadruple

Quadruple | **DAG**

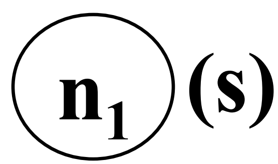
Type 0: A:=B

(:=， B， -， A)



Type 0: goto (s)

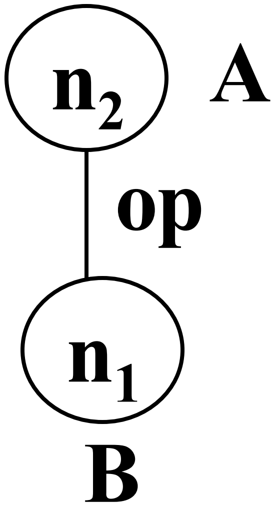
(j， -， -， (s))



Quadruple　　　　DAG

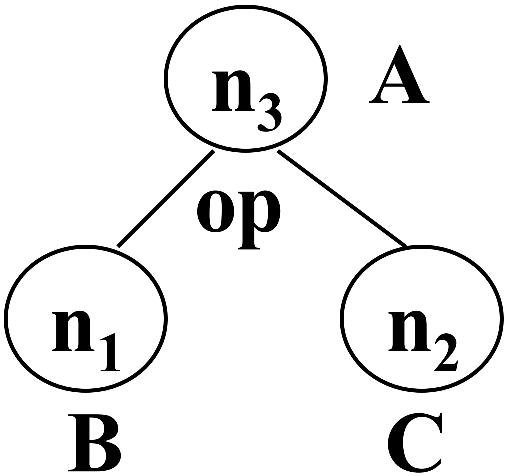
Type 1: A:=op B

(op， B， -， A)



Type 2: A:=B op C

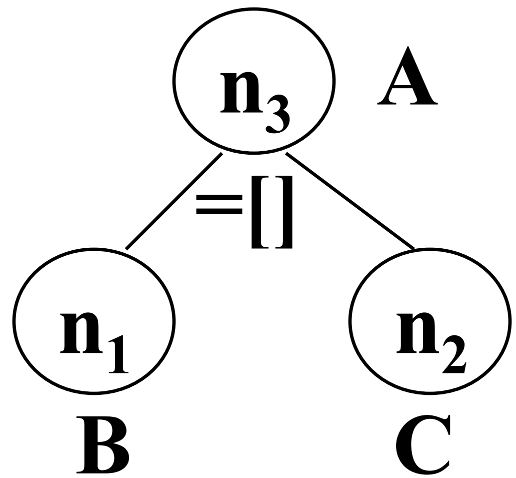
(op， B， C， A)



Quadruple　　　　　　DAG

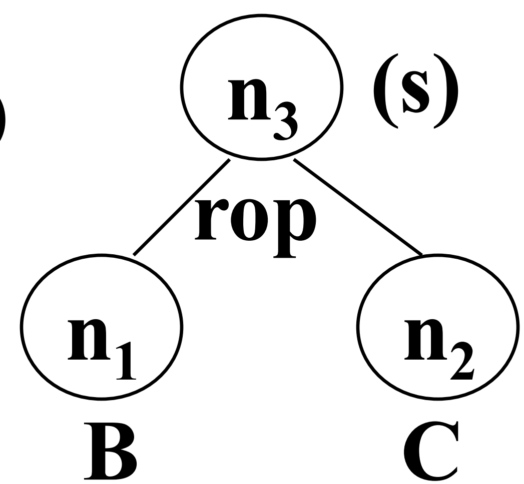
Type 2: A:=B[C]

(=[]，B，C，A)



Type 2: if B rop C goto (s)

(jrop，B，C，(s))

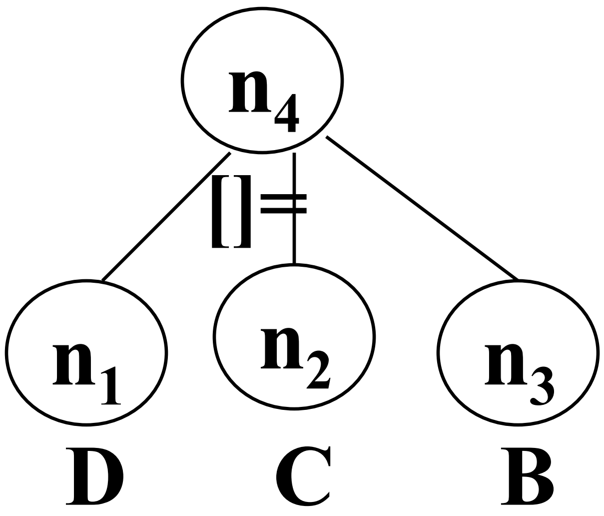


Quadruple

Type 3: D[C]:=B

([]=， B， -， D[C])

DAG



Construct the DAG for the following basic block G

(1) $T_0:=3.14$
(2) $T_1:=2*T_0$
(3) $T_2:=R+r$
(4) $A:=T_1*T_2$
(5) $B:=A$
(6) $T_3:=2*T_0$
(7) $T_4:=R+r$
(8) $T_5:=T_3*T_4$
(9) $T_6:=R-r$
(10) $B:=T_5*T_6$

(1) T_0:=3.14

(2) T_1:=2*T_0

(3) T_2:=R+r

(4) A:=T_1*T_2

(5) B:=A

(6) T_3:=2*T_0

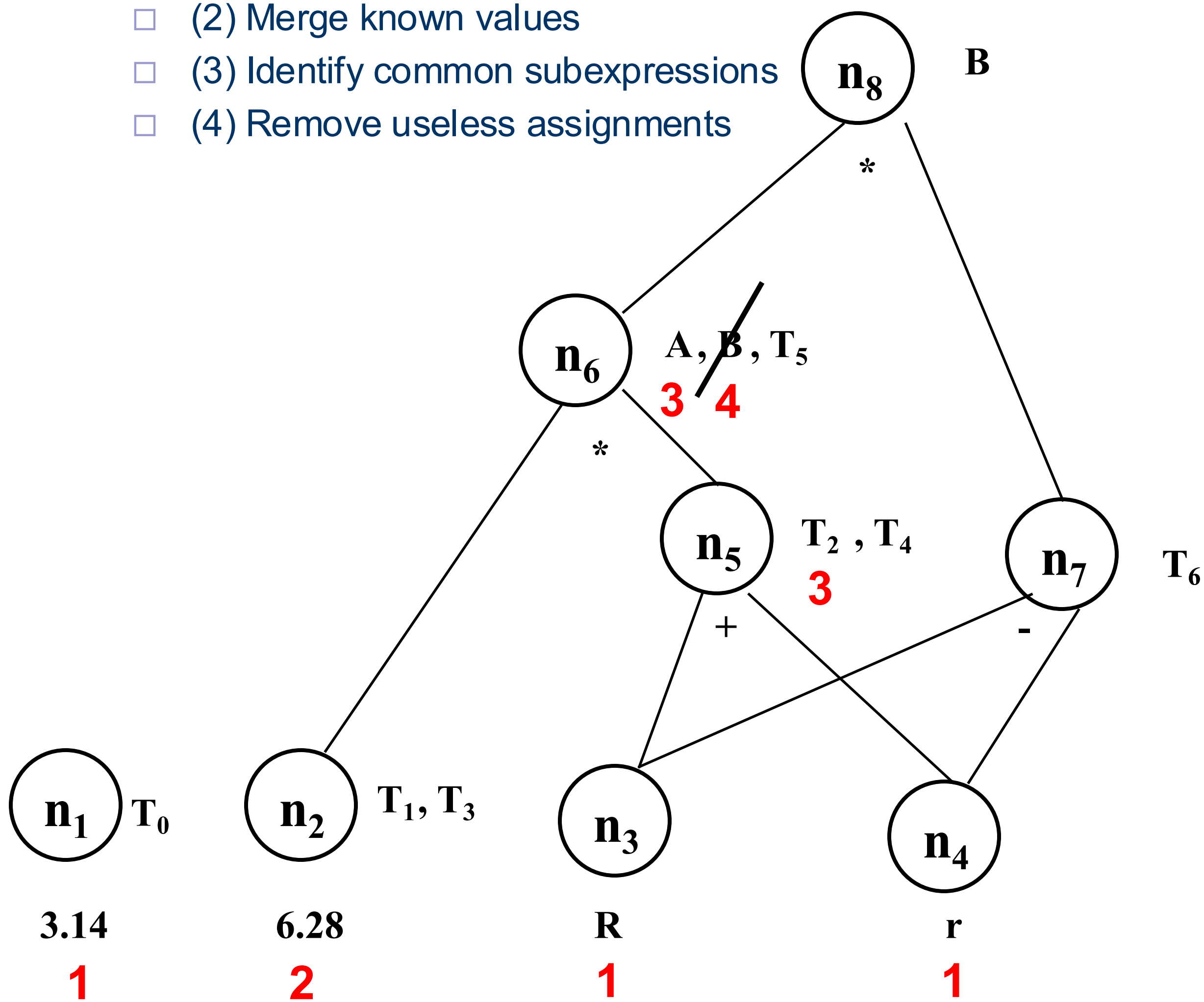
(7) T_4:=R+r

(8) T_5:=T_3*T_4

(9) T_6:=R-r

(10) B:=T_5*T_6

- Steps
  - (1) Prepare nodes for operands
  - (2) Merge known values
  - (3) Identify common subexpressions
  - (4) Remove useless assignments

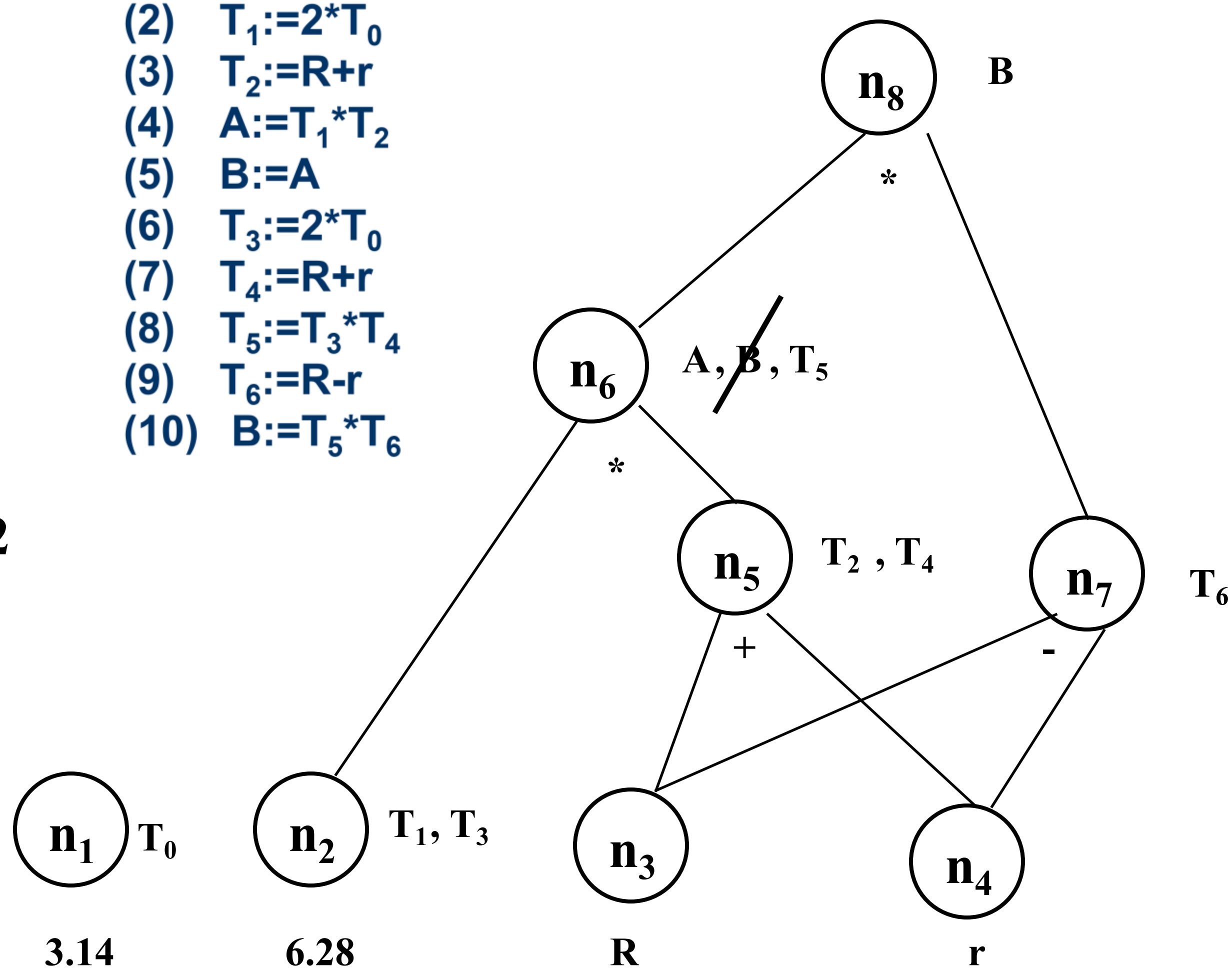


After opt.

(1) $T_0:=3.14$
(2) $T_1:=6.28$
(3) $T_3:=6.28$
(4) $T_2:=R+r$
(5) $T_4:=T_2$
(6) $A:=6.28*T_2$
(7) $T_5:=A$
(8) $T_6:=R-r$
(9) $B:=A*T_6$

Before opt.

(1) $T_0:=3.14$
(2) $T_1:=2*T_0$
(3) $T_2:=R+r$
(4) $A:=T_1*T_2$
(5) $B:=A$
(6) $T_3:=2*T_0$
(7) $T_4:=R+r$
(8) $T_5:=T_3*T_4$
(9) $T_6:=R-r$
(10) $B:=T_5*T_6$

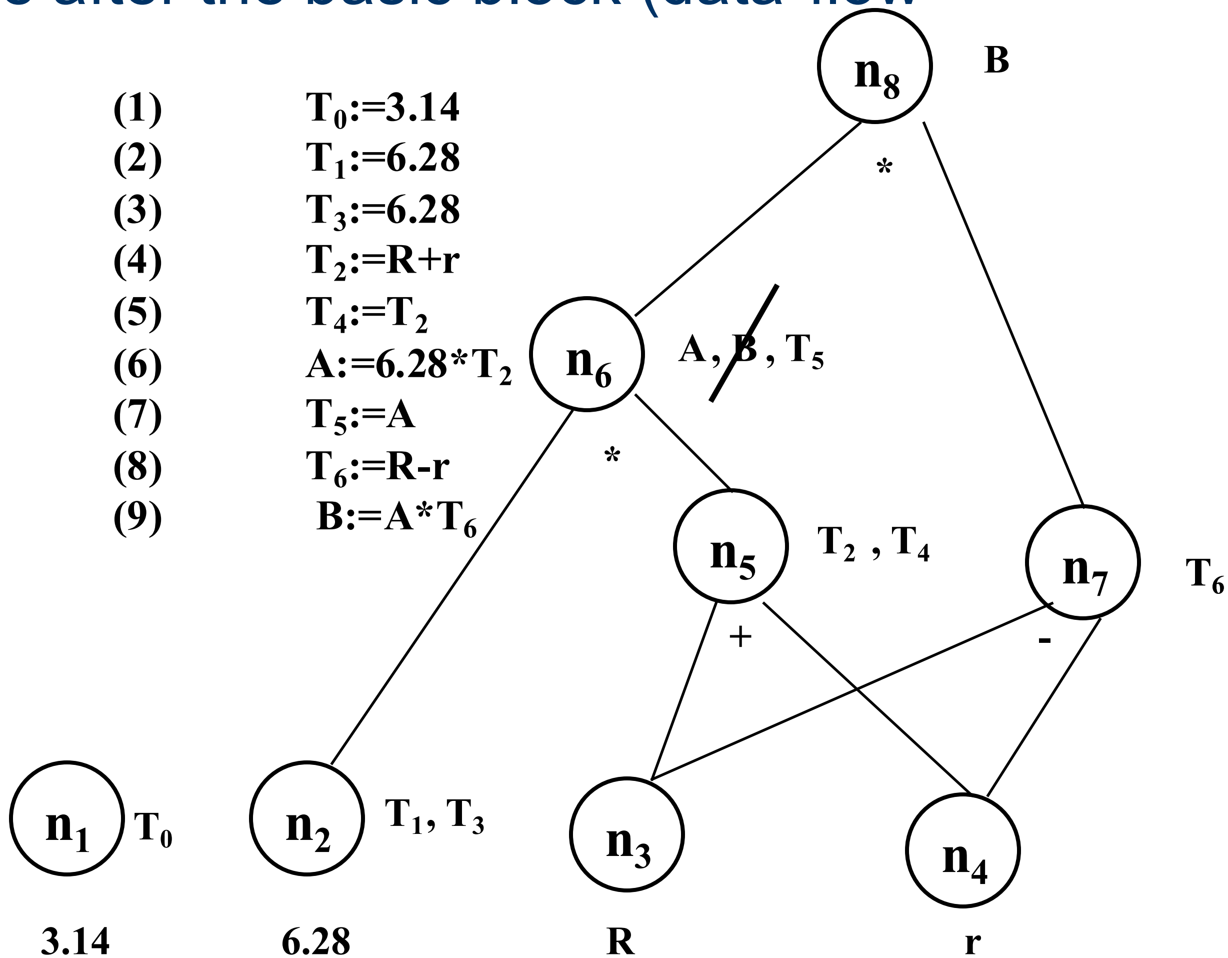


- **Optimized quadruples** — assuming only A and B are live after the basic block (data-flow analysis)

(1) T_2:=R+r
(2) A:=6.28*T_2
(3) T_6:=R-r
(4) B:=A*T_6

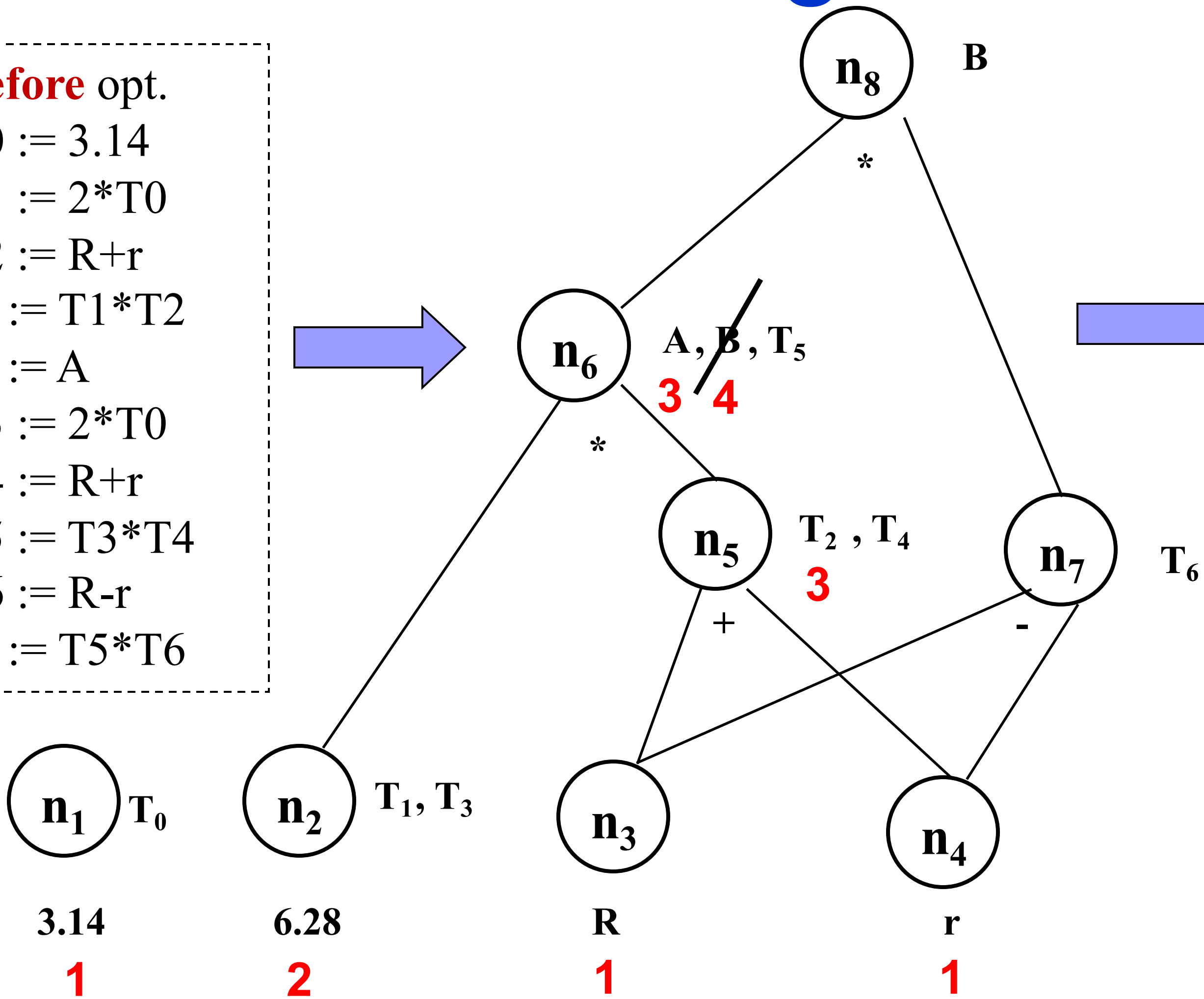
(1) S_1:=R+r
(2) A:=6.28*S_1
(3) S_2:=R-r
(4) B:=A*S_2

(1)	T_0:=3.14
(2)	T_1:=6.28
(3)	T_3:=6.28
(4)	T_2:=R+r
(5)	T_4:=T_2
(6)	A:=6.28*T_2
(7)	T_5:=A
(8)	T_6:=R-r
(9)	B:=A*T_6



Effect of DAG algorithm

Before opt.
T0 := 3.14
T1 := 2*T0
T2 := R+r
A := T1*T2
B := A
T3 := 2*T0
T4 := R+r
T5 := T3*T4
T6 := R-r
B := T5*T6



assuming only A and B are **live** after the basic block

After opt.
S1:=R+r
A:=6.28*S1
S2:=R-r
B:=A*S2

DAG construction algorithm for a basic block

- For each quadruple in the basic block, perform the following steps sequentially:
  - **1. Prepare** nodes for operands
  - **2. Merge** known values
  - **3. Find** common subexpressions
  - **4. Remove** useless assignments

1. Prepare

```
Step 1: Prepare operand nodes

if NODE(B) is undefined then
    create leaf node labeled B
    NODE(B) := that node
end if

switch (quadruple type) of current statement:
    case Type 0:   // A := B
        n := NODE(B)
        goto Step 4

    case Type 1:   // A := op B
        goto Step 2(1)

    case Type 2:   // A := B op C  or A := B[c]
        if NODE(C) is undefined then
            create leaf node labeled C
            NODE(C) := that node
        end if
        goto Step 2(2)
end switch
```

2. Merge

2.1 single operand branch
2.2 two operands branch
2.3 if constant, Type 1 calculate
2.4 if constant, Type 2 calculate

```
Step 2: Merge known values

case current quadruple type of operand(s):

    // (1) Single operand
    if NODE(B) is constant then
        goto Step 2(3)
    else
        goto Step 3(1)
    end if

    // (2) Two operands
    if NODE(B) and NODE(C) are constants the
        goto Step 2(4)
    else
        goto Step 3(2)
    end if
```

```
Step 2(3):   // compute op B
    P := compute(op, B)
    if NODE(B) was newly created in this quadruple then
        delete NODE(B)
    end if
    if NODE(P) is undefined then
        create leaf node n labeled P
        NODE(P) := n
    end if
    goto Step 4

Step 2(4):   // compute B op C
    P := compute(B op C)
    if NODE(B) or NODE(C) was newly created in this quadruple t
        delete those nodes
    end if
    if NODE(P) is undefined then
        create leaf node n labeled P
        NODE(P) := n
    end if
    goto Step 4
```

3. Find

```
Step 3: Find common subexpressions

case current quadruple type:

    // (1) Unary operation: A := op B
    if exists node n in DAG such that:
        successor(n) = NODE(B) and label(n) = op
    then
        use existing node as n
    else
        n := create new node labeled op
        set successor(n) := NODE(B)
    end if
    goto Step 4

    // (2) Binary operation: A := B op C
    if exists node n in DAG such that:
        left_successor(n) = NODE(B)
        right_successor(n) = NODE(C)
        label(n) = op
    then
        use existing node as n
    else
        n := create new node labeled op
        set left_successor(n) := NODE(B)
        set right_successor(n) := NODE(C)
    end if
    goto Step 4
```

4. Remove

```
Step 4: Delete useless assignments (finish assignment)

if NODE(A) is undefined then
    attach A to node n
    NODE(A) := n
else
    if NODE(A) is not a leaf node then
        remove A from the additional identifiers of NODE(A)
    end if
    attach A to node n
    NODE(A) := n
end if

goto process next quadruple
```

Conclusion

- **Merge** known values
- **Find** common subexpressions
- **Delete** useless assignments
- Identifiers defined outside the block but referenced inside → **labels on leaf nodes**
- Identifiers defined inside the block and still live after it → **additional identifiers on DAG nodes**

Exercise P306 3(B_1)

construct the DAG for the following basic block G

(1) A:=B*C
(2) D:=B/C
(3) E:=A+D
(4) F:=2*E
(5) G:=B*C
(6) H:=G*G
(7) F:=H*G
(8) L:=F
(9) M:=L

(1) A:=B*C

(2) D:=B/C

(3) E:=A+D

(4) F:=2*E

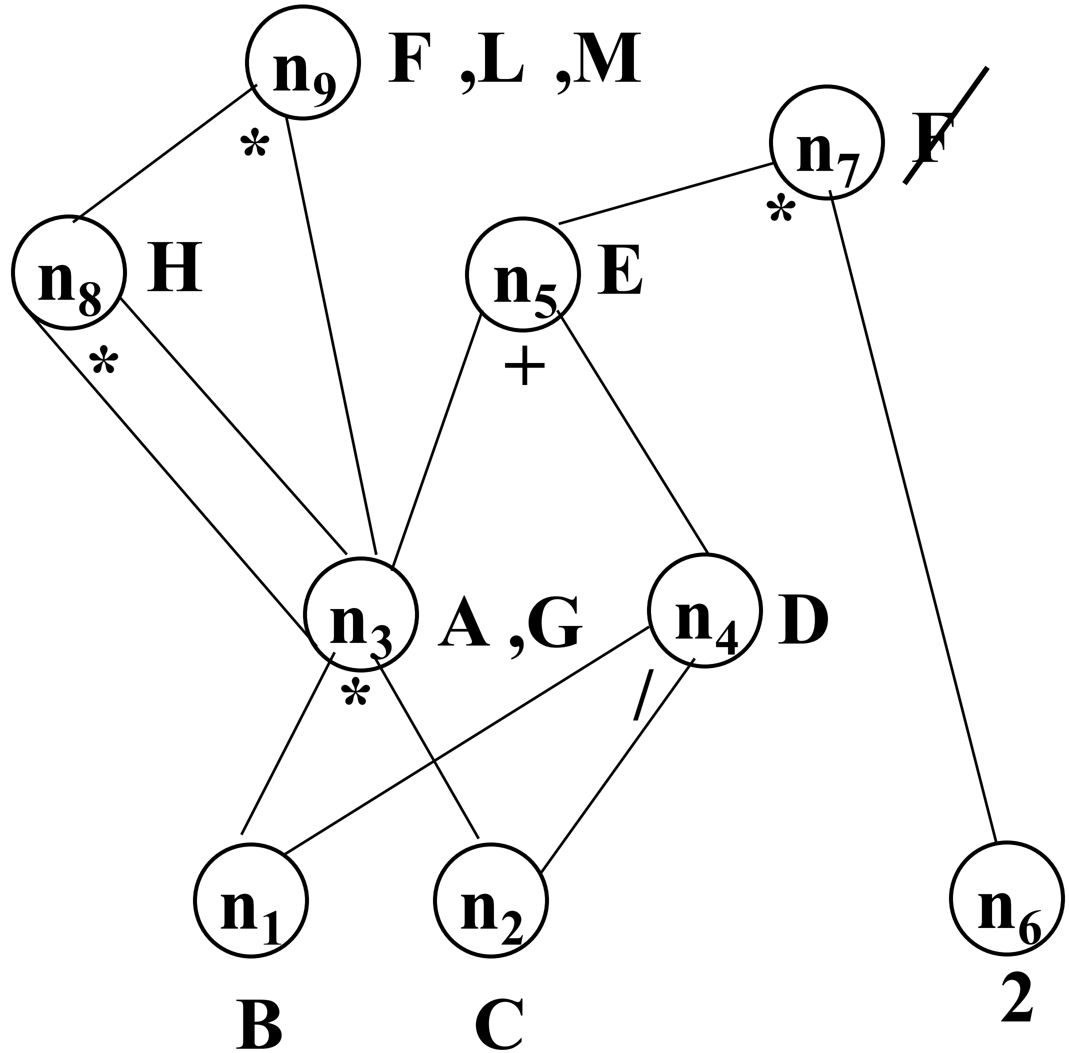
(5) G:=B*C

(6) H:=G*G

(7) F:=H*G

(8) L:=F

(9) M:=L



Look G, L, M

(1) A:=B*C

(2) D:=B/C

(3) E:=A+D

~~(4) F:=2*E~~

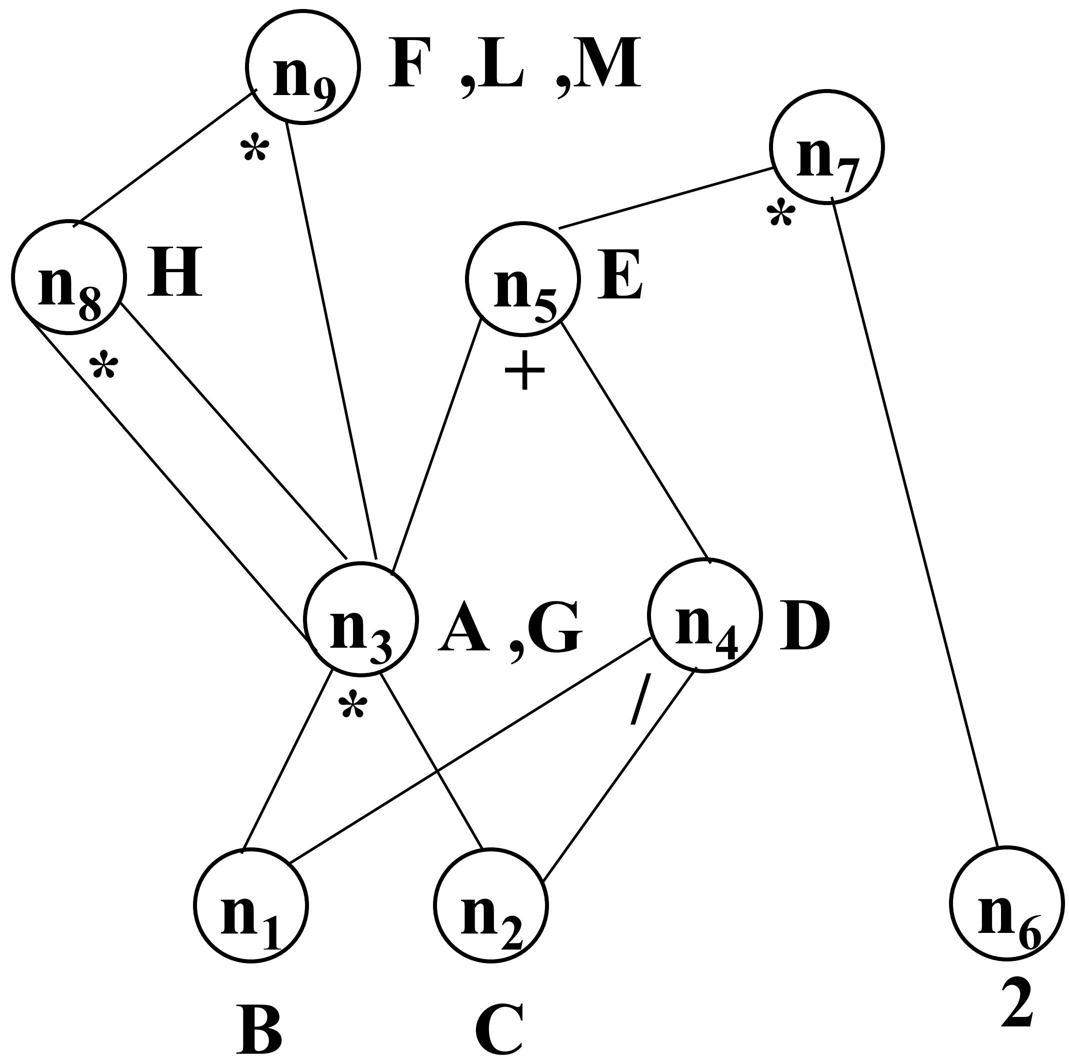
(5) G:=A

(6) H:=G*G

(7) F:=H*G

(8) L:=F

(9) M:=L



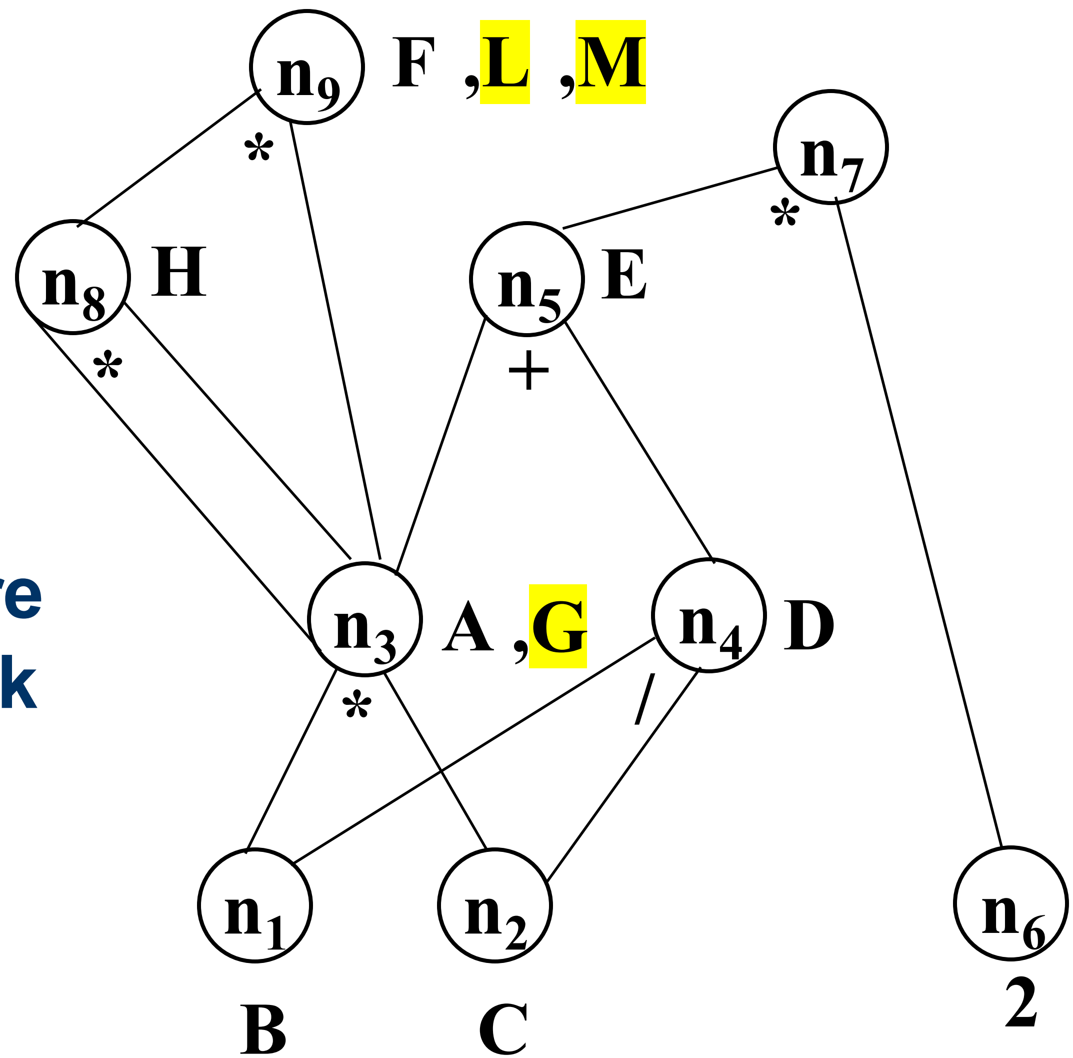
Assume only G, L, M are live after the block

(1) G:=B*C

(2) H:=G*G

(3) L:=H*G

(4) M:=L



Assume only L are live after the block

(1) G:=B*C

(2) H:=G*G

(3) L:=H*G

Quiz-Canvas

- **ch10 Optimization - DAG**

Outline

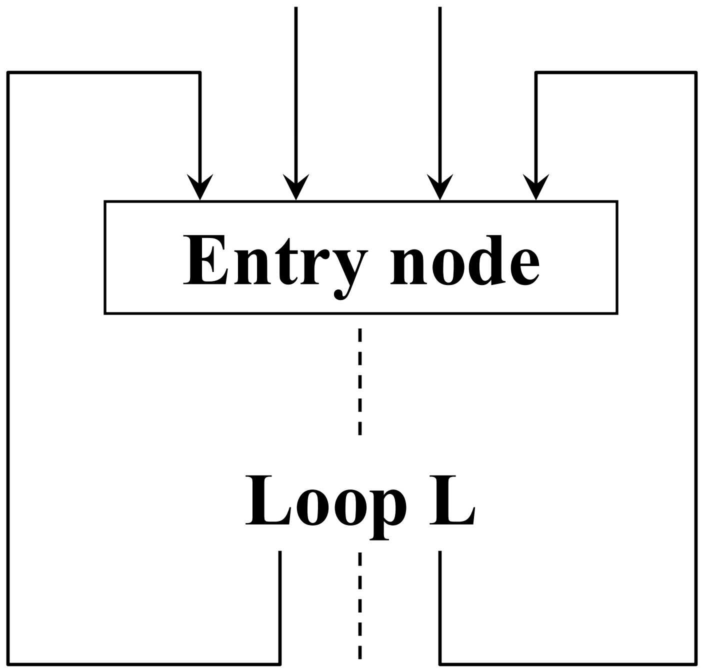
- Overview
- Local Optimization
- Loop Optimization

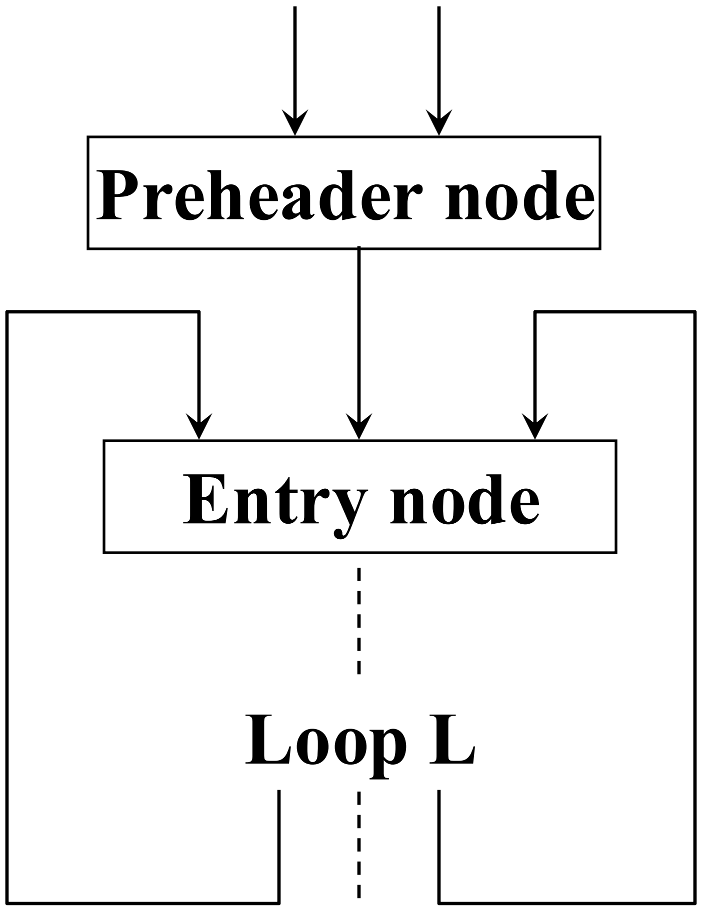
Loop optimization

- For code inside loops,
  - **Code motion (hoisting)**
  - Strength reduction
  - Induction variable elimination (transform loop control condition)

Code motion (hoisting)

Loop-invariant computation: For the quadruple `A := B op C`, if `B` and `C` are constants, or the definitions reaching `B` and `C` are outside the loop, then move the loop-invariant computation outside the loop body.





for I:=1 to 10 do A[I, 2*J] := A[I, 2*J] + 1

B_1

(1) I:=1

B_2

(2) if I>10 goto (15)

B_3

(3) T_1=2*J
(4) T_2=10*I
(5) T_3= T_2+ T_1
(6) T_4=addr(A)-11
(7) T_5=2*J
(8) T_6=10*I
(9) T_7= T_6+ T_5
(10) T_8=addr(A)-11
(11) T_9= T_8[T_7]
(12) T_4[T_3]= T_9+1
(13) I:=I+1
(14) goto B_2

(15)

B_1

(1) I:=1

B_2'

(3) T_1=2*J
(6) T_4=addr(A)-11
(7) T_5=2*J
(10) T_8=addr(A)-11

B_2

(2) if I>10 goto (15)

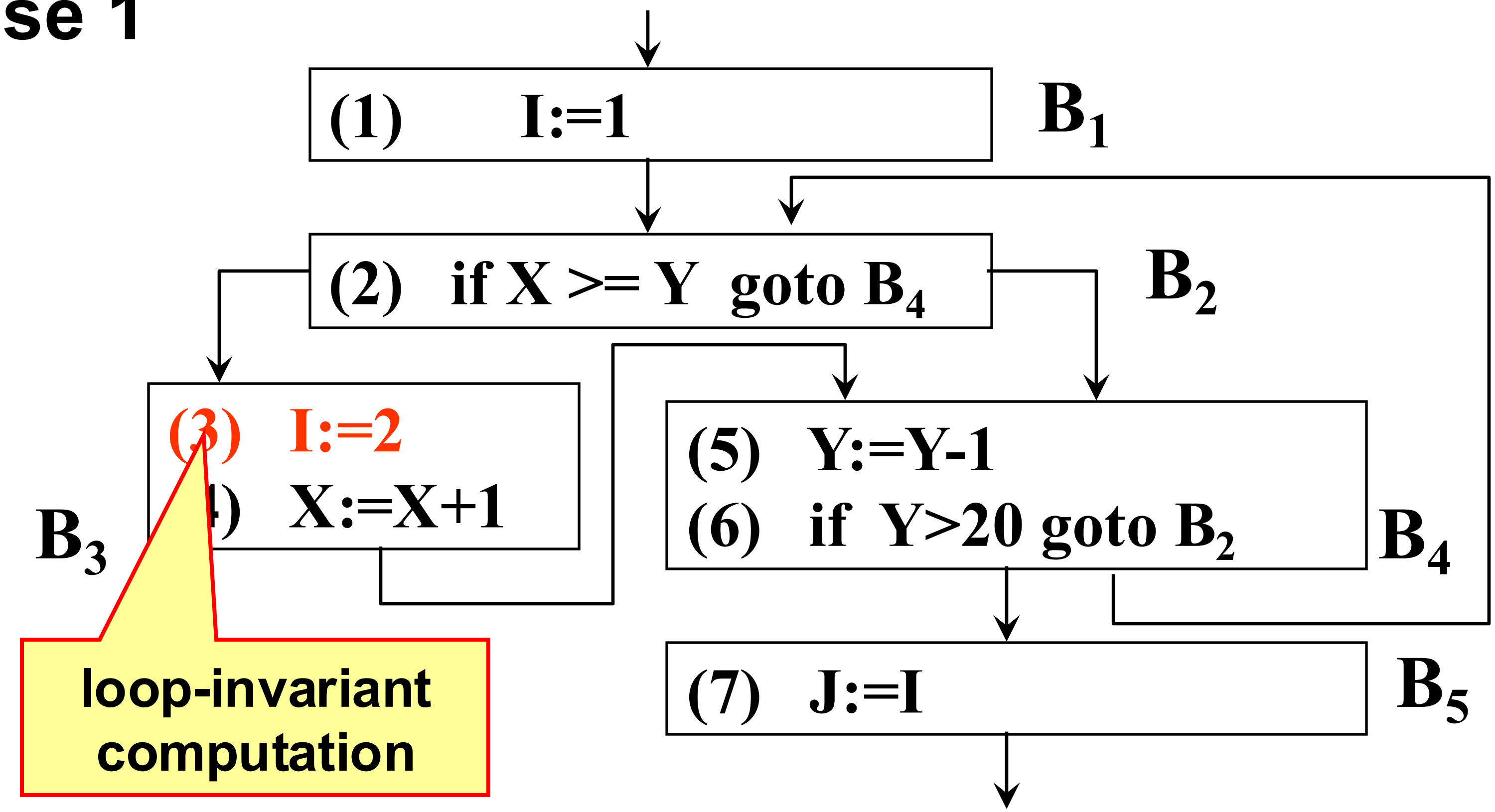
B_3

(4) T_2=10*I
(5) T_3= T_2+T_1
(8) T_6=10*I
(9) T_7= T_6+ T_5
(11) T_9= T_8[T_7]
(12) T_4[T_3]= T_9+1
(13) I:=I+1
(14) goto B_2

(15)

Case 1

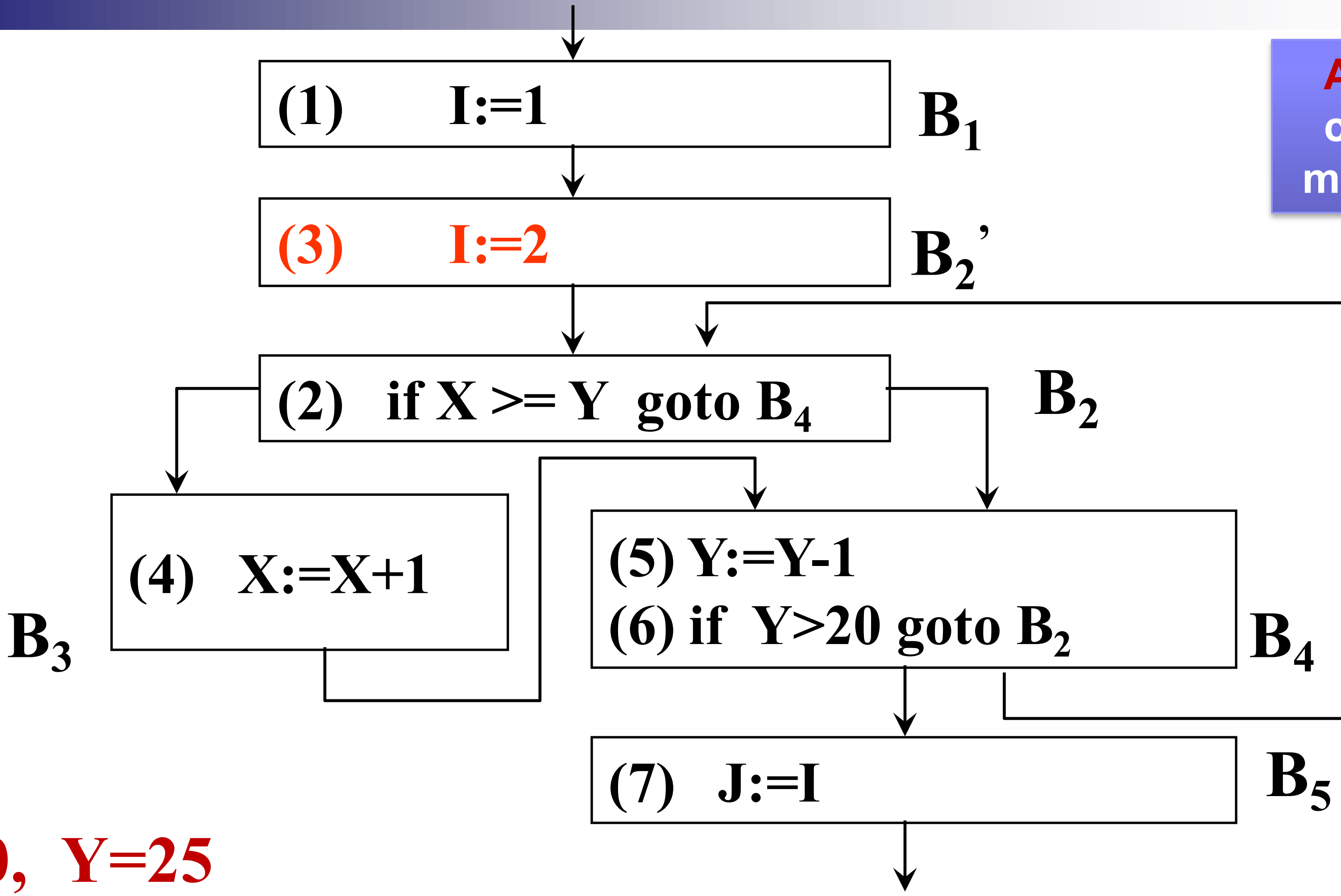
Before code motion



X=30, Y=25

$B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow \cdots \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$

J=1, I=1



After code motion

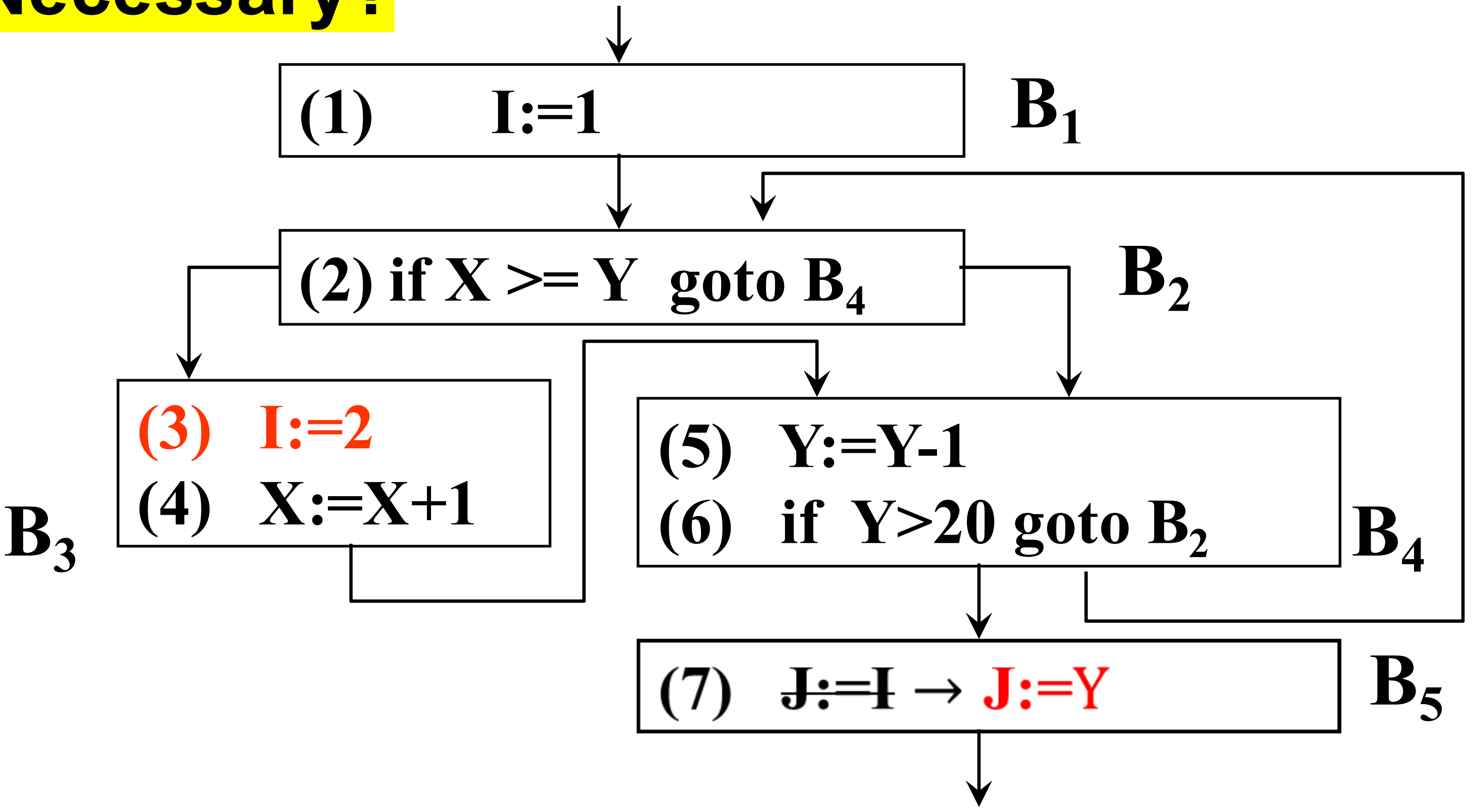
X=30, Y=25

$B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow \cdots \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$

J=2, I=2

Condition for code motion of S(A := B OP C) (1):
The invariant computation must be in a node that is always executed in the loop.

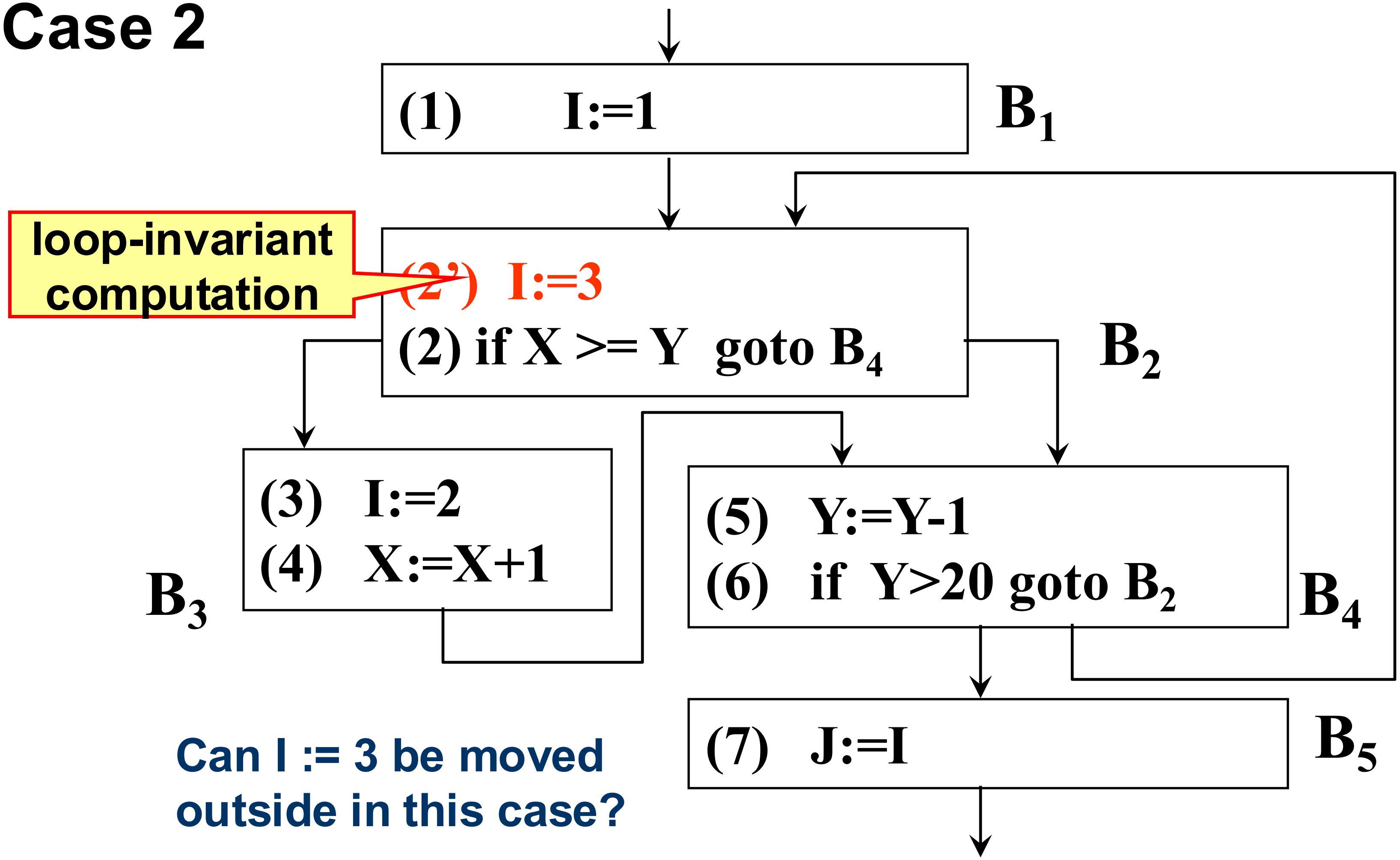
Necessary?



Revised code motion condition: If the value of I is no longer referenced after the loop, it can be moved outside even if it's not in a node that is always executed.

Case 2

Before code motion

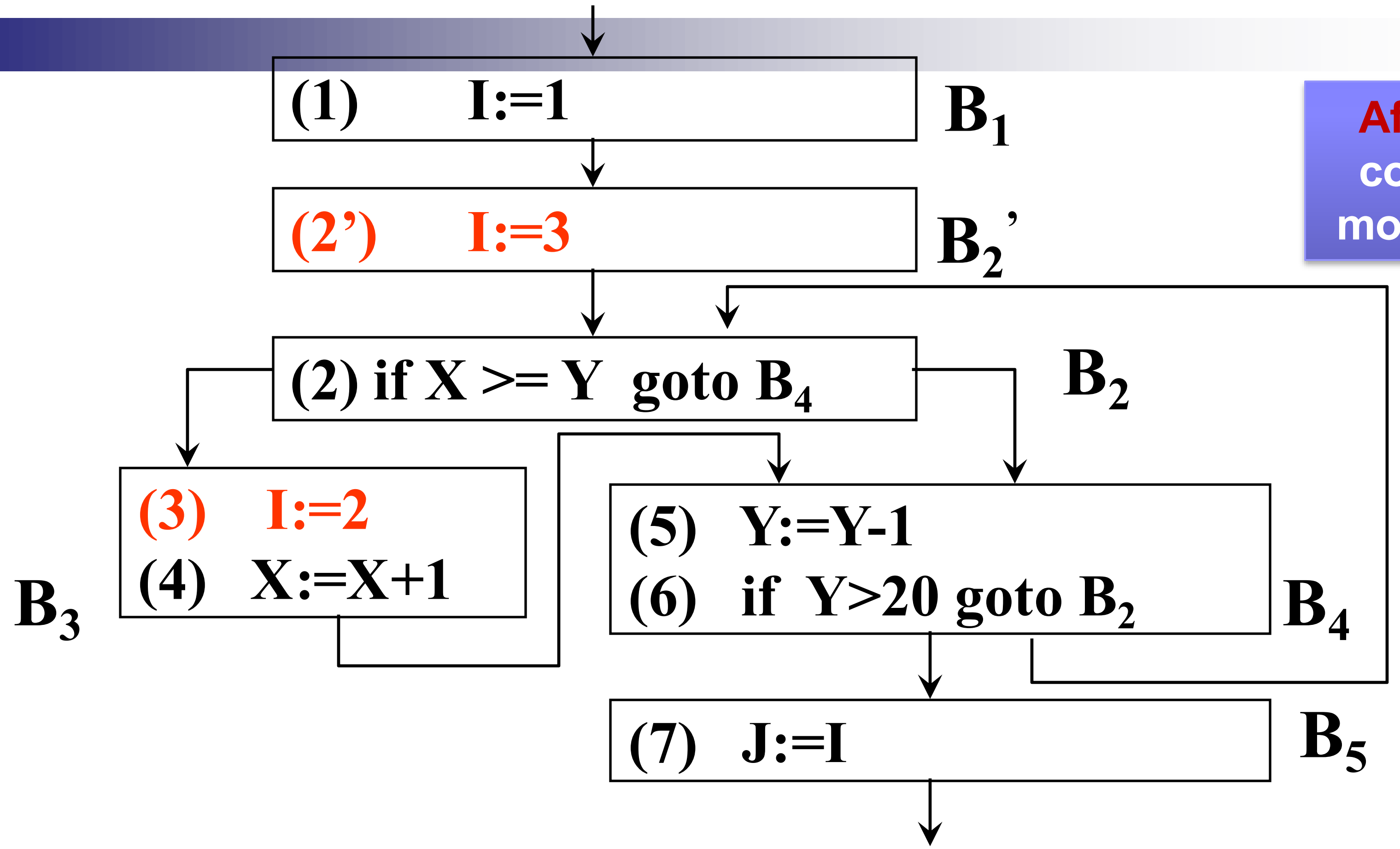


Can I := 3 be moved outside in this case?

X=21,Y=22

$B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$

I=3, J=3



After code motion

$B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$

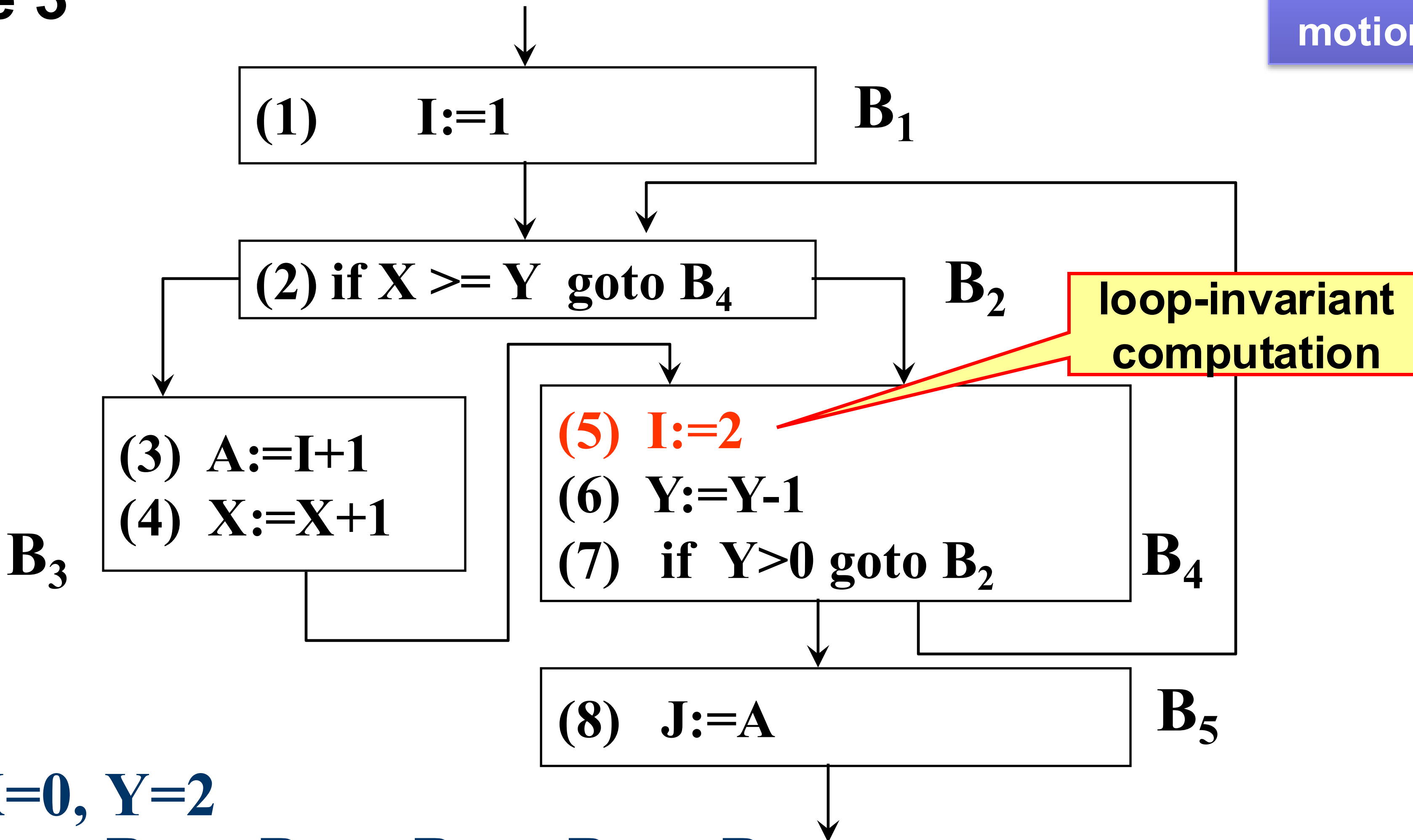
I=2, J=2

Condition for code motion of S(A := B OP C) (2):

A must not be assigned elsewhere in the loop to allow moving the loop-invariant computation A := B OP C outside.

Case 3

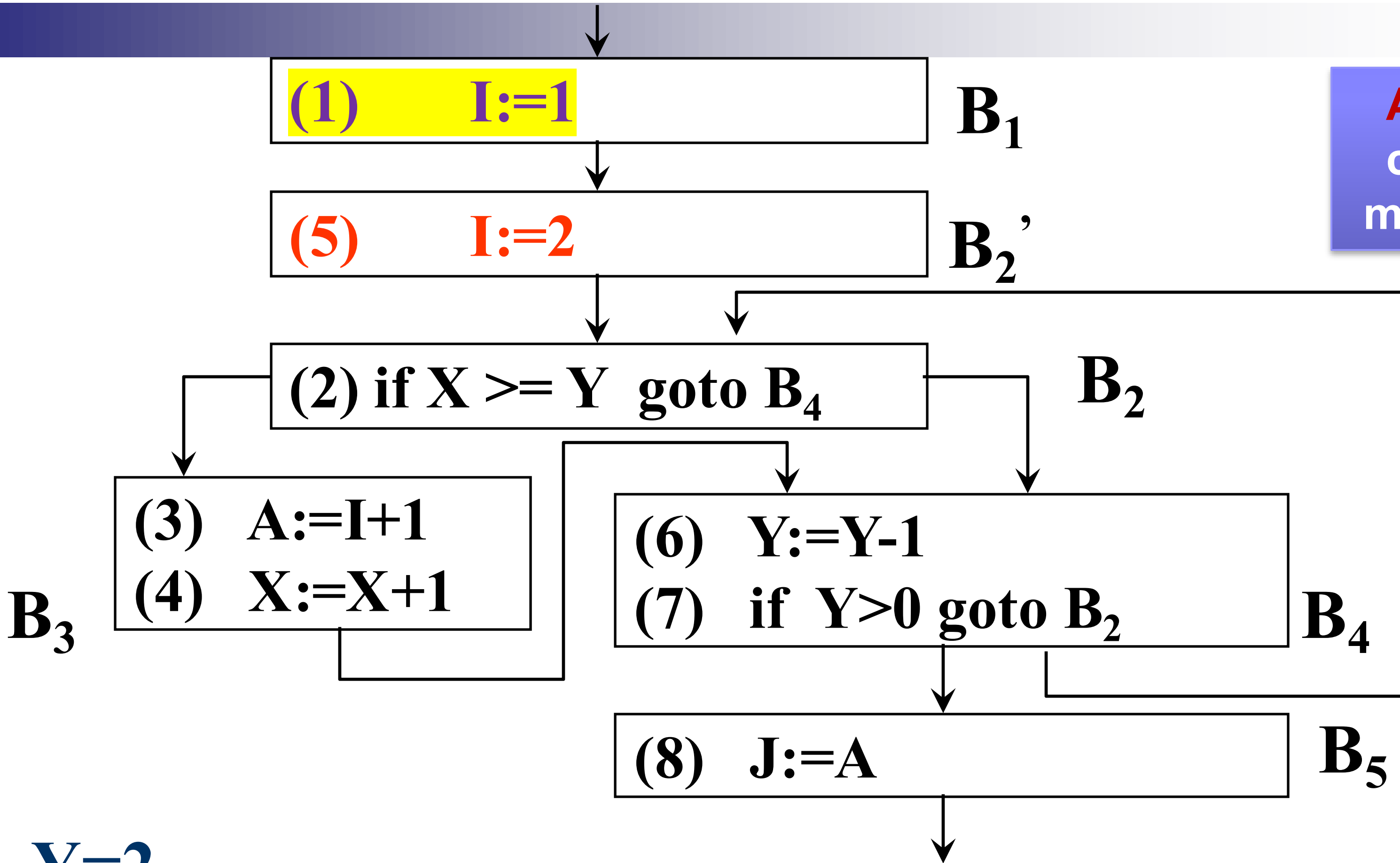
Before code motion



$X=0,\ Y=2$

$B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$

$A=2,\ \ J=2$



After code motion

$X=0,\ Y=2$

$B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$

$A=3,\ J=3$

Condition for code motion of S(A := B OP C) (3):

All uses of A in the loop must be reachable only from the definition of A in S.

Algorithm to find loop-invariant computations in loop L

- **Step 1:**
  - Examine each quadruple in each basic block of L.
  - If all operands are constants, or their definitions are outside L, mark the quadruple as **"loop-invariant"**.
- **Step 2:**
  - Repeat Step 3 until no new quadruples are marked as **"loop-invariant"**.
- **Step 3:**
  - For each quadruple not yet marked:
    - If all operands are constants, or their definitions are outside L （same with step 1）, or have only one reaching definition and that definition is already marked **"loop-invariant"**,
    - Then mark the current quadruple as **"loop-invariant"**.

Example

Single Assignment

```
t1 = a + b;     // Q1

t2 = t1 * 2;    // Q2

x  = t2 + c;    // Q3
```

Multiple Assignment

```
if (cond)
    t1 = a + b;    // Q1
else
    t1 = a + b;    // Q2
t2 = t1 * 2;       // Q3
```

?

Code Motion Algorithm

1, Find all **loop-invariant** computations in L.

2, For each invariant computation s (forms: A := B op C, A := op B, or A := B), check if it **meets all conditions**:

(i) The node containing s is a must-execute node of L, or A is not live after leaving L.

(ii) A is not assigned anywhere else in L.

(iii) All uses of A in L can only be reached through the assignment in s.

3, Following the order of invariant computations found in step 1, move each s satisfying step 2 conditions to the **preheader** of L.

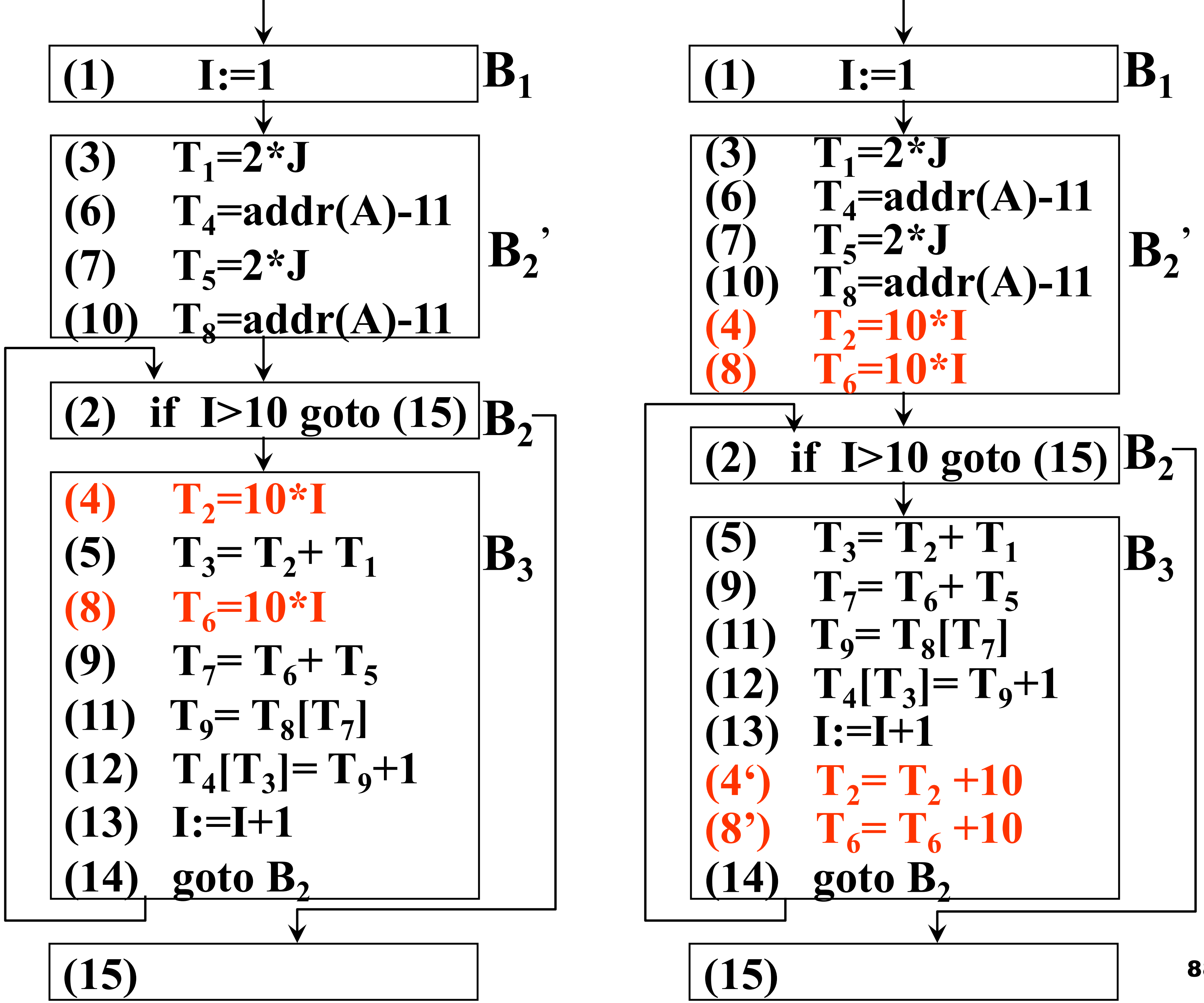
Note: If operands B or C are assigned in L, move s only after their assignments have already been moved to the preheader.

Loop optimization

- For code inside loops,
  - Code motion (hoisting)
  - **Strength reduction**
  - Induction variable elimination (transform loop control condition)

Strength Reduction

- Transform long-running operations in the program into shorter ones.
  - Such as: replace multiplication in loops with addition.



(1) I:=1
B1
(3) T1=2*J
(6) T4=addr(A)-11
(7) T5=2*J
(10) T8=addr(A)-11
B2'
(2) if I>10 goto (15)
B2
(4) T2=10*I
(5) T3= T2+ T1
(8) T6=10*I
(9) T7= T6+ T5
(11) T9= T8[T7]
(12) T4[T3]= T9+1
(13) I:=I+1
(14) goto B2
B3
(15)
(1) I:=1
B1
(3) T1=2*J
(6) T4=addr(A)-11
(7) T5=2*J
(10) T8=addr(A)-11
(4) T2=10*I
(8) T6=10*I
B2'
(2) if I>10 goto (15)
B2
(5) T3= T2+ T1
(9) T7= T6+ T5
(11) T9= T8[T7]
(12) T4[T3]= T9+1
(13) I:=I+1
(4') T2= T2 +10
(8') T6= T6 +10
(14) goto B2
B3
(15)

Loop optimization

- For code inside loops,
  - Code motion (hoisting)
  - Strength reduction
  - **Induction variable elimination (transform loop control condition)**

Eliminating Induction Variables

- Basic Induction Variable (BIV)
  - In a loop, if a variable I is assigned only in the form I := I ± C
  - And C is loop-invariant, I is called a basic induction variable.
- Derived Induction Variable (DIV)
  - If I is a BIV, and another variable J's value in the loop can always be expressed as a linear function of I:
    J = C1 * I ± C2
  - where C1 and C2 are loop-invariant, J is an induction variable, and belongs to the same family as I.
- Note: A basic induction variable is also an induction variable

Strength Reduction and Induction Variable Elimination

- 1. Using loop-invariant computation information, identify all **basic induction** variables in the loop
- 2. Identify all other **induction** variables A
- 3. For each induction variable A found in step 2 perform **strength reduction**
- 4. Remove **dead assignments** to induction variables
- 5. Eliminate basic induction variables:

  If a basic induction variable B is not used after the loop, and within the loop:

    It appears only in its own recursive assignment, and

    In statements like if B rop Y goto L

  Then:

    Replace B with a same-family induction variable M in the conditional

    Remove B's recursive assignment in the loop

```
(1)   I:=1                        B1

(3)   T1=2*J                      B2'
(6)   T4=addr(A)-11
(7)   T5=2*J
(10)  T8=addr(A)-11

(2)  if I>10 goto (15)            B2

(4)   T2=10*I                     B3
(5)   T3= T2+ T1
(8)   T6=10*I
(9)   T7= T6+ T5
(11)  T9= T8[T7]
(12)  T4[T3]= T9+1
(13)  I:=I+1
(14)  goto B2

(15)
```

```
(1)   I:=1                        B1

(3)   T1=2*J                      B2'
(6)   T4=addr(A)-11
(7)   T5=2*J
(10)  T8=addr(A)-11
(4)   T2=10*I
(8)   T6=10*I
(5)   T3= T2+ T1
(9)   T7= T6+ T5

(2)   if I>10 goto (15)           B2

(11)  T9= T8[T7]                  B3
(12)  T4[T3]= T9+1
(13)  I:=I+1
(4')  T2= T2 +10
(8')  T6= T6 +10
(5')  T3= T3+ 10
(9')  T7= T7+ 10
(14)  goto B2

(15)
```

```
(1)   I:=1                        B1

(3)   T1=2*J                      B2'
(6)   T4=addr(A)-11
(7)   T5=2*J
(10)  T8=addr(A)-11
(4)   T2=10*I
(8)   T6=10*I
(5)   T3= T2+ T1
(9)   T7= T6+ T5
(21)  R=100+T1
                                  T3=10*I+T1
(2)  if T3>R goto (15)            B2

(11)  T9= T8[T7]                  B3
(12)  T4[T3]= T9+1
(5')  T3= T3+ 10
(9')  T7= T7+ 10
(14)  goto B2

(15)
```

Conclusion

- Strength reduction targets assignments of ***induction*** variables linearly related to ***basic induction*** variables.
- New ***dead assignments*** may appear after strength reduction and can be removed.
- ***Basic induction*** variables can also be eliminated.
- Very effective for reducing subscript/address calculation costs.

（for I:=1 to 10 do A[I, 2*J] := A[I, 2*J] + 1）

Quiz-Canvas

- **ch10 Optimization – Loop Optimization**

Dank u
Dutch

Merci
French

Спасибо
Russian

Gracias
Spanish

شكراً
Arabic

धन्यवाद
Hindi

감사합니다
Korean

Tack så mycket
Swedish

תודה רבה
Hebrew

Obrigado
Brazilian Portuguese

谢谢
Chinese

Dankon
Esperanto

Thank You !

ありがとうございます
Japanese

Trugarez
Breton

Tak
Danish

Danke
German

Grazie
Italian

நன்றி
Tamil

děkuji
Czech

ขอบคุณ
Thai

go raibh maith agat
Gaelic