Compiler Optimizations

Dragon Book, Chapter 9, Section 9.1.7

Local vs. Global Optimizations

- Local: inside a single basic block
- Global: **intra**procedurally, across basic blocks
- In all of these techniques, we often use the results of control-flow analysis and data-flow analysis
- Our objective: discuss a few examples of optimizations, without details

Local Common Subexpression Elimination

a = b + c
b = a – d
c = b + c
d = a – d

a – d is a common subexpression: when evaluated the second time, it will produce the same value

Optimized code:

a = b + c	**a = b + c**
b = a – d	**d = a – d**
c = b + c	**c = d + c**
d = b	
if b is live	*if b is not live*

Question: can we eliminate **b + c** with this approach?

Algebraic Identities

- Arithmetic

  $x + 0 = 0 + x = x$ $x * 1 = 1 * x = x$

  $x - 0 = x$ $x / 1 = x$

- Strength reduction: replace a more expensive operator with a cheaper one

  $2 * x = x + x$ $x / 2 = 0.5 * x$

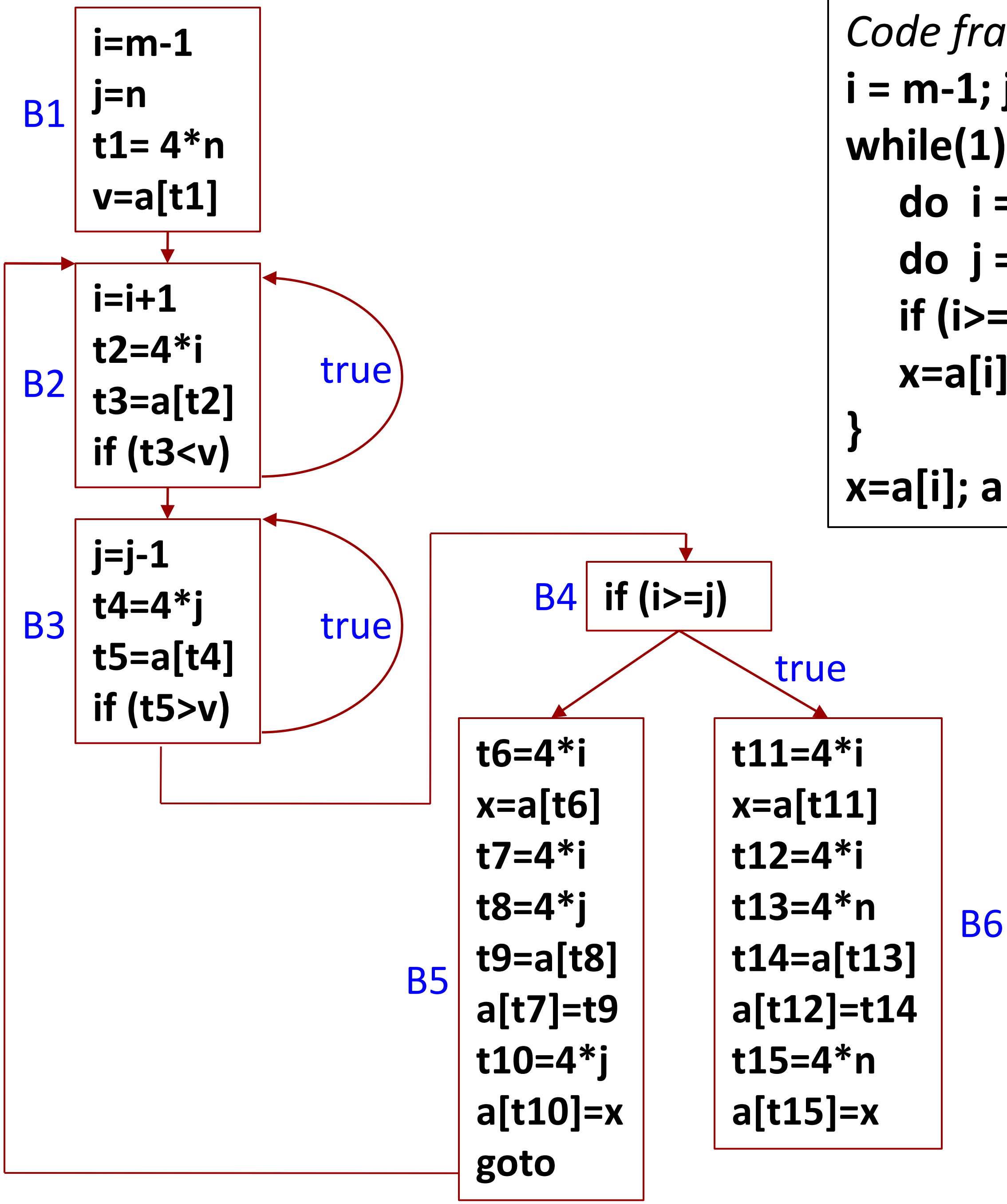
- Constant folding: evaluate expressions at compile time and use the result

  x = 2*3.14 is replaced with x = 6.28

  needed with symbolic constants (#define in C, final vars/fields in Java) or after constant propagation

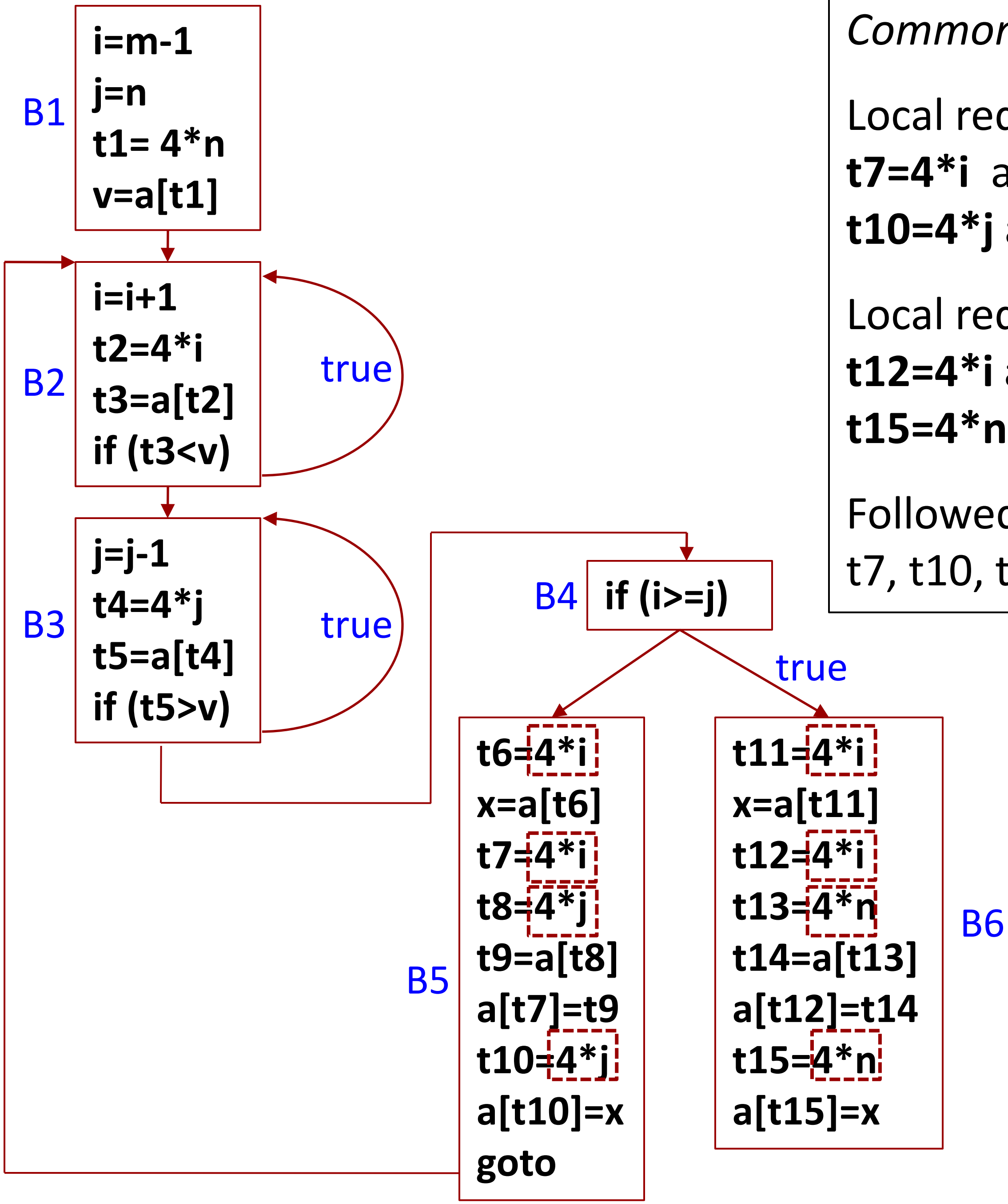
Other Global Optimizations [you are not responsible for this material]

- Code motion for loop-invariant computations
- Common subexpression elimination
- Copy propagation
- Dead code elimination
- Elimination of induction variables
  – Variables that essentially count the number of iterations around a loop
- Partial redundancy elimination
  – Powerful generalization of code motion and common subexpression elimination (Dragon book, Section 9.5)



Code fragment from quicksort:

```
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;
```

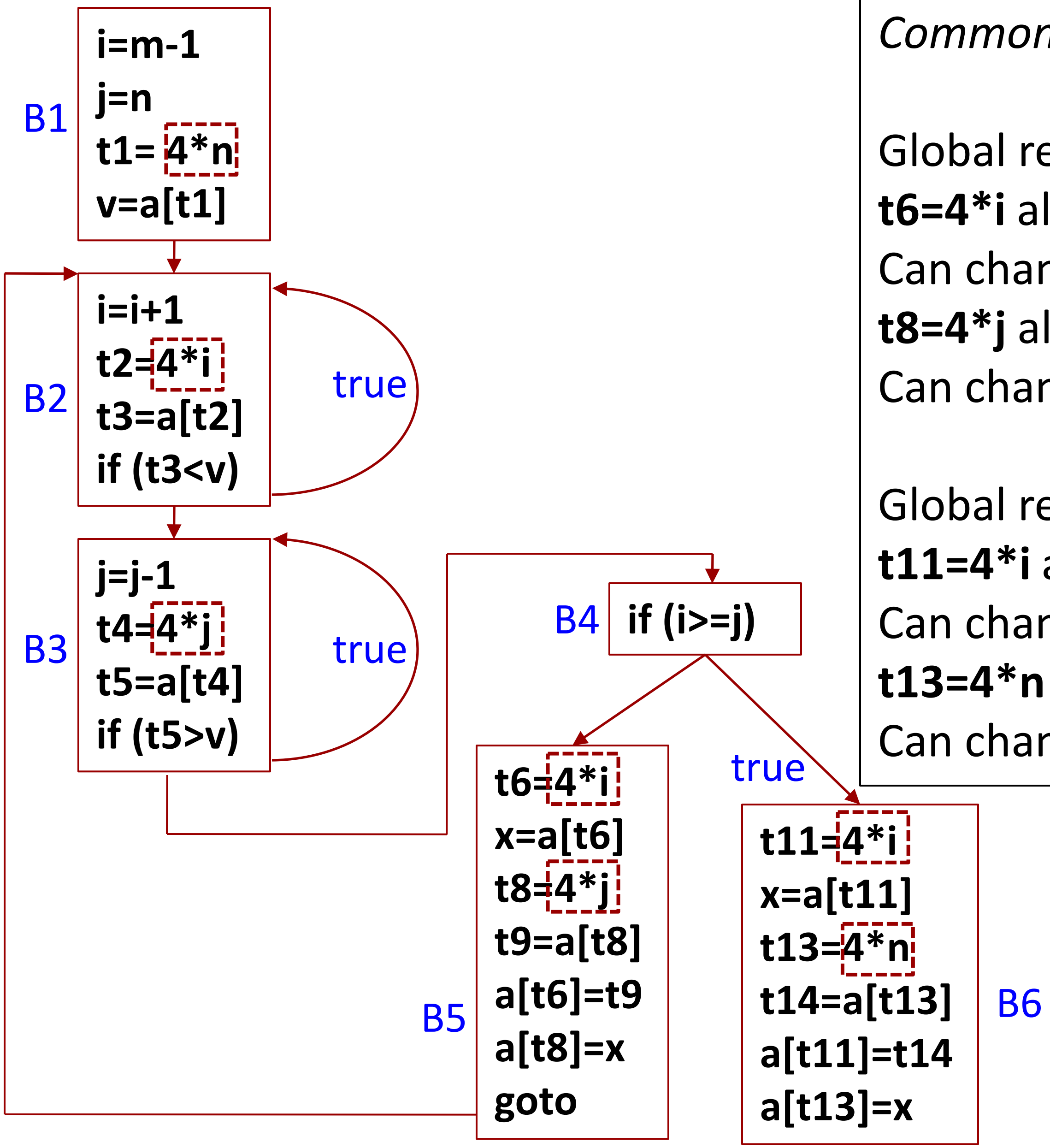


Common subexpression elimination

Local redundancy in B5:
t7=4*i already available in **t6**
t10=4*j already available in **t8**

Local redundancy in B6:
t12=4*i already available in **t11**
t15=4*n already available in **t13**

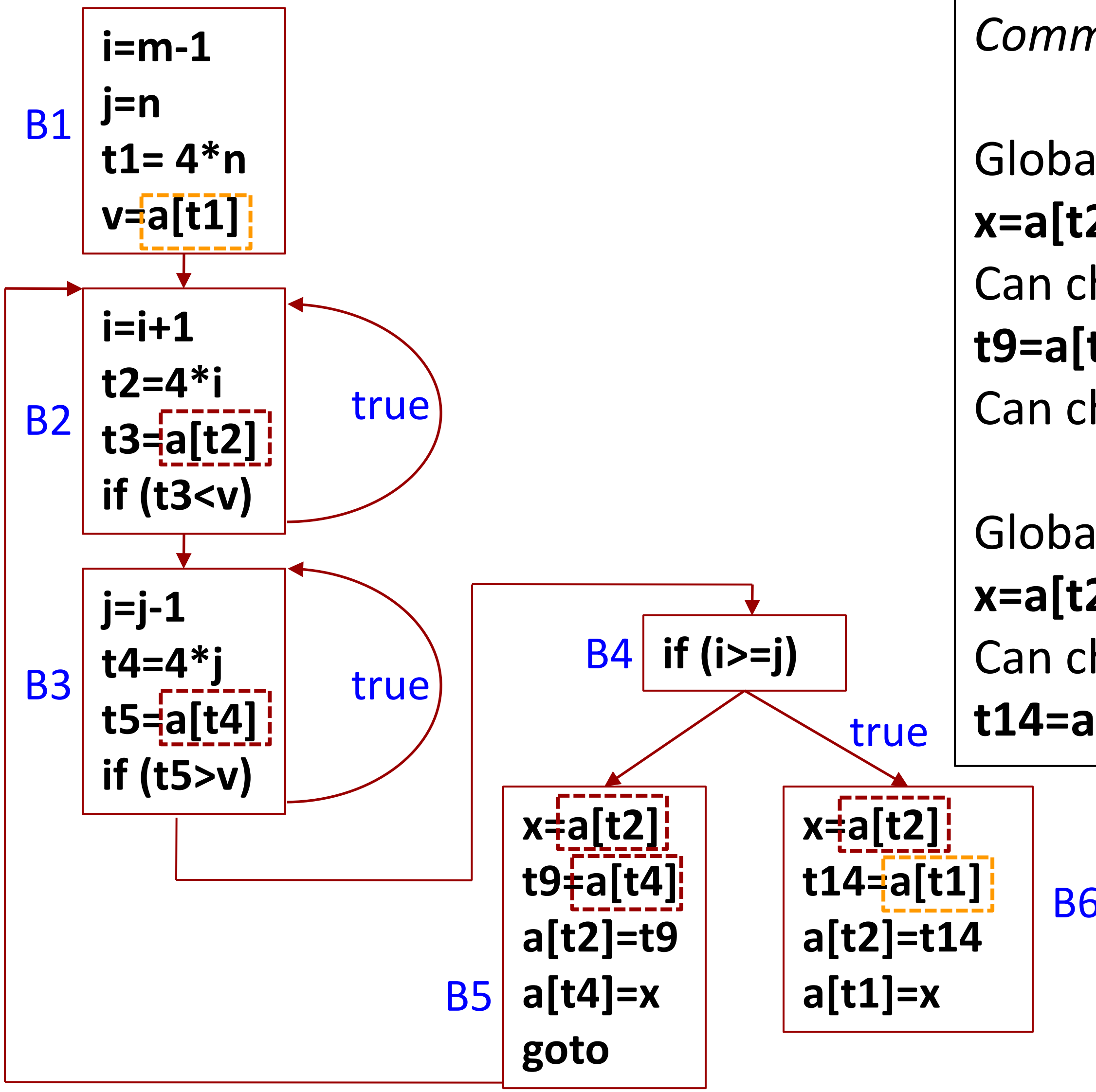
Followed by *dead code elimination* for t7, t10, t12, t15



Common subexpression elimination

Global redundancy in B5:
t6=4*i already available in **t2**
Can change **x=a[t6]** and **a[t6]=t9**
t8=4*j already available in **t4**
Can change **t9=a[t8]** and **a[t8]=x**

Global redundancy in B6:
t11=4*i already available in **t2**
Can change **x=a[t11]** and **a[t11]=x**
t13=4*n already available in **t1**
Can change **t14=a[t13]** and **a[t13]=x**



Common subexpression elimination

Global redundancy in B5:

x=a[t2] already available in **t3**

Can change **x=a[t2]**

t9=a[t4] already available in **t5**

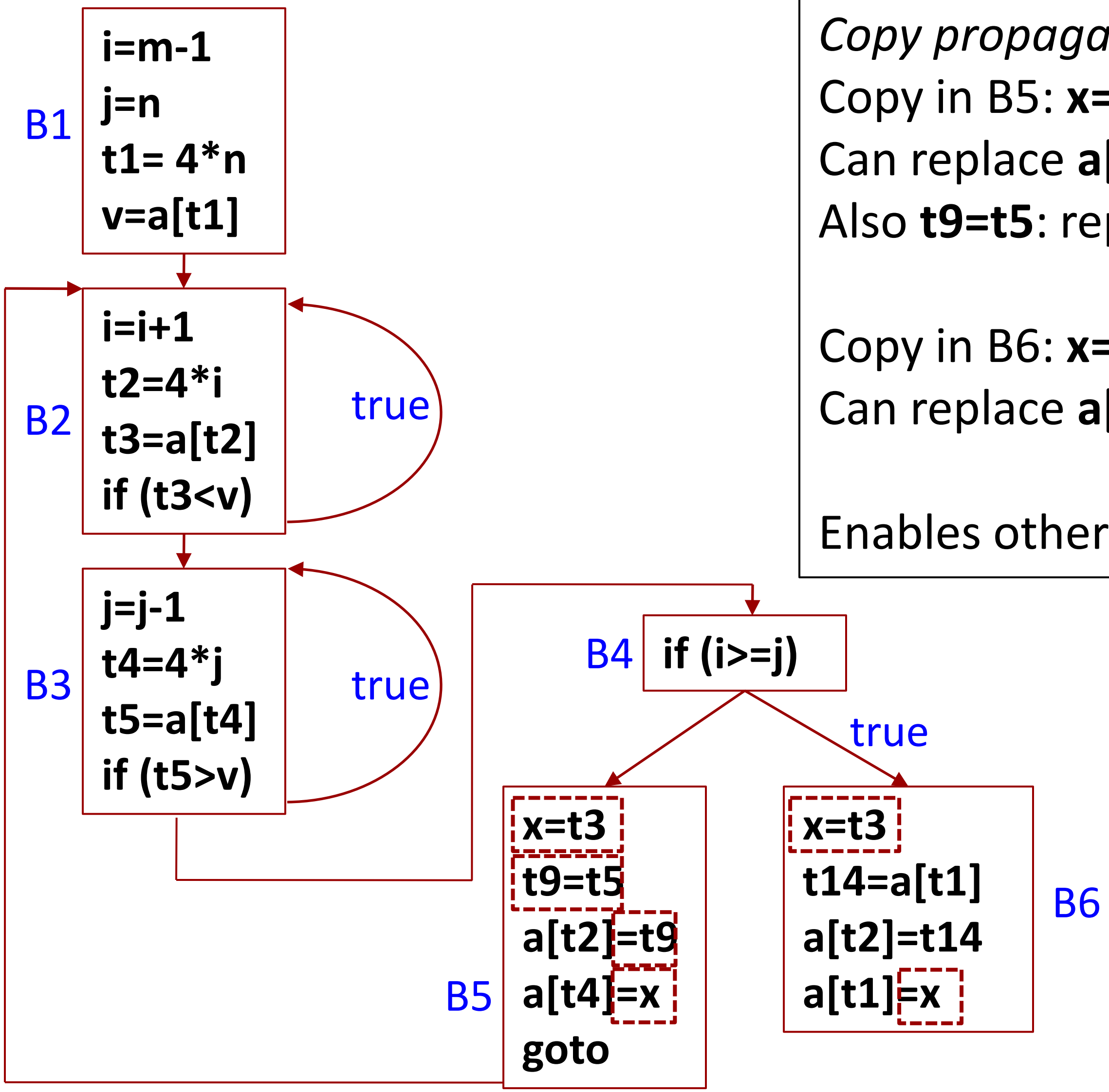
Can change **t9=a[t4]**

Global redundancy in B6:

x=a[t2] already available in **t3**

Can change **x=a[t2]**

t14=a[t1] **not** available in **v**. **Why?**



Copy propagation

Copy in B5: **x=t3**

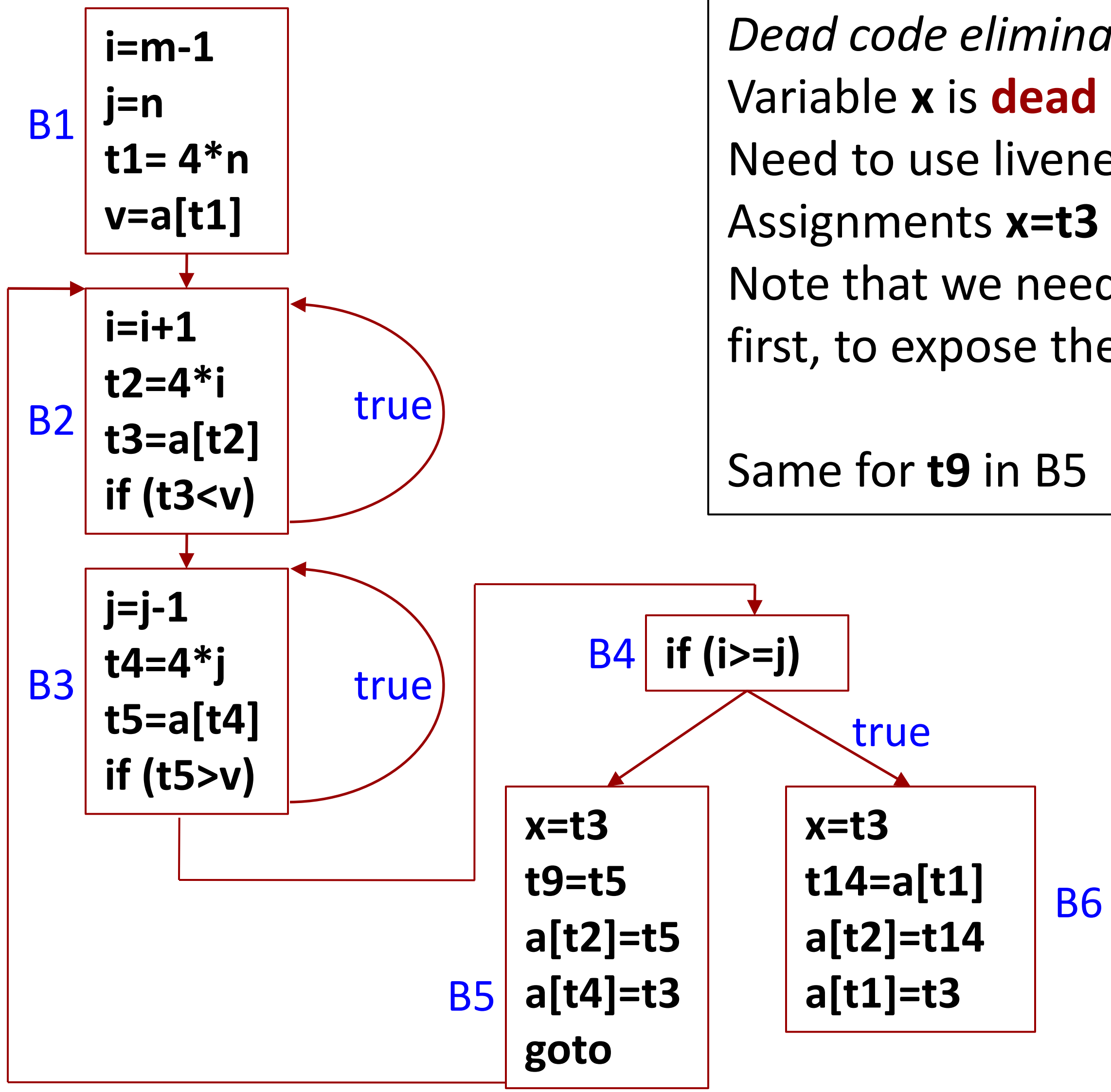
Can replace **a[t4]=x** with **a[t4]=t3**

Also **t9=t5**: replace **a[t2]=t9** with **a[t2]=t5**

Copy in B6: **x=t3**

Can replace **a[t1]=x** with **a[t1]=t3**

Enables other optimizations



Dead code elimination

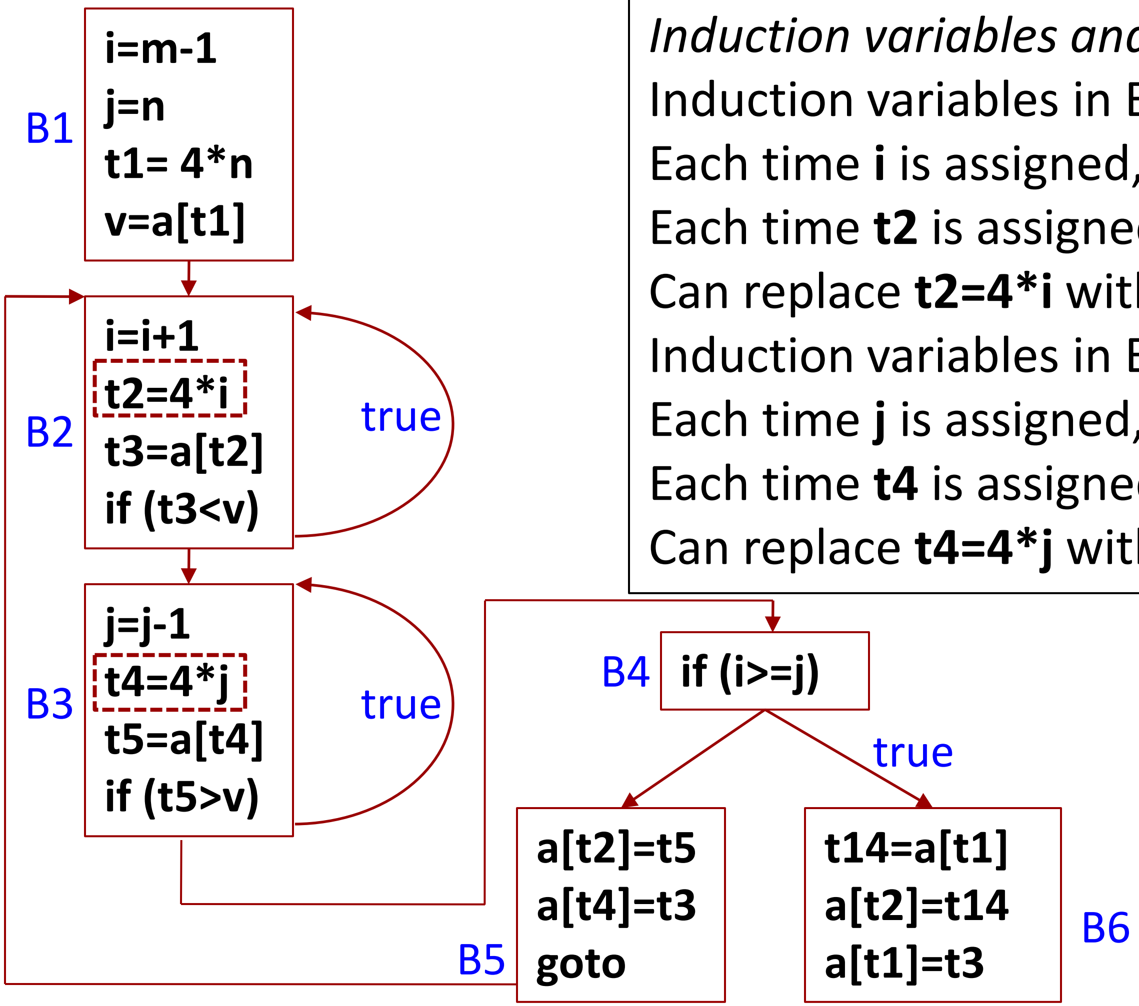
Variable **x** is **dead** immediately after B5 and B6

Need to use liveness analysis for this.

Assignments **x=t3** in B5 and B6 are dead code

Note that we needed to do copy propagation first, to expose the “deadness” of x.

Same for **t9** in B5



Induction variables and strength reduction

Induction variables in B2:

Each time **i** is assigned, its value increases by 1

Each time **t2** is assigned, its value increases by 4

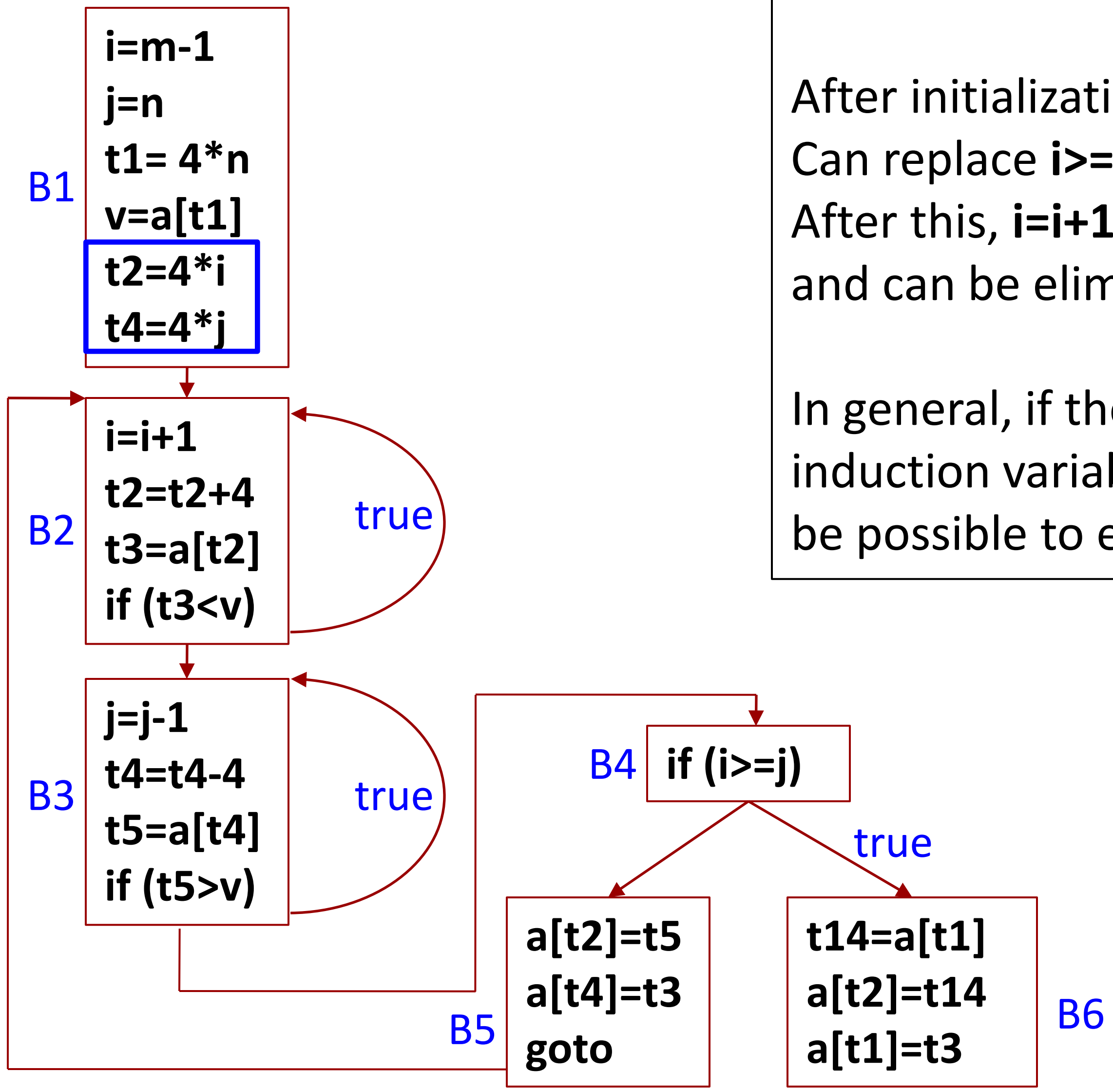
Can replace **t2=4*i** with **t2=t2+4**

Induction variables in B2:

Each time **j** is assigned, its value decreases by 1

Each time **t4** is assigned, its value decreases by 4

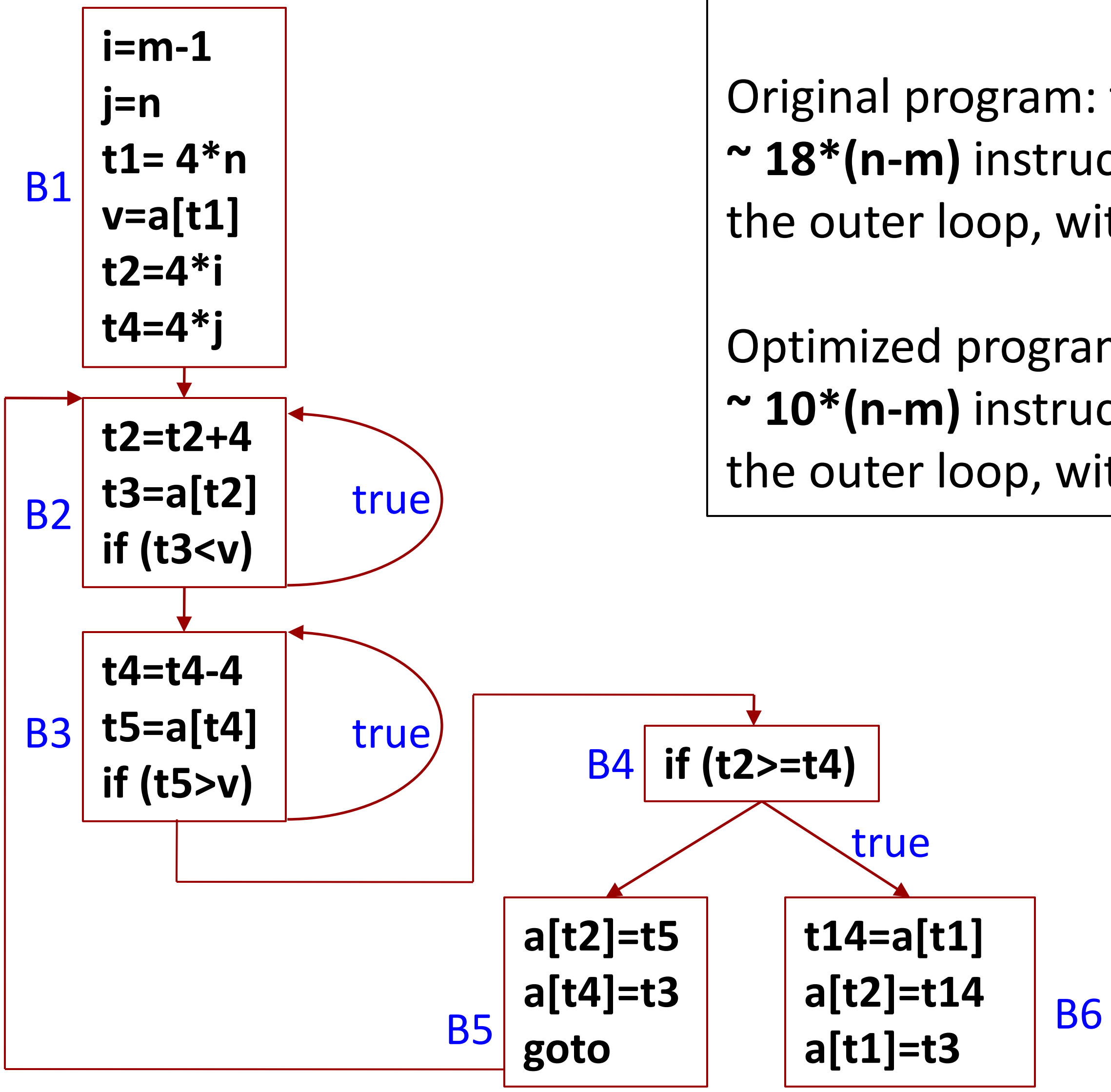
Can replace **t4=4*j** with **t4=t4-4**



Elimination of induction variables

After initialization, **i** and **j** are used only in B4
Can replace **i>=j** with **t2>=t4** in B4
After this, **i=i+1** and **j=j-1** become dead code and can be eliminated

In general, if there are two or more induction variables in the same loop, it may be possible to eliminate all but one of them



Final program after all optimizations

Original program: for the worst-case input, **~ 18*(n-m)** instructions would be executed in the outer loop, with **~ 6*(n-m)** multiplications

Optimized program: for the worst-case input, **~ 10*(n-m)** instructions would be executed in the outer loop, without any multiplications