

# Compiler Optimizations

---

Chapter 8, Section 8.5

Chapter 9, Section 9.1.7

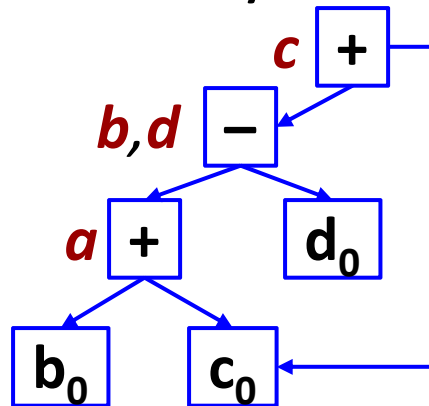
# Local vs. Global Optimizations

- Local: inside a single basic block
  - Simple forms of **common subexpression elimination**, **dead code elimination**, **strength reduction**, etc.
  - May sometimes require the results of a dataflow analysis (e.g., Live Variables analysis)
- Global: intraprocedurally, across basic blocks
  - Code motion, more complex common subexpression elimination, etc.
- Interprocedural optimizations: across procedure boundaries
  - Tricky; not used often
  - Sometimes we do procedure inlining and use intraprocedural optimizations

# Part 1: Local Optimizations

- Based on a DAG representation of a basic block
- DAG nodes are
  - The initial values of variables (before the basic block)
  - Nodes labeled with operations – e.g. +, −, etc.
- DAG construction
  - Traverse the sequence of instructions in the basic block
  - Maintain a mapping: (variable  $v$ , DAG node that corresponds to the last write to this variable)
  - Construct new nodes only when necessary
  - Example

$a = b + c$   
 $b = a - d$   
 $c = b + c$   
 $d = a - d$



# Common Subexpression Elimination

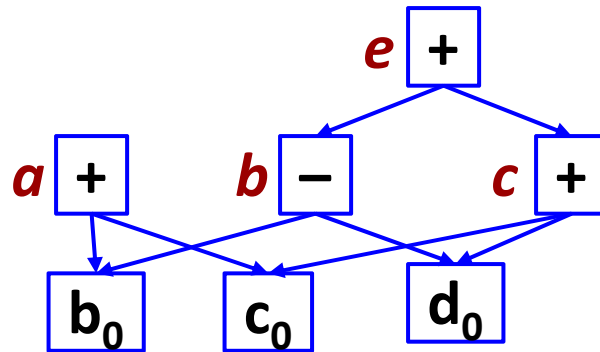
- In the previous example, we identified the common subexpression **a - d**
  - This can be used to optimize the code – eventually we will reassemble the basic block from this DAG, and the result will be more efficient code (see Section 8.5.7)
- This does not always work

$$a = b + c$$

$$b = b - d$$

$$c = c + d$$

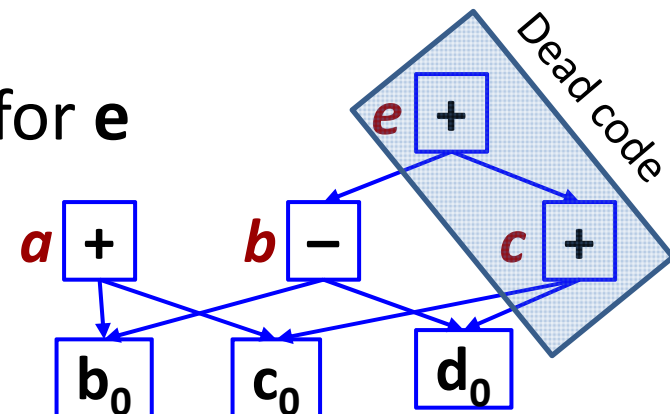
$$e = b + c$$



There are no common subexpressions here, but in fact **a** and **e** both have the value **b<sub>0</sub> + c<sub>0</sub>**

# Dead Code Elimination

- Assumes the output of Live Variables analysis (aka **liveness analysis**)
- Consider the DAG together with the map of DAG node that have the last writes of variables
- If a DAG node has zero incoming edges, and has zero writes of **live** variables, it can be eliminated
  - And this may enable other removals
  - Example (from before): suppose that **e** and **c** are not live at the exit of the basic block
    - First we can remove the node for **e**
    - Then, the node for **c**



# Algebraic Identities

- Arithmetic

$$x + 0 = 0 + x = x$$

$$x - 0 = x$$

$$x * 1 = 1 * x = x$$

$$x / 1 = x$$

- **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  $x = 6.28$  (this often happens due to the use of symbolic constants with `#define`, etc.)

- Commutativity and associativity

- E.g.  $a = x * y$  and  $b = y * x$ : are these common subexpr?
- Need to be careful: e.g. underflows, overflows, etc.

# Local vs. Global Optimizations

- Local: inside a single basic block
  - Simple forms of common subexpression elimination, dead code elimination, strength reduction, etc.
  - May sometimes require the results of a dataflow analysis (e.g., Live Variables analysis)
- Global: intraprocedurally, across basic blocks
  - **Code motion**, more complex common subexpression elimination, etc.
- Interprocedural optimizations: across procedure boundaries
  - Tricky; not used often
  - Sometimes we do procedure inlining and use intraprocedural optimizations

## Part 2: Loop-Invariant Computations

- Motivation: avoid redundancy

a = ...

b = ...

c = ...

*start loop*

...

**d = a + b**

**e = c + d**

...

*end loop*

All instructions whose right-hand side operands have reaching definitions that are **only** from outside the loop

But, this also applies transitively ...

# Complete Definition and Algorithm

- Add an instruction to *Inv* if each operand on the right-hand side satisfies one of the following
  - It is a constant: e.g., -5 or 3.1415
  - All its reaching definitions are from outside of the loop
- After this initialization, repeat until no further changes to *Inv* are possible
  - Find an instruction that is not in *Inv* but each operand on the right-hand side satisfies one of the following
    - It is a constant
    - All its reaching definitions from outside of the loop
    - It has exactly one reaching definition, and that definition is already in *Inv*
  - Add this instruction to *Inv*
  - Note: use the ud-chains to find all reaching defs

# The Last Condition

- A right-hand side operand has exactly one reaching def, and that def is already in *Inv*
- Why not **two** reaching defs, both already in *Inv*?
  - if (...) a = 5; else a = 6;**  
**b = a+1;**
    - Even though each definition of **a** is in *Inv*, the value of **b** is no guaranteed to be the same for each iteration
- But even this is not enough: not every loop-invariant computation (i.e., member of *Inv*) can actually be moved out of the loop
  - **for (...) { a = 1; ... if (...) break; ... b = a+1; ... }**
  - **a = 1; b = a+1; for (...) { ... if (...) break; ... }**

# Conditions for the Actual Code Motion

- Given: a loop-invariant instruction from *Inv*
- Condition 1: the basic block that contains the instruction must dominate all exits of the loop
  - i.e., all nodes that are sources of loop-exit edges
  - This means that it is impossible to exit the loop before the instruction is executed
    - And thus, it is safe to move this instruction before the loop header
- Condition 2: to move instruction **a = ...**, this must be the only assignment to **a** in the loop
  - **for (...)** { **a = 1;** ... **if (...)** { ... **a = 2;** ... } ... **b = a+1;** ... }
  - **a = 1;** **for (...)** { ... **if (...)** { ... **a = 2;** ... } ... **b = a+1;** ... }

# Conditions for the Actual Code Motion

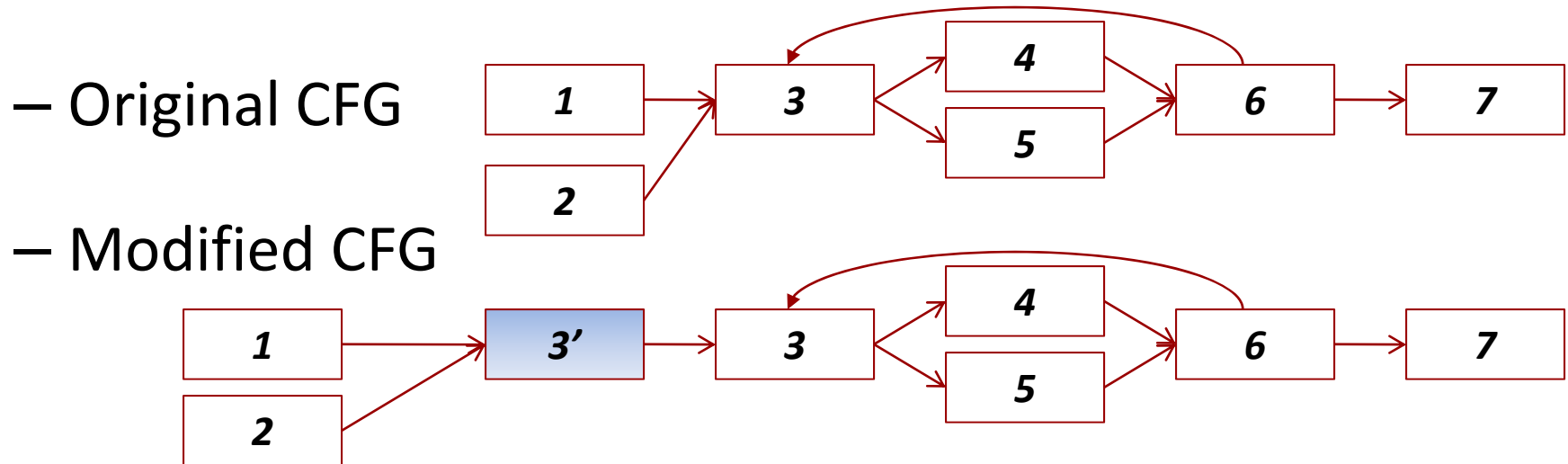
- Condition 3: to move instruction **a = ...**, every use of **a** in the loop must be reached **only** by this definition of **a**
  - Condition 2 ensures that there does not exist another definition of **a** in the loop; Condition 3 guarantees that even if there are definitions of **a** outside of the loop, they do not reach any use of **a** in the loop
    - **a = 1; for (...)** { ... **b = a+1; ... a = 2; ... }**
    - **a = 1; a = 2; for (...)** { ... **b = a+1; ... }**
- Given the subset of instructions from *Inv* that satisfy the three conditions from above, we can now modify the three-address code

# Complications with Arrays

- Reaching definitions and ud-chains for arrays:
  - Any  $x[y] = z$  is treated as a “definition” of the entire array  $x$  (really, a **potential** def for each array element)
  - Any  $x = y[z]$  is treated as a “use” of the entire array  $y$  (really, a **potential** use for each array element)
  - This is very approximate ... (better approaches exist)
- Various issues
  - $a[x] = y$
  - $a[z] = w$
  - $u = a[v]$
  - Both definitions of  $a$  are reaching for the use of  $a$ 
    - Unless we can prove that  $z$  is always equal to  $x$
  - To simplify our life for code motion, in this course we will never move instructions  $a[x] = y$  and  $x = a[y]$

# Code Transformation

- First, create a **preheader** for the loop



- Next, consider all instructions in *Inv*, in the order in which they were added to *Inv*
  - Each instruction that satisfies the three conditions is added at the end of the preheader, and removed from its basic block (basically, it is moved to a different position in the sequence of three-address instructions)

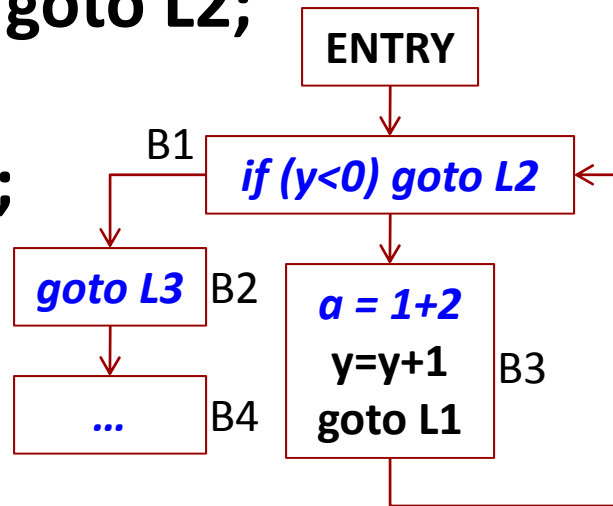
# Alternative Approach

- Move the invariant expressions to the preheader and assign them to temporaries
  - Easier to deal with correctness
- Condition 1 from before
  - **for (...)** { **a = 1;** **if (...)** **break;** **b = a+1;** }
  - **t1 = 1;** **t2 = t1+1;** **for (...)** { **a = t1;** **if (...)** **break;** **b = t2;** }
- Condition 2 from before
  - **for (...)** { **a = 1;** ... **if (...)** { ... **a = 2;** } ... **b = a+1;** }
  - **t1 = 1;** **for (...)** { **a = t1;** ... **if (...)** { ... **a = 2;** } ... **b = a+1;** }
- Condition 3 from before
  - **a = 1;** **for (...)** { ... **b = a+1;** ... **a = 2;** ... }
  - **a = 1;** **t1 = 2;** **for (...)** { ... **b = a+1;** ... **a = t1;** ... }

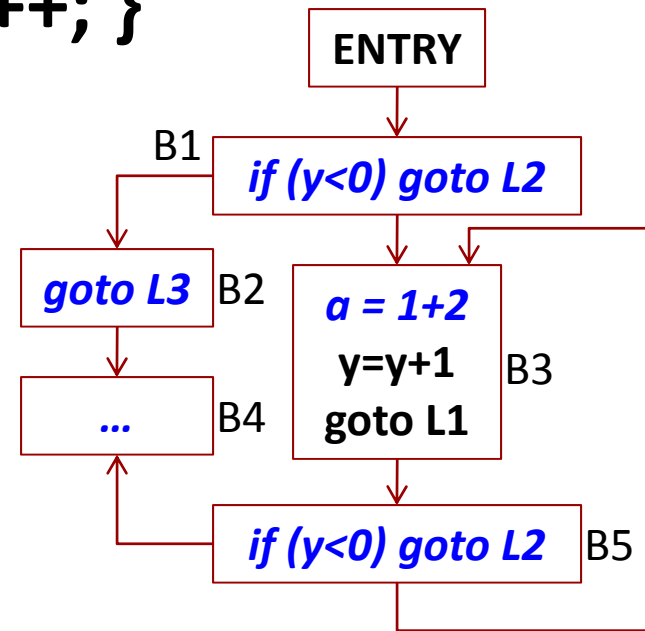
# Potentially Necessary Pre-Transformation

- Recall that all exits of a loop should be dominated by the instruction we want to move (Condition 1)
- Consider **while(y<0) { a = 1+2; y++; }**

**L1: if (y<0) goto L2;**  
**goto L3;**  
**L2: a = 1+2;**  
**y = y + 1;**  
**goto L1;**  
**L3: ...**



**a = 1+2** does **not** dominate the exit node B1



loop header is now B3 and **a = 1+2** dominates the exit node B5

# Old SDD for Translation

- $S \rightarrow \text{while } (B) S_1$ 
  - $begin = \text{newLabel}()$
  - $B.true = \text{newLabel}()$
  - $B.false = S.next$
  - $S_1.next = begin$
  - $S.code = \text{label}(begin) \ || \ B.code \ || \ \text{label}(B.true) \ || \ S_1.code \ || \ \text{gen}(\text{"goto" } begin)$
  - Example: **while (x < 0) y = 1;**

*begin* **L2: if (x < 0) goto L3;**

**goto L1;**

*B.true* **L3: y = 1;**

**goto L2;**

**L1: noop;**

# New SDD for Translation

- $S \rightarrow \text{while } (B) S_1$ 
  - $B.true = \text{newLabel}()$
  - $B.false = S.next$
  - $S_1.next = B.true$
  - $S.code = B.code \ || \ \text{label}(B.true) \ || \ S_1.code \ || \ B.code$
  - Example: **while (x < 0) y = 1;**

**if (x < 0) goto L2;**

**goto L1;**

$B.true$  **L2; y = 1;**

**if (x < 0) goto L2;**

**goto L1;**

$B.false$  **L1; noop;**

Note: also need to do something similar for **for** loops

## Part 3: Other Global Optimizations

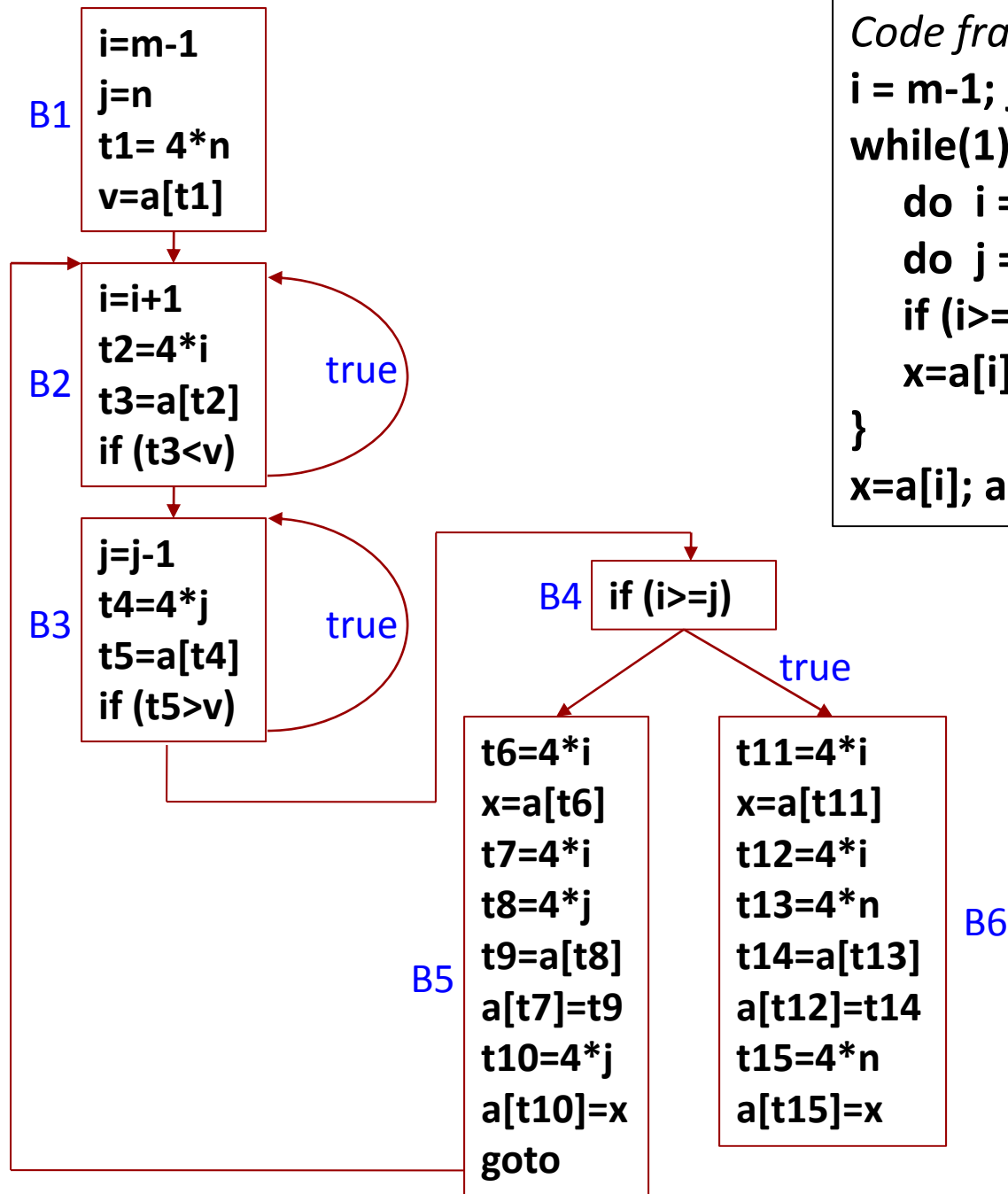
- We have already have seen one simple form of code motion for loop-invariant computations
- Common subexpression elimination
- Copy propagation
- Dead code elimination
- Elimination of induction variables
  - Variables that essentially count the number of iterations around a loop
- Partial redundancy elimination
  - Powerful generalization of code motion and common subexpression elimination (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:

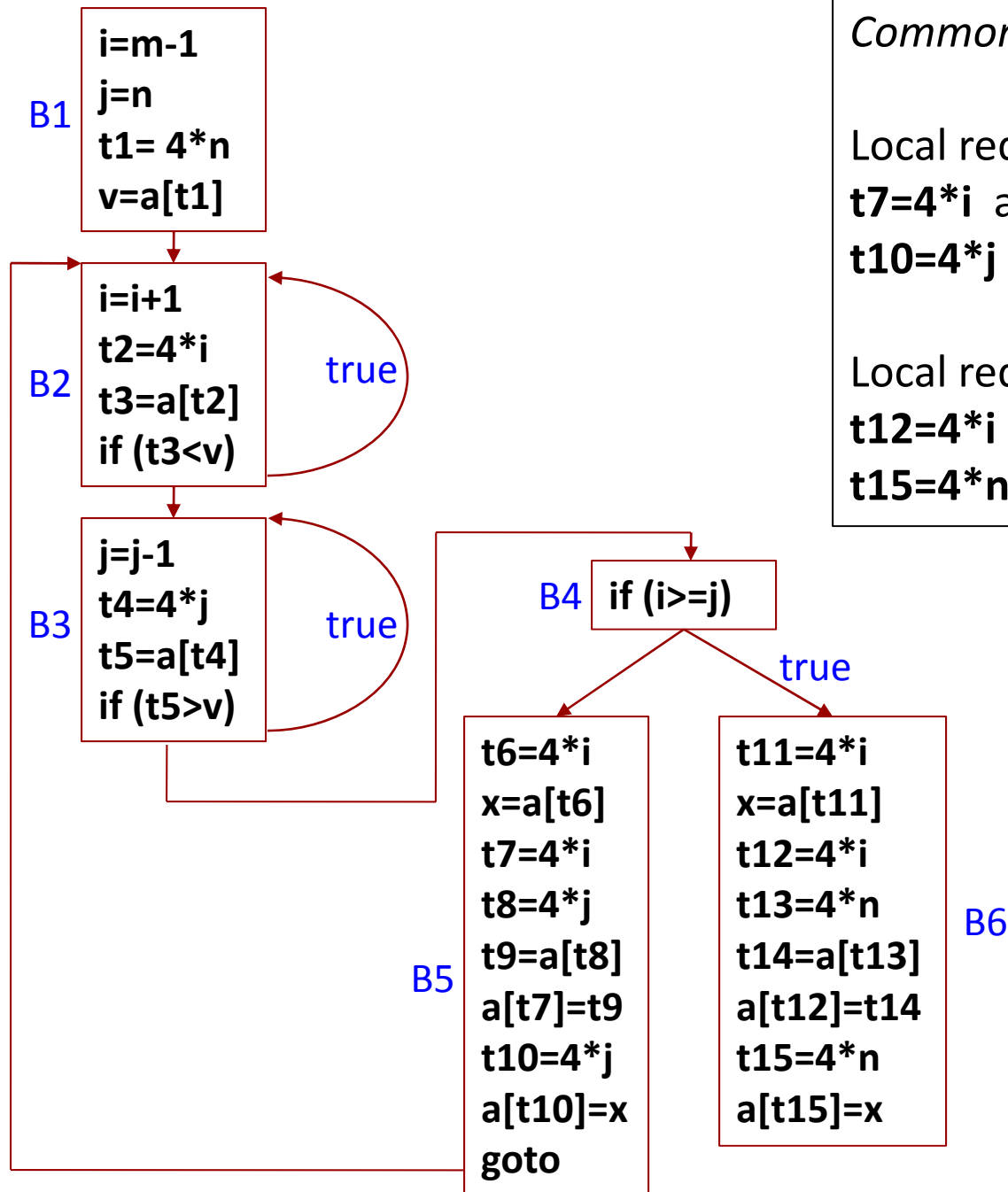
**t7=4\*i** already available in **t6**

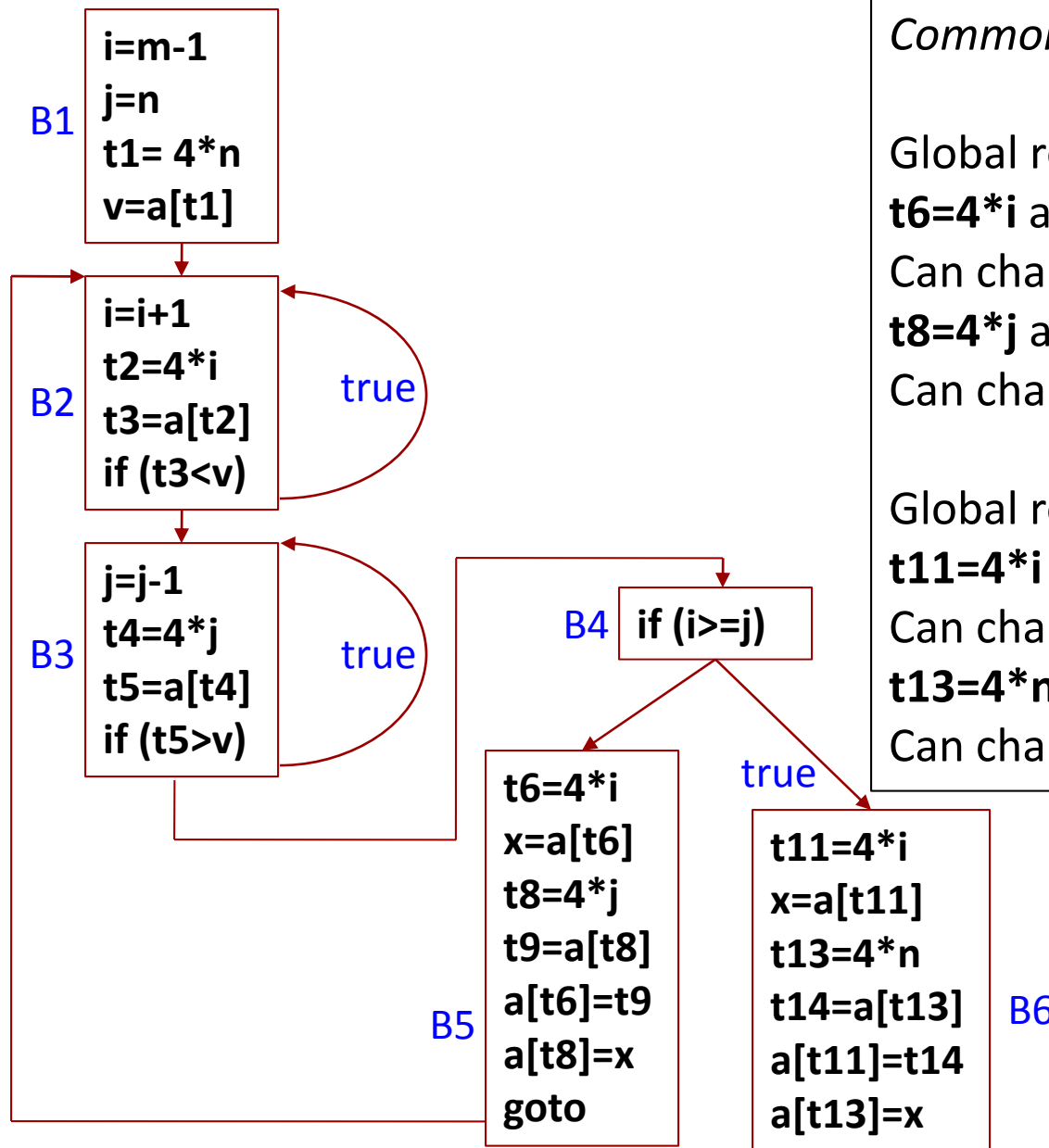
**t10=4\*j** already available in **t8**

Local redundancy in B6:

**t12=4\*i** already available in **t11**

**t15=4\*n** already available in **t13**





### 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**

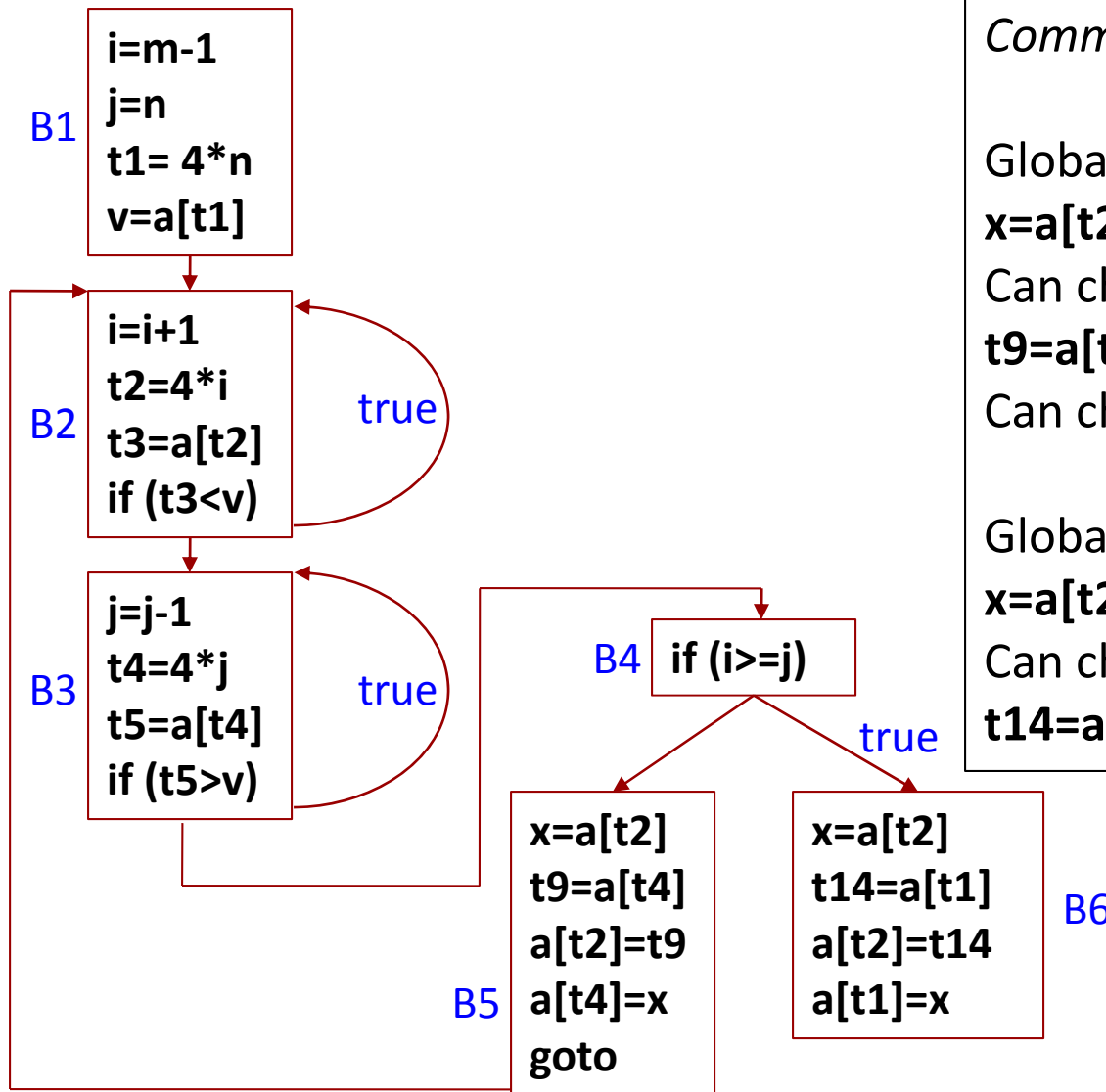
Global redundancy in B6:

**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**

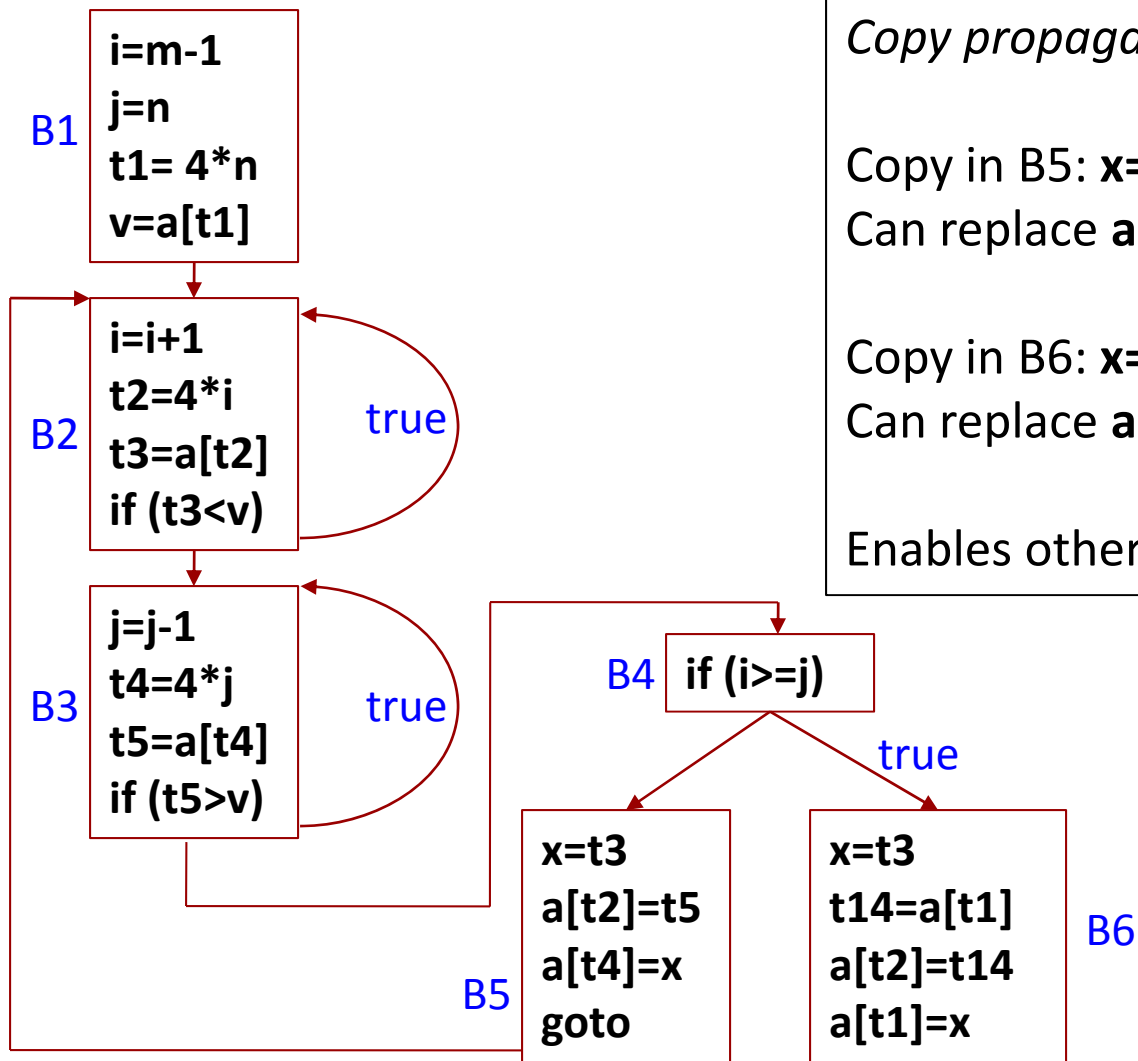
Can change  $a[t2]=t9$

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**

Copy in B6: **x=t3**

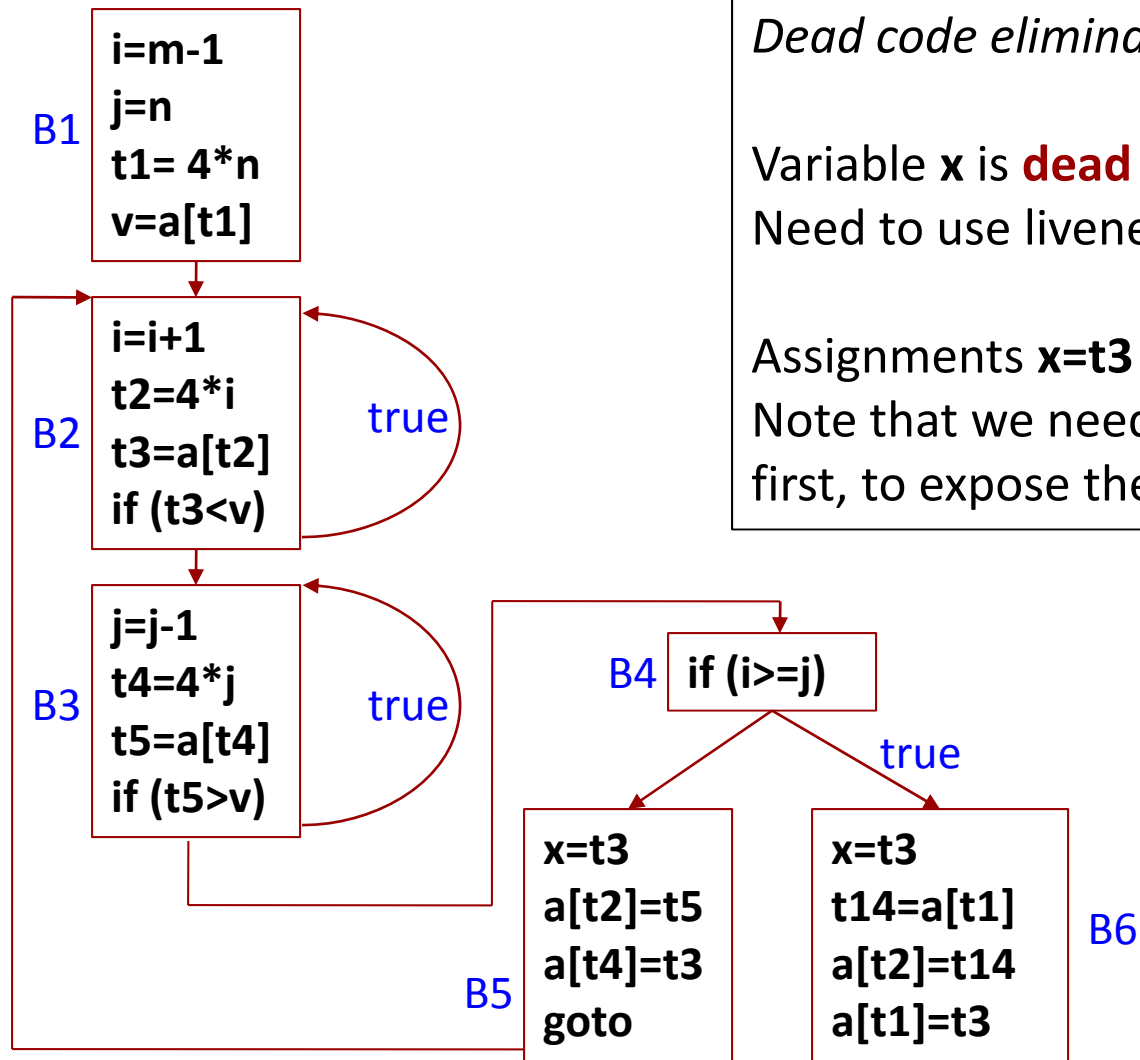
Can replace **a[t1]=x** with **a[t4]=t3**

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.



## 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

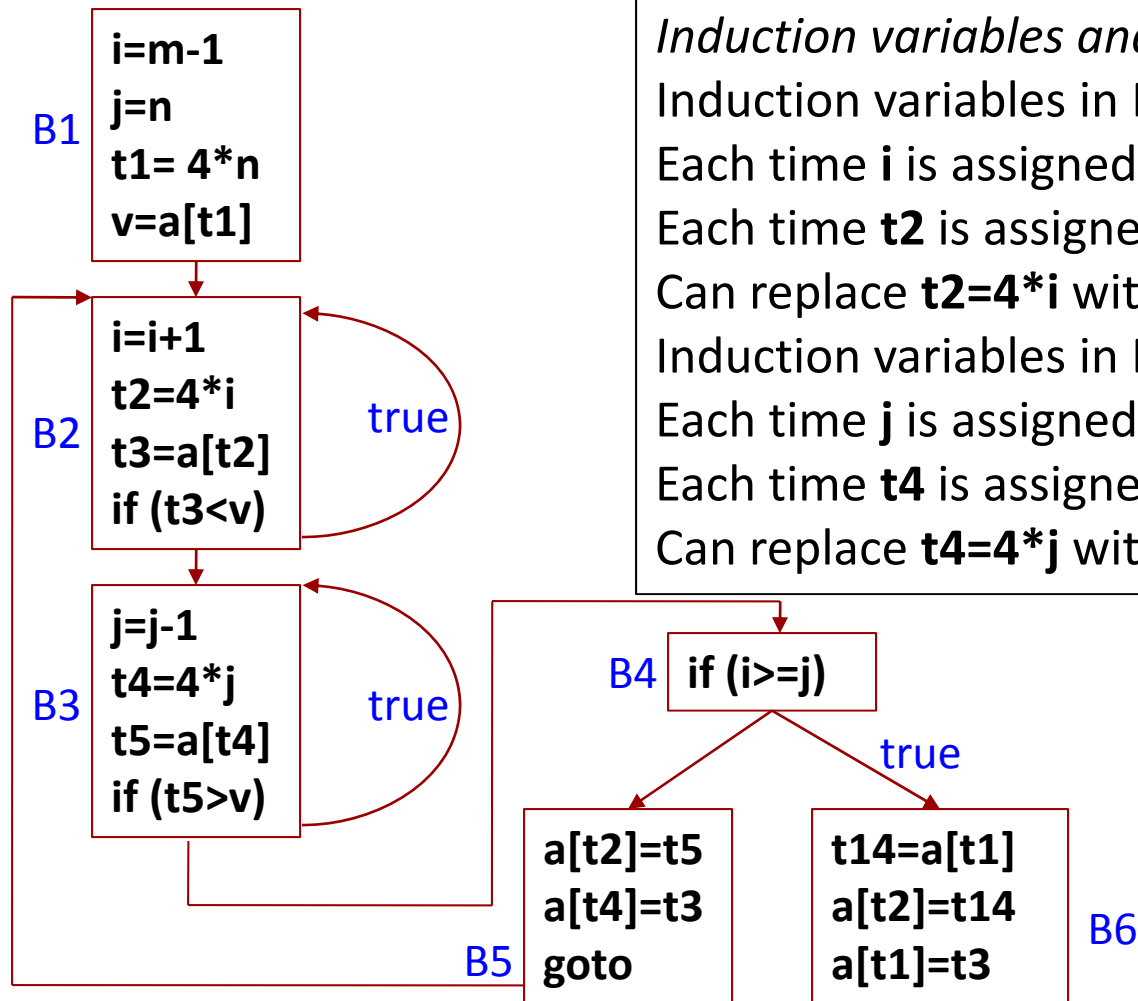
Can replace  $t2=4*i$  with  $t2=t2+4$

Induction variables in B3:

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

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

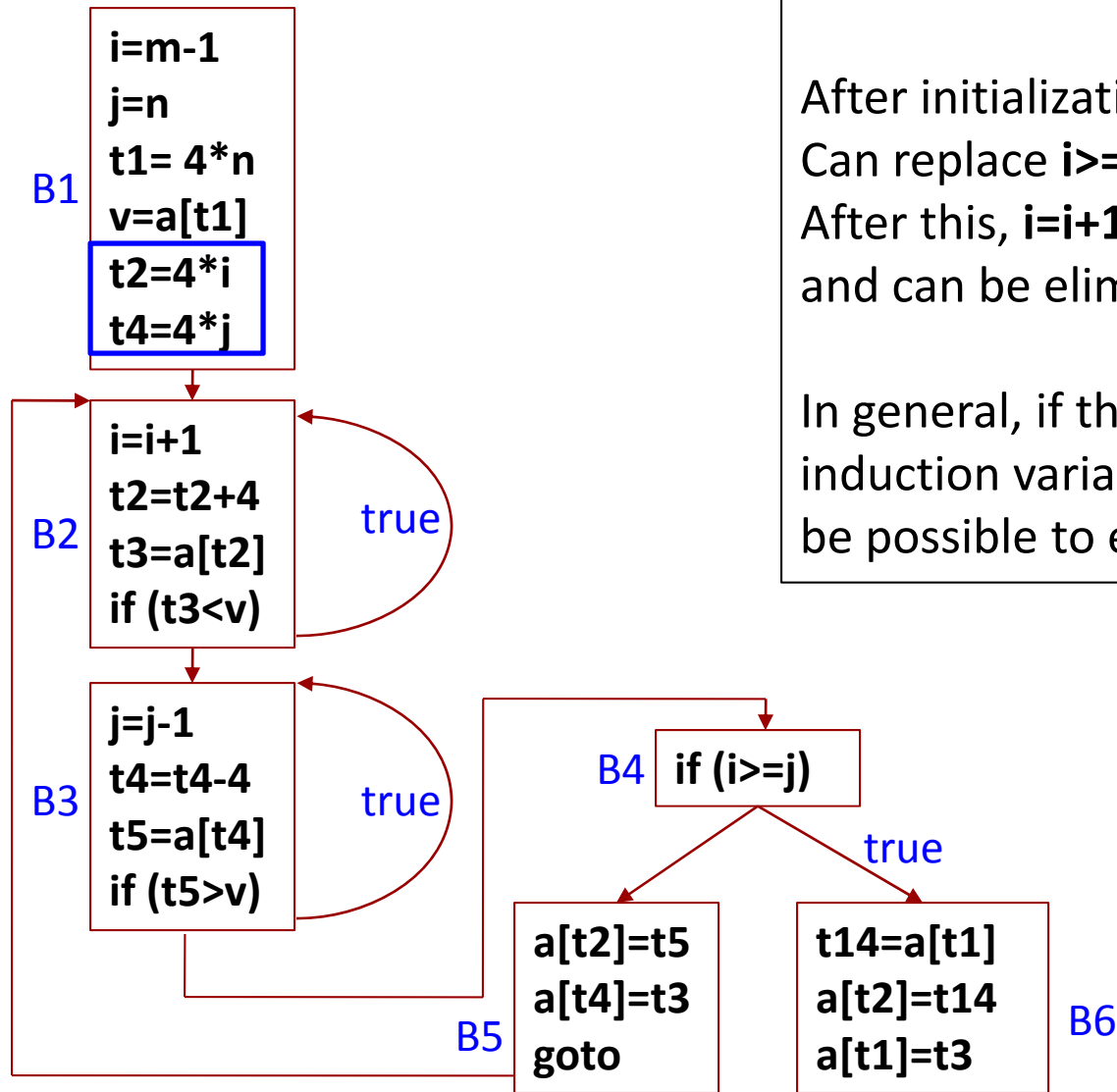
Can replace  $t4=4*j$  with  $t4=t4-4$

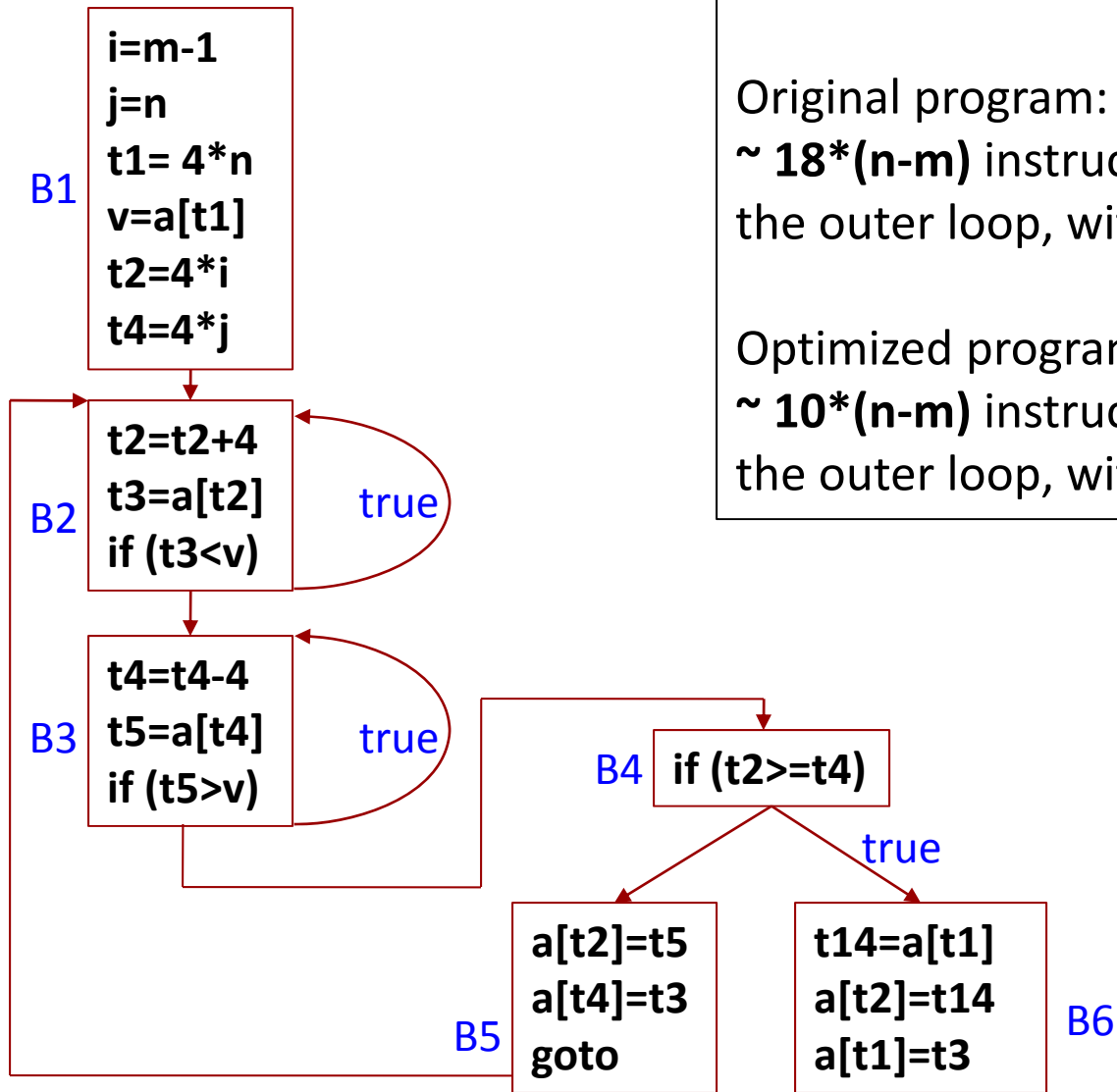


## Elimination of induction variables

After initialization,  $i$  and  $j$  are used only in B4  
Can replace  $i \geq j$  with  $t2 \geq t4$  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





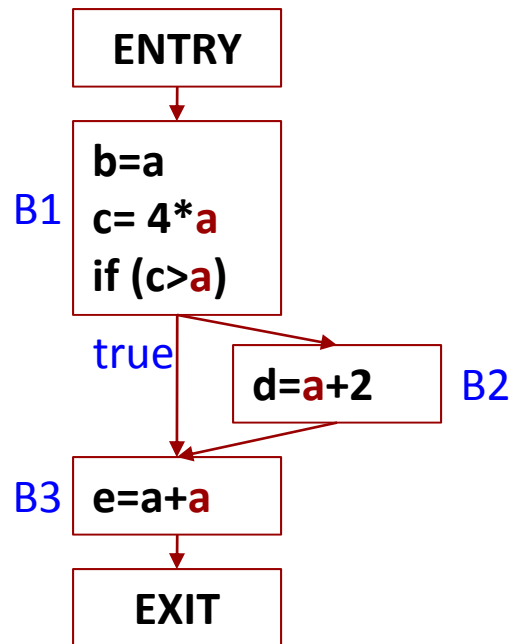
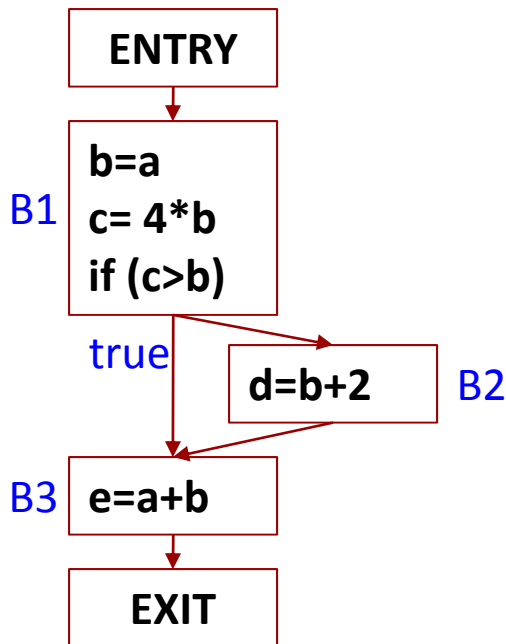
*Final program after all optimizations*

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

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

# (start detour) Another Dataflow Analysis

- **Copy propagation**: for  $x = y$ , replace subsequent uses of  $x$  with  $y$ , as long as  $x$  and  $y$  have not changed along the way
  - Creates opportunities for **dead code elimination**: e.g., after copy propagation we may find that  $x$  is not live



- 1) Dead code elimination:  $b=a$
- 2) Strength reduction:  $e=a+a$  use left shift instead of addition

# Formulation as a System of Equations

- For each CFG node  $n$  (assume nodes = instructions)

$$\text{IN}[n] = \bigcap_{m \in \text{Predecessors}(n)} \text{OUT}[m]$$

$$\text{OUT}[\text{ENTRY}] = \emptyset$$

$$\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

- $\text{IN}[n]$  is a set of copy instructions  $\mathbf{x}=\mathbf{y}$  such that neither  $\mathbf{x}$  nor  $\mathbf{y}$  is assigned along **any** path from  $\mathbf{x}=\mathbf{y}$  to  $n$
- $\text{GEN}[n]$  is
  - A singleton set containing the copy instruction, when  $n$  is indeed a copy instruction
  - The empty set, otherwise
- $\text{KILL}[n]$ : if  $n$  assigns to  $\mathbf{x}$ , kill every  $\mathbf{y}=\mathbf{x}$  and  $\mathbf{x}=\mathbf{y}$
- Note that we must use **intersection** of  $\text{OUT}[m]$

## Worklist Algorithm (end detour)

$IN[n]$  = *the set of all copy instructions*, for all  $n$

Put the successor of ENTRY on *worklist*

While (*worklist* is not empty)

1. Remove a CFG node  $m$  from the worklist
2.  $OUT[m] = (IN[m] - KILL[m]) \cup GEN[m]$
3. For each successor  $n$  of  $m$   
     $old = IN[n]$   
     $IN[n] = IN[n] \cap OUT[m]$   
    If ( $old \neq IN[n]$ ) add  $n$  to *worklist*

In Reaching Definitions, we initialized  $IN[n]$  to the empty set; here we cannot do this, because of  $IN[n] = IN[n] \cap OUT[m]$

- Here the “meet” operation of the lattice is *set intersection*; the top element of the lattice is the set of all copy instructions
- In Reaching Definitions, “meet” is *set union*; “top” is the empty set

## Part 4: Loop Optimizations

- Loops are important for performance
- Parallelization
  - May need scalar expansion
- Loop peeling
- Loop unrolling
- Loop fusion
- Many more (see CSE 621)
  - Loop permutation (interchange)
  - Loop distribution (fission)
  - Loop tiling
  - Loop skewing
  - Index set splitting (generalization of peeling)

# Loop Parallelization

- When all iterations of a loop are independent of each other

```
for ( i = 0 ; i < 4096 ; i++ )  
    c[i] = a[i] + b[i];
```

- Needs some form of **loop dependence analysis**, which often involves reasoning about arrays
- May require enabling pre-transformations to make it parallel (e.g., **scalar expansion** or privatization)

```
for ( i = 0 ; i < 4096 ; i++ ) {  
    t = a[i] + b[i];  
    c[i] = t*t; }
```



```
double tx[4096];  
for ( i = 0 ; i < 4096 ; i++ ) {  
    tx[i] = a[i] + b[i];  
    c[i] = tx[i]*tx[i]; }  
t = tx[4095];
```

- Code generation – often with OpenMP

# Loop Peeling

- Goal: extract the first (or last) iteration
  - E.g. wraparound variables for cylindrical coordinates

```
j = N;  
for ( i = 0 ; i < N ; i++ ) {  
    b[i] = (a[i] + a[j]) / 2;  
    j = i; } // assume j is not live here
```

```
b[0] = (a[0] + a[N]) / 2;  
b[1] = (a[1] + a[0]) / 2;  
b[2] = (a[2] + a[1]) / 2  
...
```

- Peel-off the first iteration, then do copy propagation and induction variable analysis

```
j = N;  
if (N >= 1) {  
    b[0] = (a[0] + a[j]) / 2;  
    j = 0;  
    for ( i = 1 ; i < N ; i++ ) {  
        b[i] = (a[i] + a[j]) / 2;  
        j = i; } } }
```



```
if (N >= 1) {  
    b[0] = (a[0] + a[N]) / 2;  
    for ( i = 1 ; i < N ; i++ ) {  
        b[i] = (a[i] + a[i-1]) / 2; } }  
// now we can do unrolling
```

# Loop Unrolling

- Loop unrolling: extend the body

```
for ( i = 0 ; i < 4096 ; i++ )  
    c[i] = a[i] + b[i];
```



```
for ( i = 0 ; i < 4095 ; i +=2 ) {  
    c[i] = a[i] + b[i];  
    c[i+1] = a[i+1] + b[i+1];  
} // unroll factor of 2
```

- Reduces the “control overhead” of the loop: makes the loop exit test ( $i < 4096$ ) execute less frequently
- Hardware advantages: instruction-level parallelism; fewer pipeline stalls
- Issue: loop bound may not be a multiple of unroll factor
- Problem: high unroll factors may degrade performance due to register pressure and spills (more later)

# Loop Fusion

- Merge two loops with compatible bounds

```
for ( i = 0 ; i < N ; i++ )  
    c[i] = a[i] + b[i];  
for ( j = 0 ; j < N ; j++ )  
    d[i] = c[i] * 2;
```



```
for ( k = 0 ; k < N ; k++ ) {  
    c[k] = a[k] + b[k];  
    d[k] = c[k] * 2;  
}
```

- Reduces the loop control overhead (i.e., loop exit test)
- May improve cache use (e.g. the reuse of **c[k]** above) – especially producer/consumer loops [↓capacity misses]
- Fewer parallelizable loops and increased work per loop: reduces parallelization overhead (cost of spawn/join)
- If the loop bounds are “±1 off” – use peeling
- Not always legal – need dependence analysis