

CSL302: Compiler Design

Machine Independent Optimizations

Vishwesh Jatala

Assistant Professor

Department of CSE

Indian Institute of Technology Bhilai

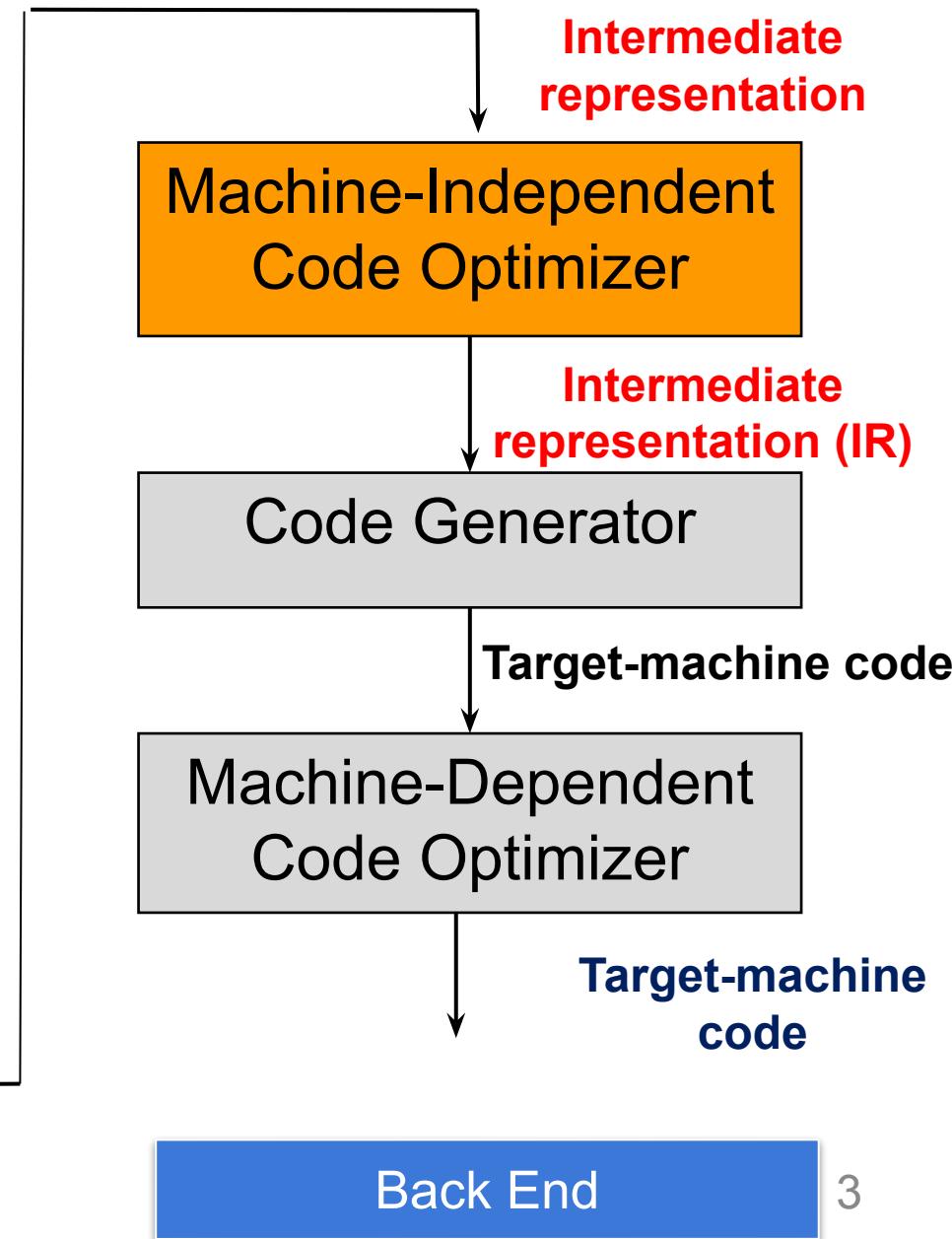
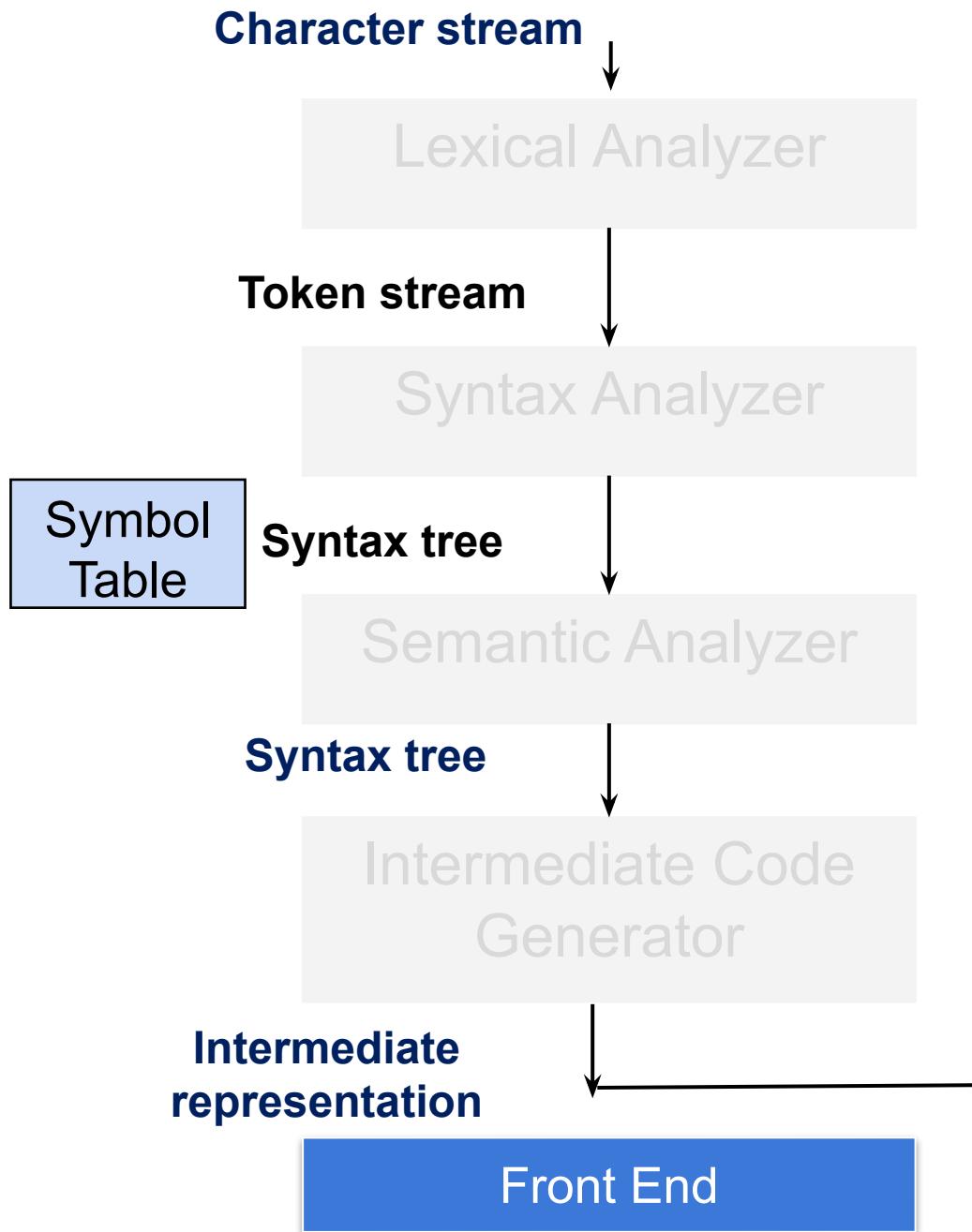
vishwesh@iitbhilai.ac.in



Acknowledgement

- References for today's slides
 - *Stanford University*
<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/>
 - *Prof. Y. N Srikant, IISc Bangalore*
<https://iith.ac.in/~ramakrishna/Compilers-Aug14/slides/>
 - *http://sei.pku.edu.cn/~yaoguo/ACT11/slides/lect2-opt.ppt*
 - *Course textbook*

Compiler Design



Examples

```
1) i = 1  
2) j = 1  
3) t1 = 10 * i  
4) t2 = t1 + j  
5) t3 = 8 * t2  
6) t4 = t3 - 88  
7) a[t4] = 0.0  
8) j = j + 1  
9) if j <= 10 goto (3)  
10) i = i + 1  
11) if i <= 10 goto (2)  
12) i = 1  
13) t5 = i - 1  
14) t6 = 88 * t5  
15) a[t6] = 1.0  
16) i = i + 1  
17) if i <= 10 goto (13)
```

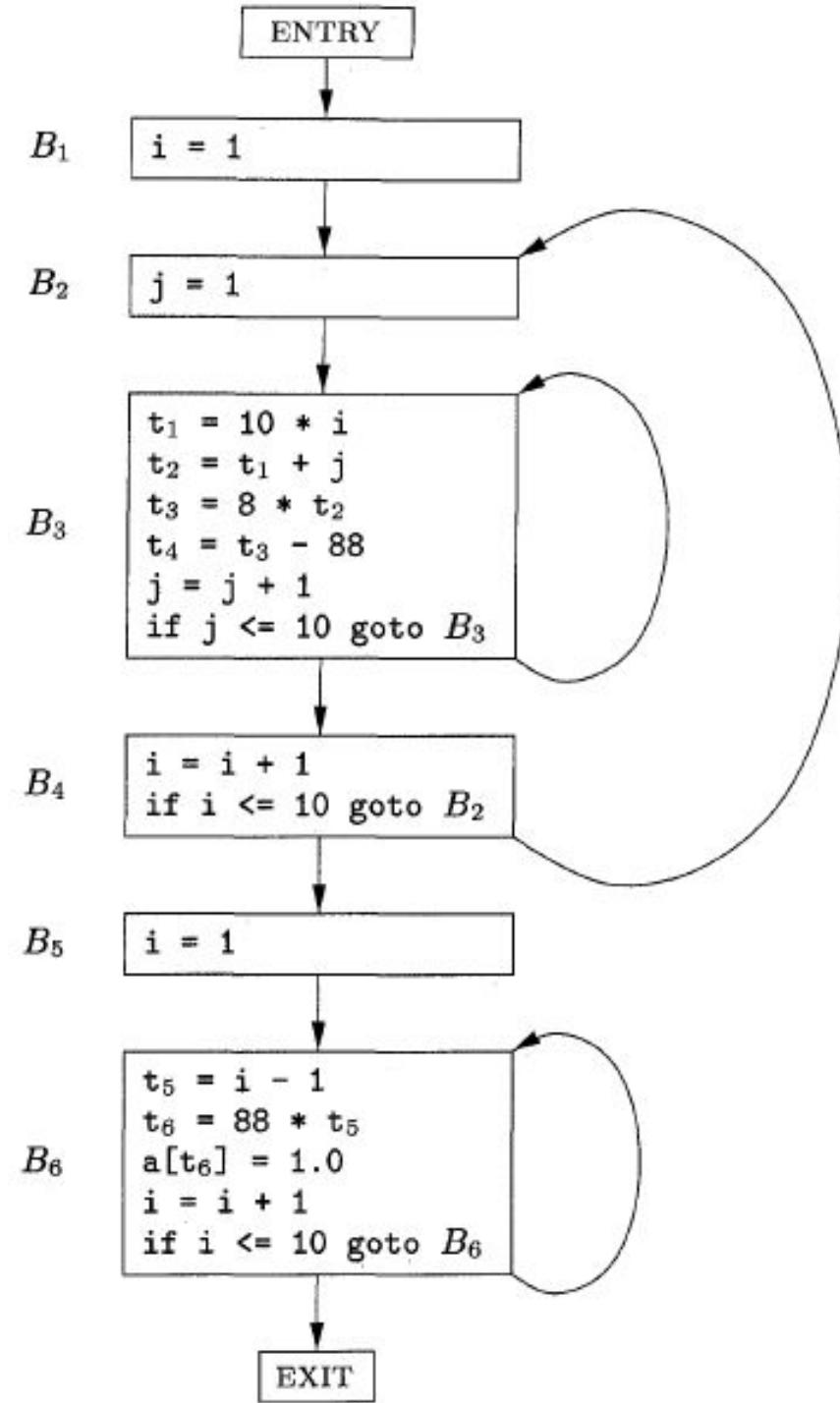
```
for i from 1 to 10 do  
  for j from 1 to 10 do  
    a[i,j]=0.0
```

```
for i from 1 to 10 do  
  a[i,i]=0.0
```

Control Flow

```
for i from 1 to 10 do  
  for j from 1 to 10 do  
    a[i,j]=0.0
```

```
for i from 1 to 10 do  
  a[i,i]=0.0
```



Basic Blocks

- A **basic block** is a maximal sequence of consecutive three-address instructions with the following properties:
 - The flow of control can only enter the basic block thru the 1st instr.
 - There is exactly one spot where control leaves the sequence, which must be at the end of the sequence.
- Basic blocks become the nodes of a **flow graph**, with edges indicating the order.

Identifying Basic Blocks

- Input: sequence of instructions $instr(i)$
- Output: A list of basic blocks
- Method:
 - Identify **leaders**:
the first instruction of a basic block
 - Iterate: add subsequent instructions to basic block
until we reach another leader

Identifying Leaders

- Rules for finding leaders in code
 - First instr in the code is a leader
 - Any instr that is the target of a (conditional or unconditional) jump is a leader
 - Any instr that immediately follow a (conditional or unconditional) jump is a leader

Basic Block Example

```
i = 1  
j = 1  
t1 = 10 * i  
t2 = t1 + j  
t3 = 8 * t2  
t4 = t3 - 88  
a[t4] = 0.0  
j = j + 1  
if j <= 10 goto (3)  
i = i + 1  
if i <= 10 goto (2)  
i = 1  
t5 = i - 1  
t6 = 88 * t5  
a[t6] = 1.0  
i = i + 1  
if i <= 10 goto (13)
```

A
B

C

D
E

F



Control Flow Graphs

- **Control-flow graph:**
 - Node: an instruction or sequence of instructions
(a **basic block**)
 - Two instructions i, j in same basic block
iff execution of i guarantees execution of j
 - Directed edge: ***potential*** flow of control
 - Distinguished start node ***Entry & Exit***
 - First & last instruction in program

Control Flow Edges

- Basic blocks = nodes
- Edges:
 - Add directed edge between B1 and B2 if:
 - Branch from last statement of B1 to first statement of B2 (B2 is a leader), or
 - B2 immediately follows B1 in program order and B1 does not end with unconditional branch (goto)
 - Definition of **predecessor** and **successor**
 - B1 is a predecessor of B2
 - B2 is a successor of B1

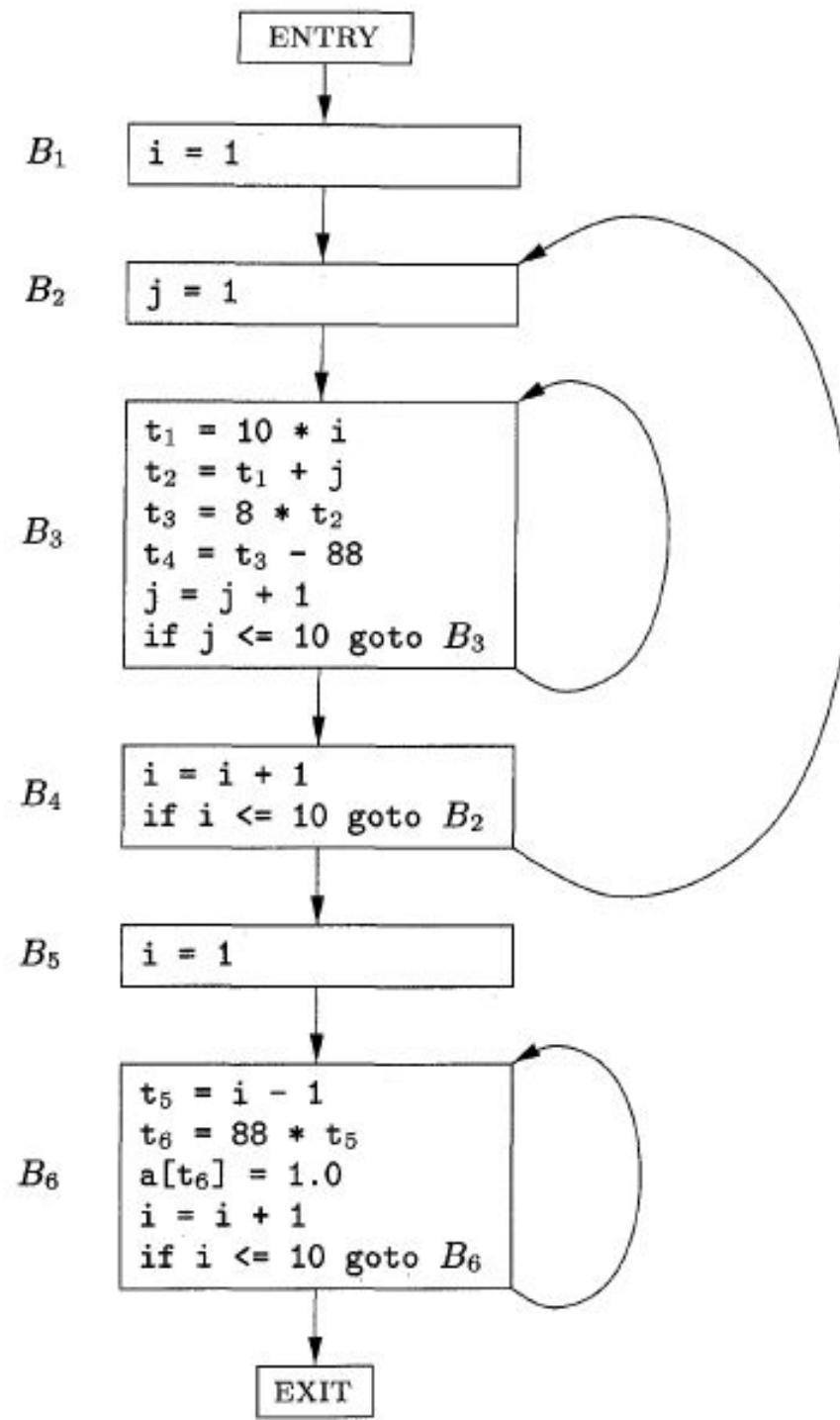
Control Flow Algorithm

Input: $\text{block}(i)$, sequence of basic blocks

Output: CFG where nodes are basic blocks

```
for i = 1 to the number of blocks
    x = last instruction of block(i)
    if instr(x) is a branch
        for each target y of instr(x),
            create edge (i -> y)
    if instr(x) is not unconditional branch,
        create edge (i -> i+1)
```

CFG Example



Optimizations

- Global common subexpression elimination
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction

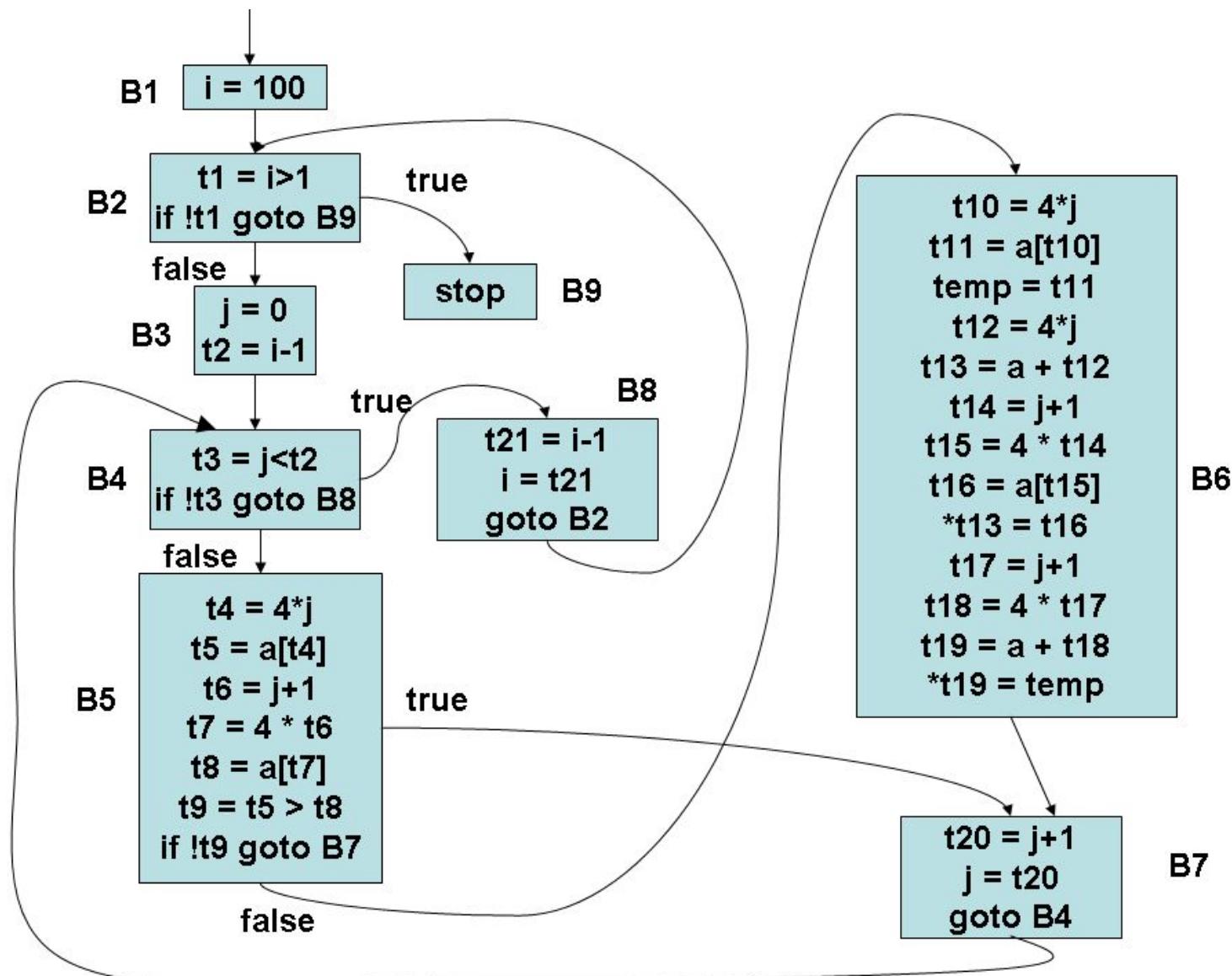
Example

Bubble Sort

```
for (i=100; i>1; i--) {  
    for (j=0; j<i-1; j++) {  
        if (a[j] > a[j+1]) {  
            temp = a[j];  
            a[j+1] = a[j];  
            a[j] = temp;  
        }  
    }  
}
```

- int a[100]
- array a runs from 0 to 99
- No special jump out if array is already sorted

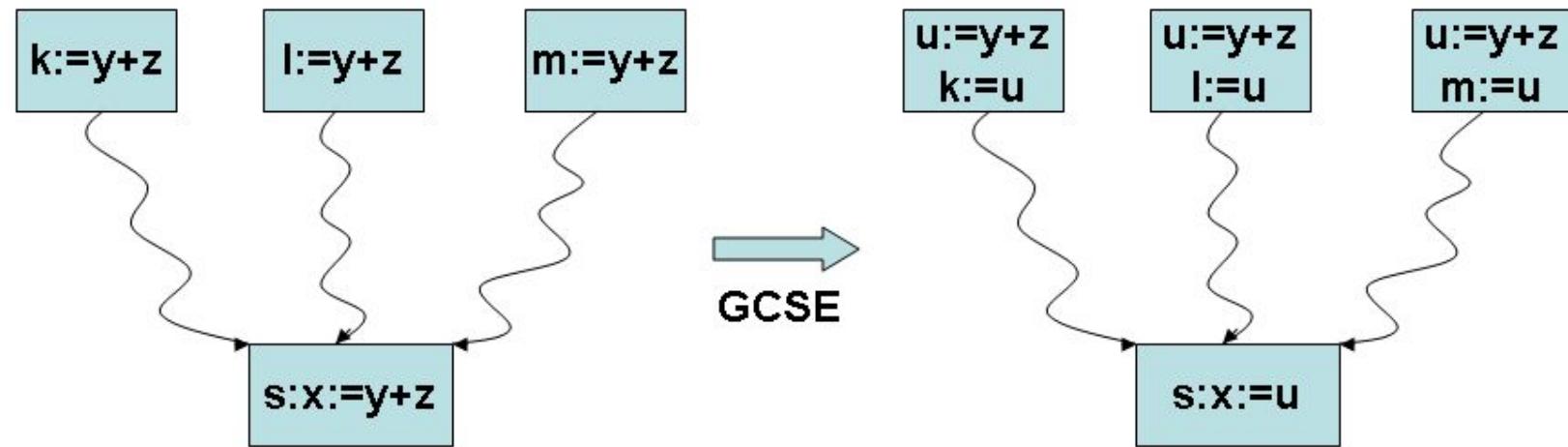
Control Flow Graph of Bubble Sort



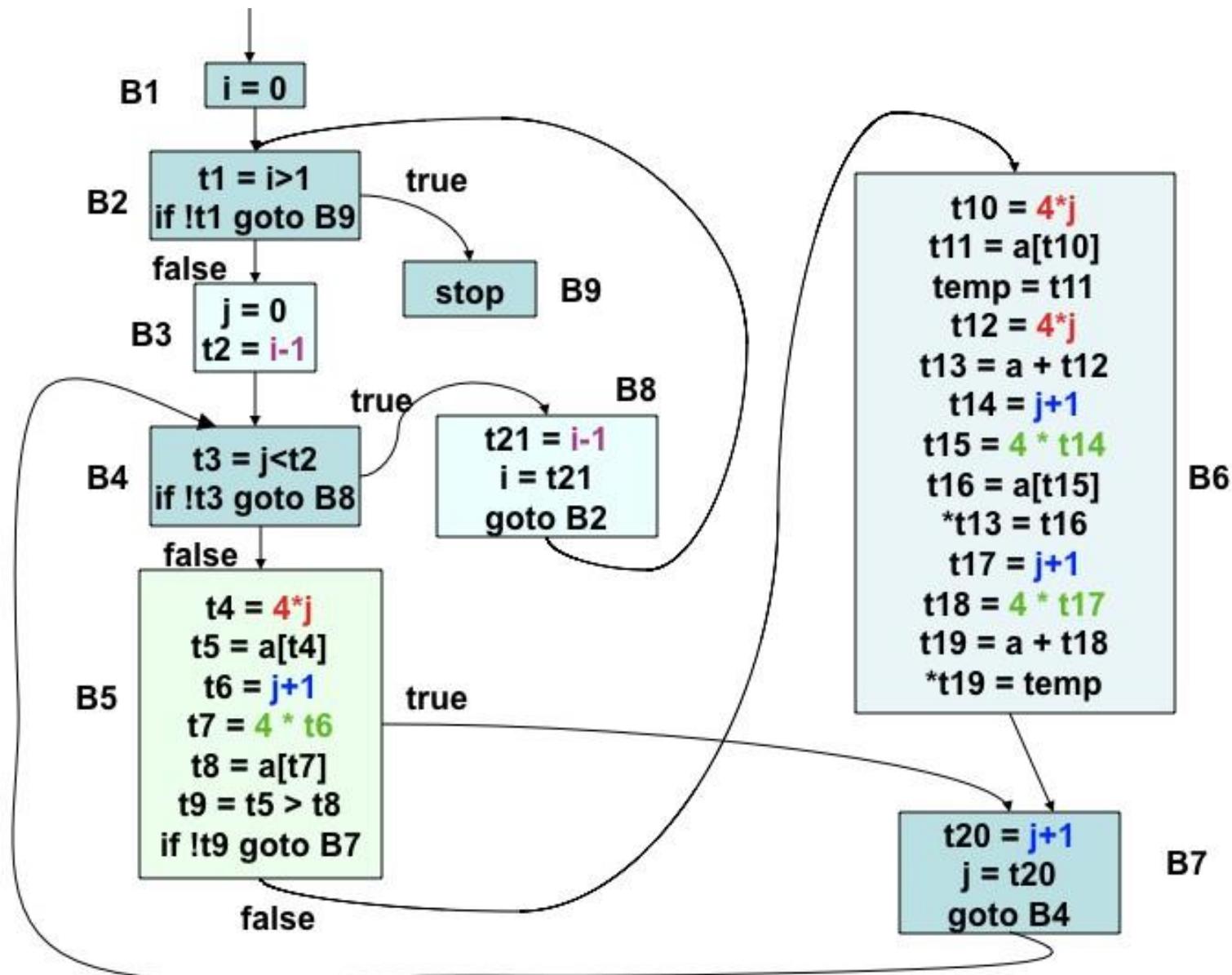
Optimizations

- **Global common subexpression elimination**
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction

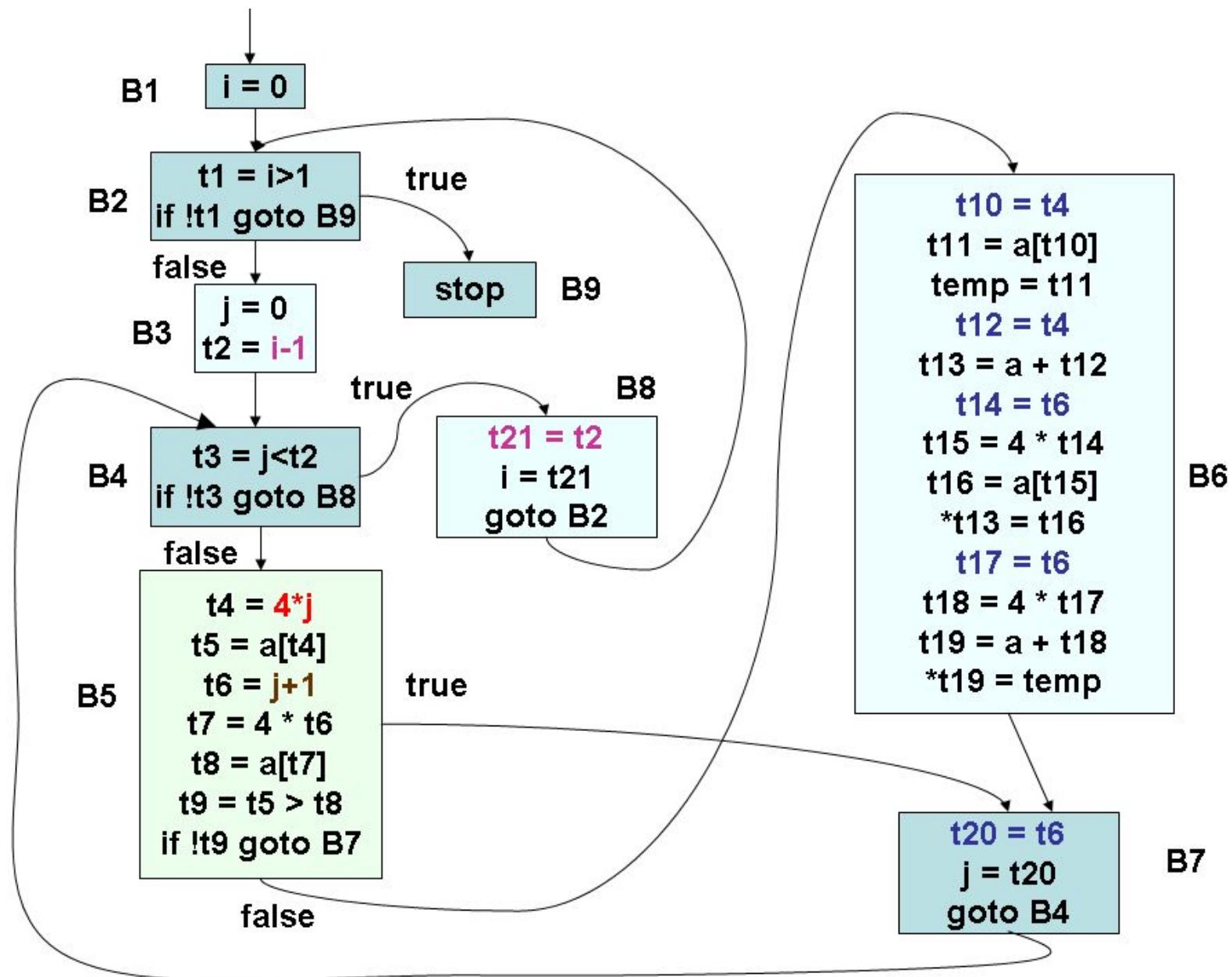
Global Common Subexpression Elimination (GCSE)



Global Common Subexpression Elimination (GCSE)



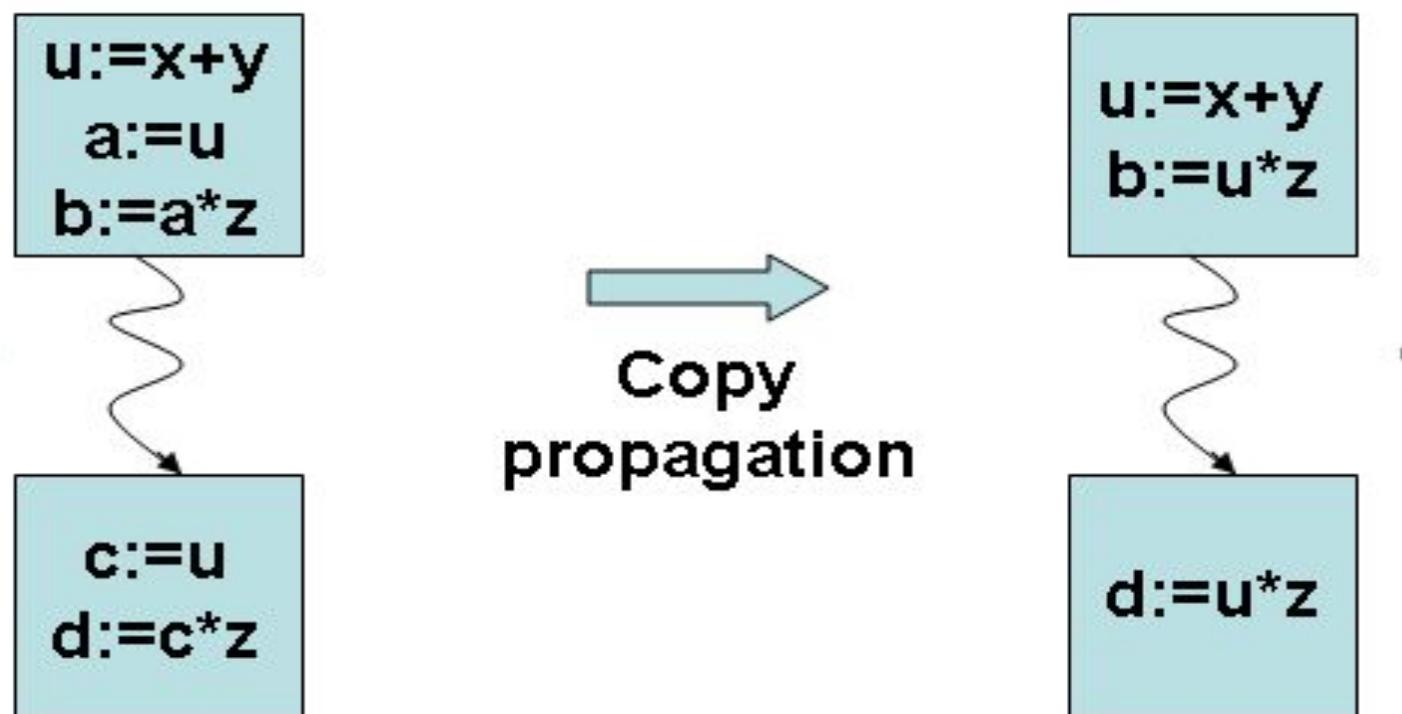
Global Common Subexpression Elimination (GCSE)



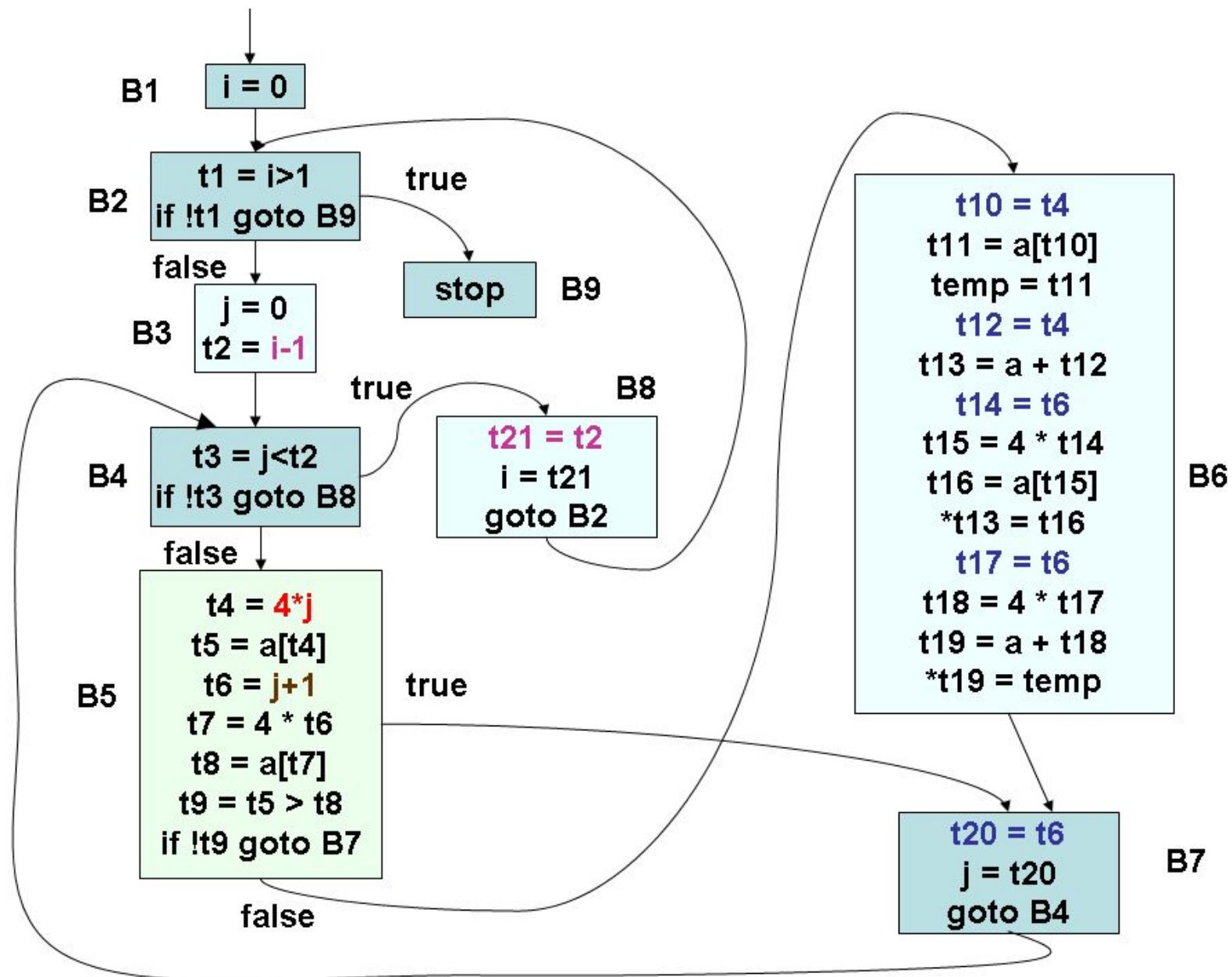
Optimizations

- Global common subexpression elimination
- **Copy propagation**
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction

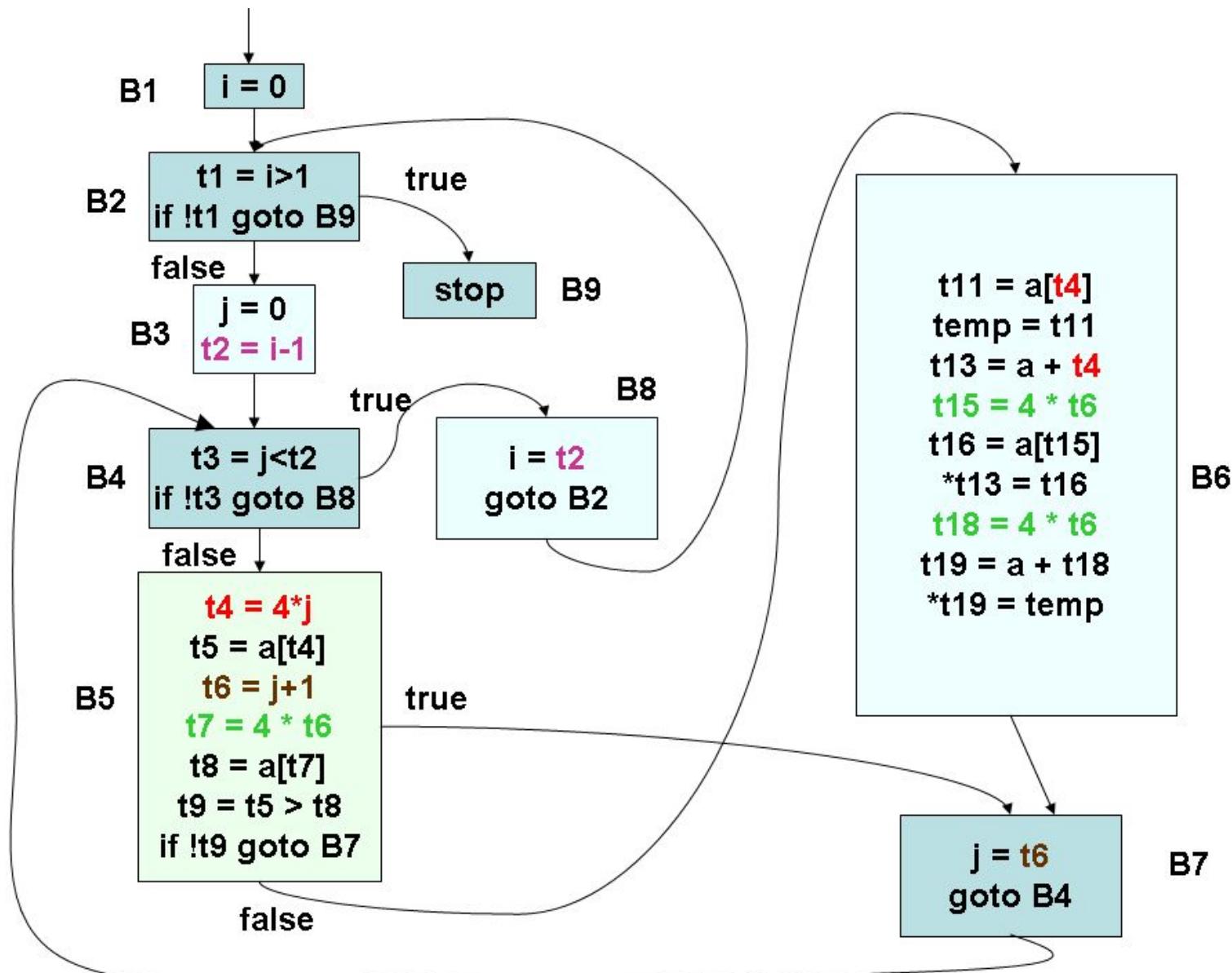
Copy Propagation



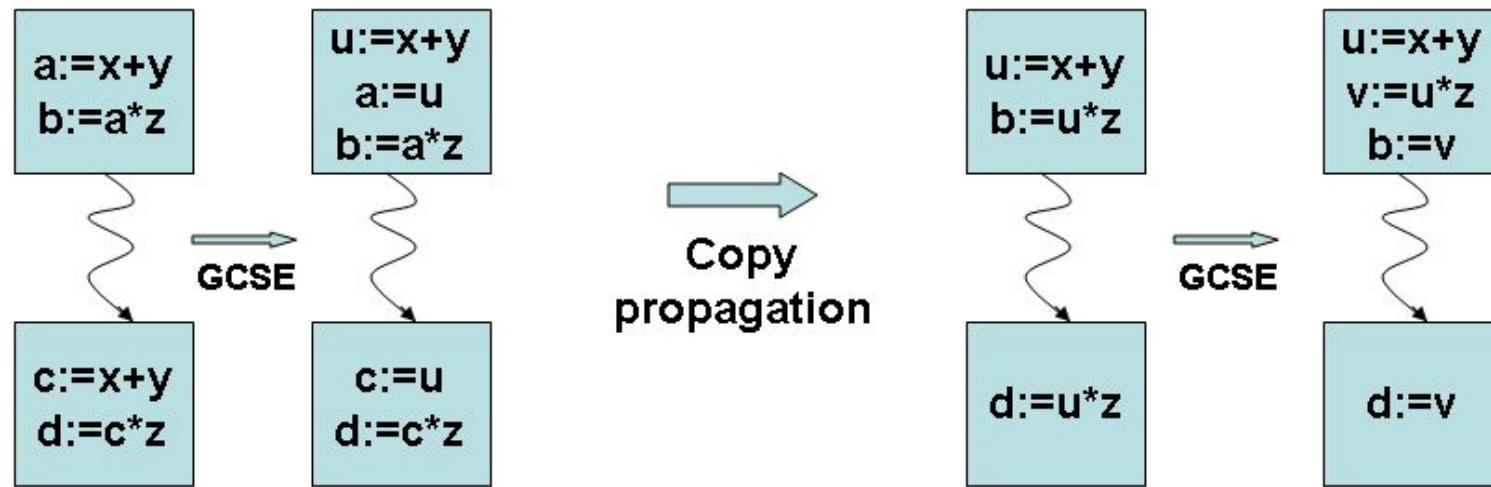
Global Common Subexpression Elimination (GCSE)



Copy Propagation

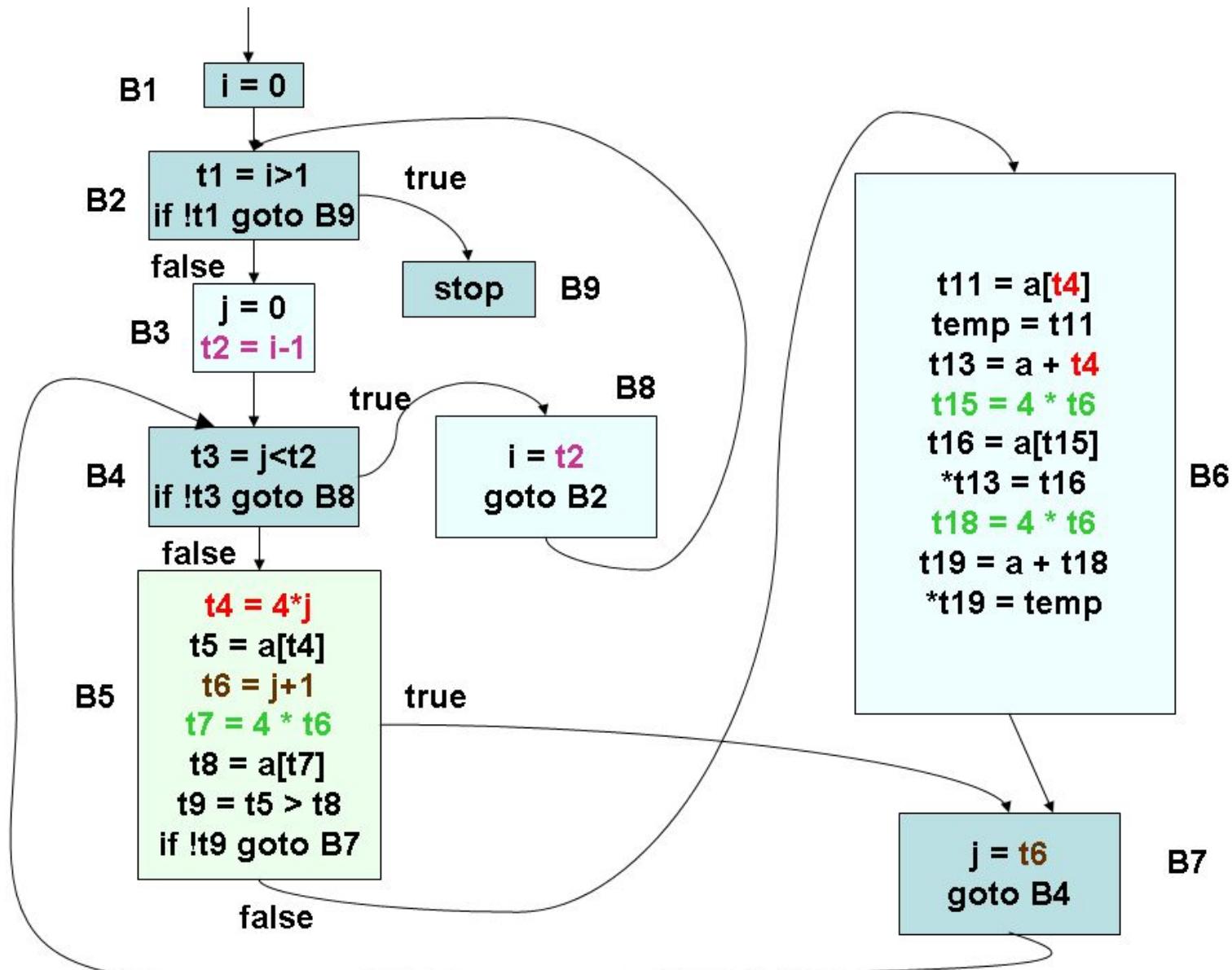


GCSE and Copy Propagation

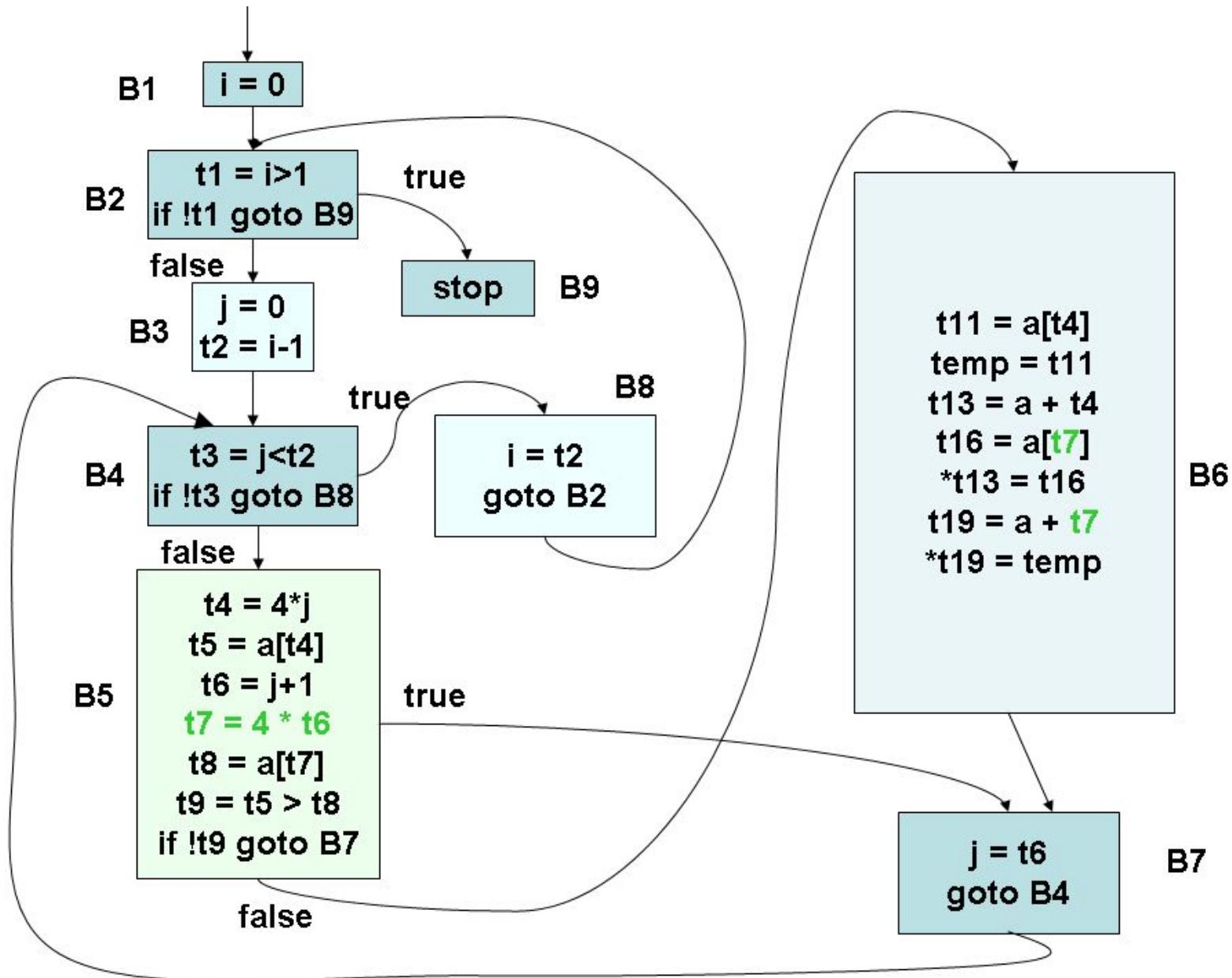


Demonstrating the need for repeated application of GCSE

GCSE and Copy Propagation



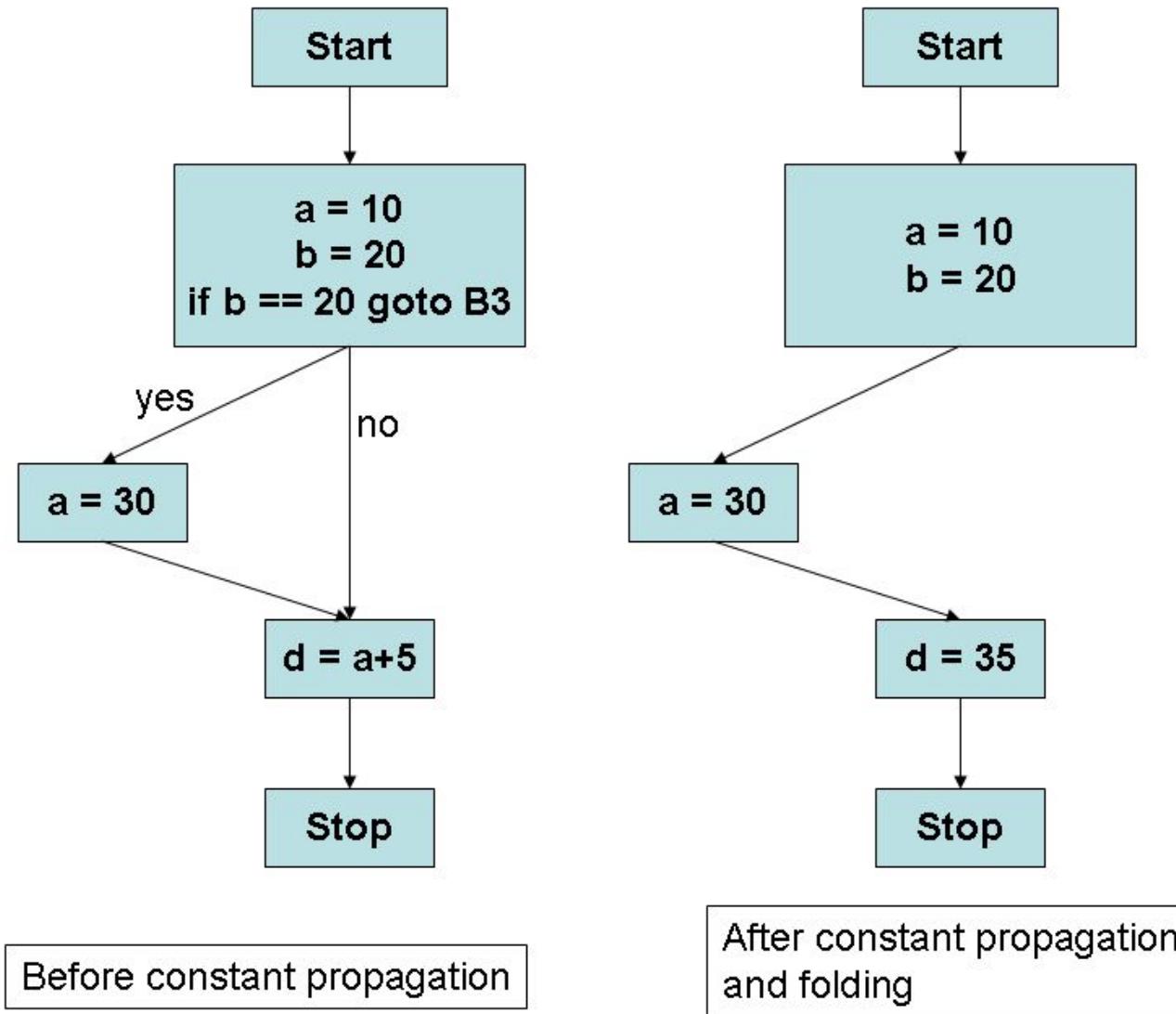
GCSE and Copy Propagation



Optimizations

- Global common subexpression elimination
- Copy propagation
- **Constant propagation and constant folding**
- Loop invariant code motion
- Induction variable elimination and strength reduction

Copy Propagation and Constant Folding



Optimizations

- Global common subexpression elimination
- Copy propagation
- Constant propagation and constant folding
- **Loop invariant code motion**
- Induction variable elimination and strength reduction

Loop Invariant Code Motion Example

```
t1 = 202  
i = 1  
L1: t2 = i>100  
    if t2 goto L2  
    t1 = t1-2  
    t3 = addr(a)  
    t4 = t3 - 4  
    t5 = 4*i  
    t6 = t4+t5  
    *t6 = t1  
    i = i+1  
    goto L1  
  
L2:
```

Before LIV
code motion

```
t1 = 202  
i = 1  
t3 = addr(a)  
t4 = t3 - 4  
L1: t2 = i>100  
    if t2 goto L2  
    t1 = t1-2  
    t5 = 4*i  
    t6 = t4+t5  
    *t6 = t1  
    i = i+1  
    goto L1  
  
L2:
```

After LIV
code motion

Optimizations

- Global common subexpression elimination
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- **Induction variable elimination and strength reduction**

Strength Reduction

```
t1 = 202  
i = 1  
t3 = addr(a)  
t4 = t3 - 4  
L1: t2 = i>100  
    if t2 goto L2  
    t1 = t1-2  
t5 = 4*i  
    t6 = t4+t5  
    *t6 = t1  
    i = i+1  
    goto L1  
L2:
```

Before strength reduction for t5

```
t1 = 202  
i = 1  
t3 = addr(a)  
t4 = t3 - 4  
t7 = 4  
L1: t2 = i>100  
    if t2 goto L2  
    t1 = t1-2  
    t6 = t4+t7  
    *t6 = t1  
    i = i+1  
t7 = t7 + 4  
    goto L1  
L2:
```

After strength reduction for t5 and copy propagation

Induction Variable Elimination

```
t1 = 202  
i = 1  
t3 = addr(a)  
t4 = t3 - 4  
t7 = 4  
L1: t2 = i > 100  
    if t2 goto L2  
    t1 = t1-2  
    t6 = t4+t7  
    *t6 = t1  
    i = i+1  
    t7 = t7 + 4  
    goto L1  
L2:
```

```
t1 = 202  
t3 = addr(a)  
t4 = t3 - 4  
t7 = 4  
L1: t2 = t7 > 400  
    if t2 goto L2  
    t1 = t1-2  
    t6 = t4+t7  
    *t6 = t1  
    t7 = t7 + 4  
    goto L1  
L2:
```

After eliminating i and
replacing it with t7

Before induction variable
elimination (i)

Summary

- Machine Independent Optimizations
 - Improve the quality of code: performance, memory, and energy efficiency
 - Still hot area of research
- Formalize:
 - Basic blocks
 - Control flow graph
- Optimizations:
 - Examples

Next Lecture

