Machine-Independent Optimizations
Lecture 18

Code optimization

- Elimination of unnecessary instructions
- Replacement of one sequence of instructions by a faster sequence of instructions

- Local optimization
- Global optimizations
  - based on data flow analyses

The Principal Sources of Optimization

- Optimization
  - Preserves the semantics of the original program

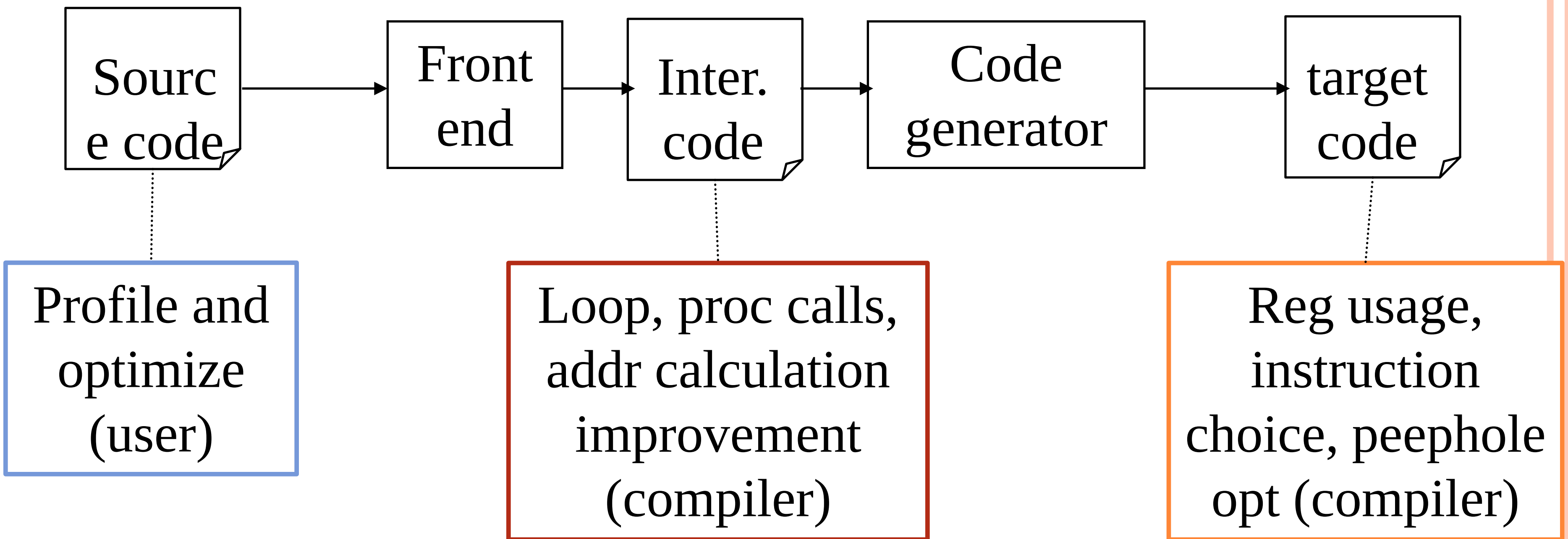
    Except in very special circumstances, once a programmer chooses and implements a particular algorithm, the compiler cannot understand enough about the program to replace it with a substantially different and more efficient algorithm.

  - Applies relatively low-level semantic transformations

    using general facts such as algebraic identities like $i + 0 = i$ or program semantics such as the fact that performing the same operation on the same values yields the same result.

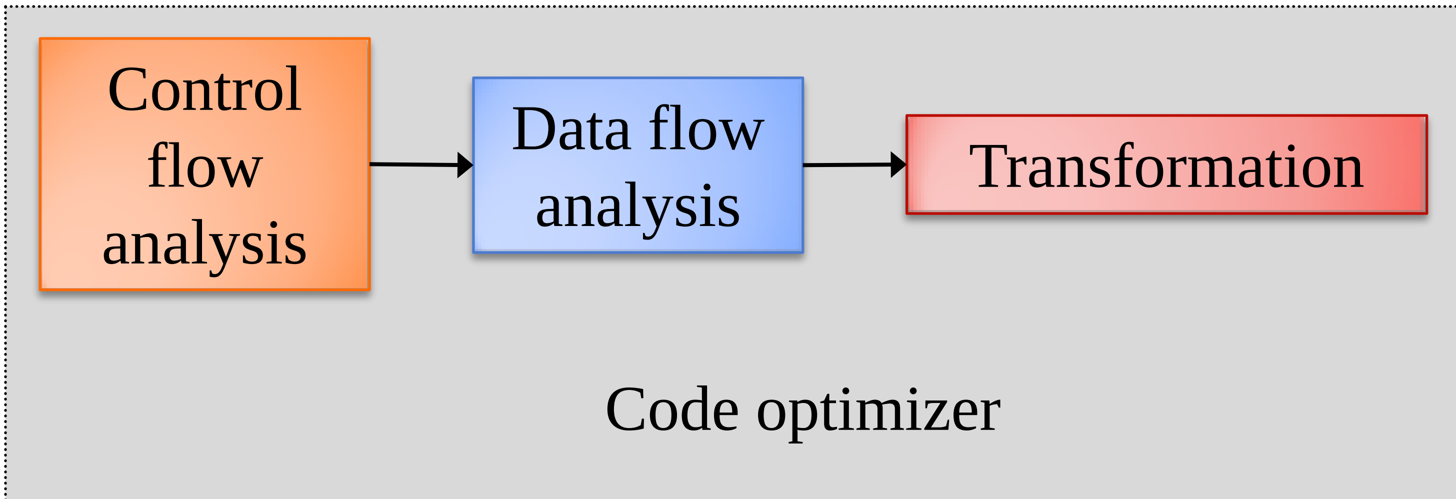
INTRODUCTION

- Optimization can be done in almost all phases of compilation.



INTRODUCTION

- Organization of an optimizing compiler



Themes behind Optimization Techniques

- Avoid redundancy: something already computed need not be computed again
- Smaller code: less work for CPU, cache, and memory!
- Less jumps: jumps interfere with code pre-fetch
- Code locality: codes executed close together in time is generated close together in memory – increase locality of reference
- Extract more information about code: More info – better code generation

Causes of Redundancy

- Redundant operations are
  - at the source level
  - a side effect of having written the program in a high-level language
- Each of high-level data-structure accesses expands into a number of low-level arithmetic operations
- Programmers are not aware of these low-level operations and cannot eliminate the redundancies themselves.
- By having a compiler eliminate the redundancies
  - The programs are both efficient and easy to maintain.

Scalar Compiler Optimizations

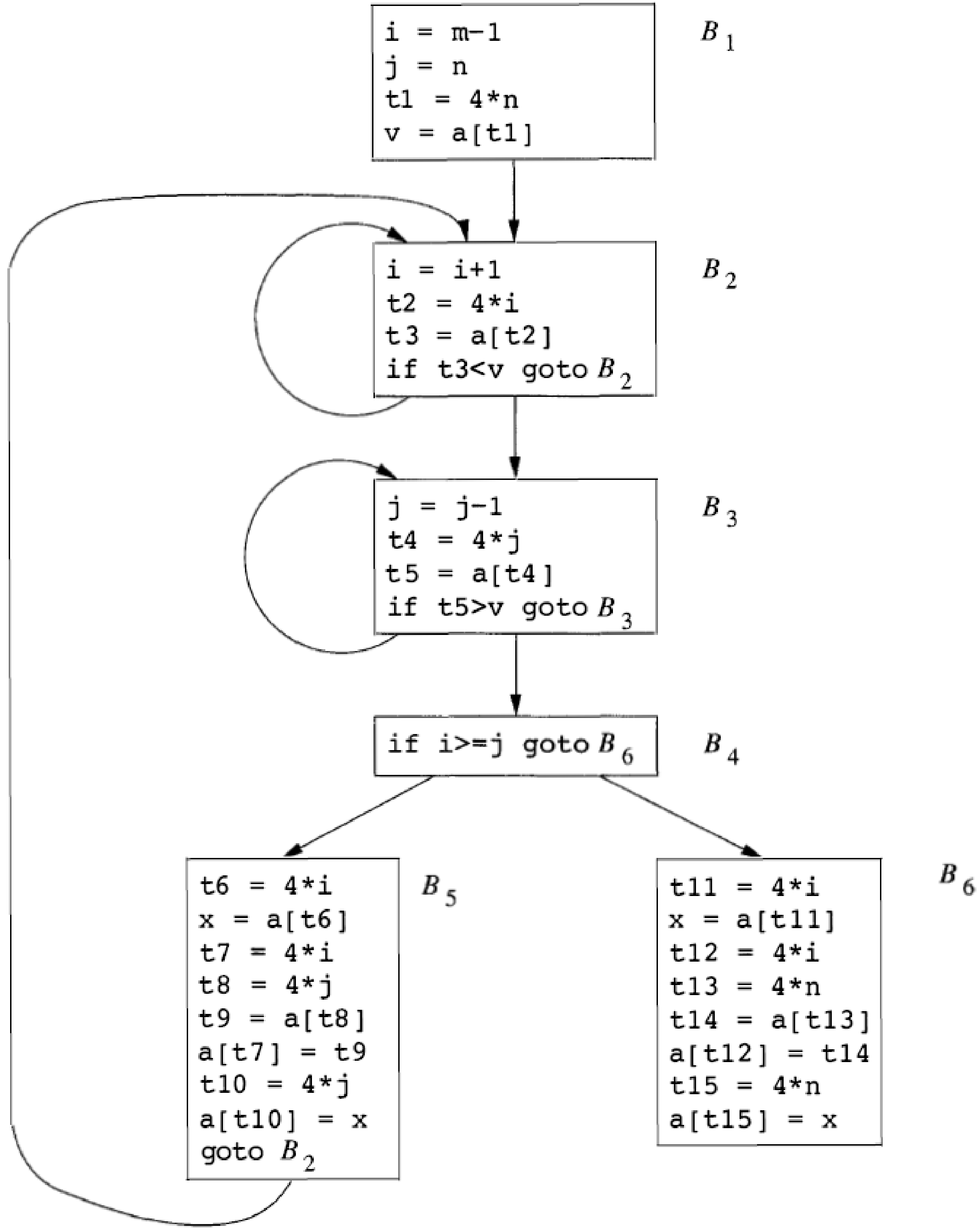
- Machine independent optimizations
  - Enable other transformations
    - Procedure inlining, cloning, loop unrolling
  - Eliminate redundancy
    - Redundant expression elimination
  - Eliminate useless and unreachable code
    - Dead code elimination
  - Specialization and strength reduction
    - Constant propagation, peephole optimization
  - Move operations to less-frequently executed places
    - Loop invariant code motion

Scalar Compiler Optimizations

- Machine dependent (scheduling) transformations
  - Take advantage of special hardware features
    - Instruction selection, prefetching
  - Manage or hide latency, introduce parallelism
    - Instruction scheduling, prefetching
  - Manage bounded machine resources
    - Register allocation

A Running Example: Quicksort

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[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; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```


i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
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
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
B6

Semantics-Preserving Transformations

- A number of ways in which a compiler can improve a program without changing the function it computes
  - ***Common-sub expression elimination***
  - ***Copy propagation***
  - ***Dead-code elimination***
  - ***Constant folding***

Common Subexpressions Elimination

- Common subexpression
  - Previously computed
  - The values of the variables not changed

- Local:

```
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
```

B_5

(a) Before.

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

B_5

(b) After.

Common Subexpressions Elimination

Original code

```
m := 2 * y * z
n := 3 * y * z
o := 2 * y - z
```

Rewritten code

```
t0:=2 * y
m := t0 * z
n := 3 * y * z
o := t0 - z
```

- The second 2*y computation is redundant
- What about y*z?
  - 2*y*z → (2*y) * z   not 2*(y*z)
  - 3*y*z→ (3*y) * z   not 3*(y*z)
  - Change associativity may change evaluation result
    - For integer operations, optimization is sensitive to ordering of operands
- Typically applied only to integer expressions due to precision concerns

Common Subexpressions Elimination

```
a := x + y
b := x + y
a := 17
c := x + y
```

(1)

```
m := 2 * y * z
y := 3 * y * z
o := 2 * y - z
```

(2)

```
m := 2 * y * z
*p := 3 * y * z
o := 2 * y - z
```

(3)

(1) The expression `x+y' is redundant, but no longer available in 'a' when being assigned to `c'

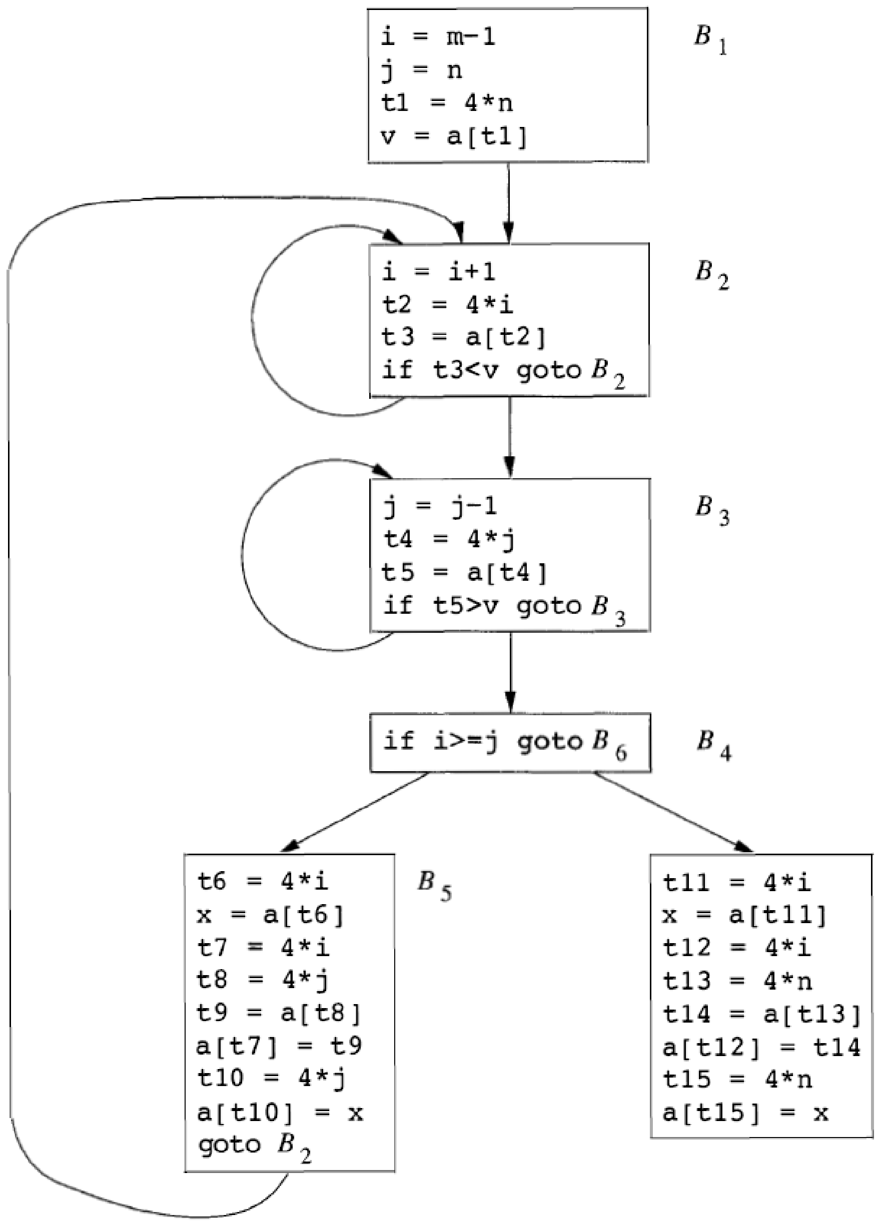
- Keep track of available variables for each value number
- Create new temporary variables for value numbers if necessary

(2) The expression 2*y is not redundant

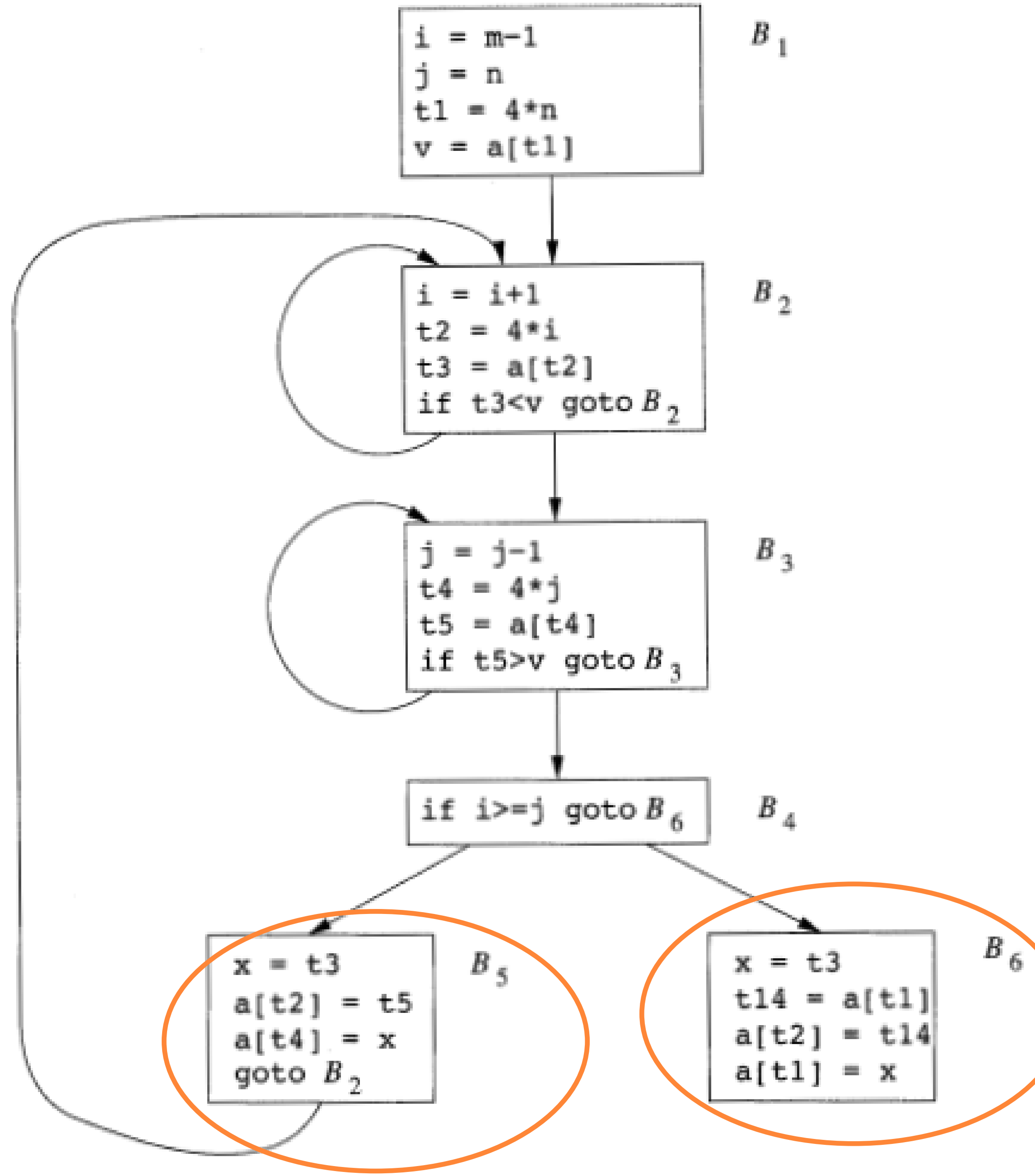
- the two 2*y evaluation have different values

(3) Pointer Variables could point to anywhere

- If p points to y, then 2*y is no longer redundant
- All variables (memory locations) may be modified from modifying *p
- Pointer analysis ---reduce the set of variables associated with p



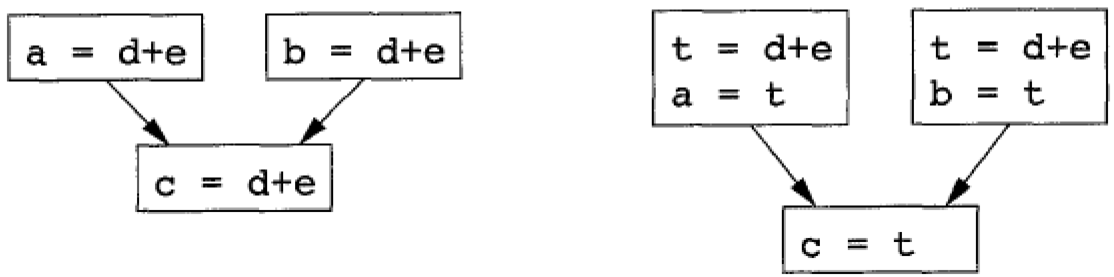
i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
B5
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x



i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6

Copy Propagation

- *Copy statements* or *Copies*
  - u = v



(a) (b)

```
b := z + y          b := z + y
a := b       ⇒      a := b
x := 2 * a          x := 2 * b
```

Copy Propagation & Constant folding

- *Constant folding*
  - Deducing at compile time that the value of an expression is a constant and using the constant instead

```
a := 5                a := 5
x := 2 * a     ⇒      x := 10
y := x + 6            y := 16
t := x * y            t := x << 4
```

Copy Propagation

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

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

Dead-Code Elimination

- Eliminate instructions whose results are never used
  - mark all critical instructions as useful
    - Instructions that return values, perform input/output, or modify externally visible storage
  - Mark all instructions that affect already marked instruction i
    - Instructions that define operands of i or control the execution of i

```
void foo(int b, int c) {
  int a, d, e, f;
  a := b + c;
  d := b – c;
  e := b * c;
  f := b / c;
  return e;
}
```

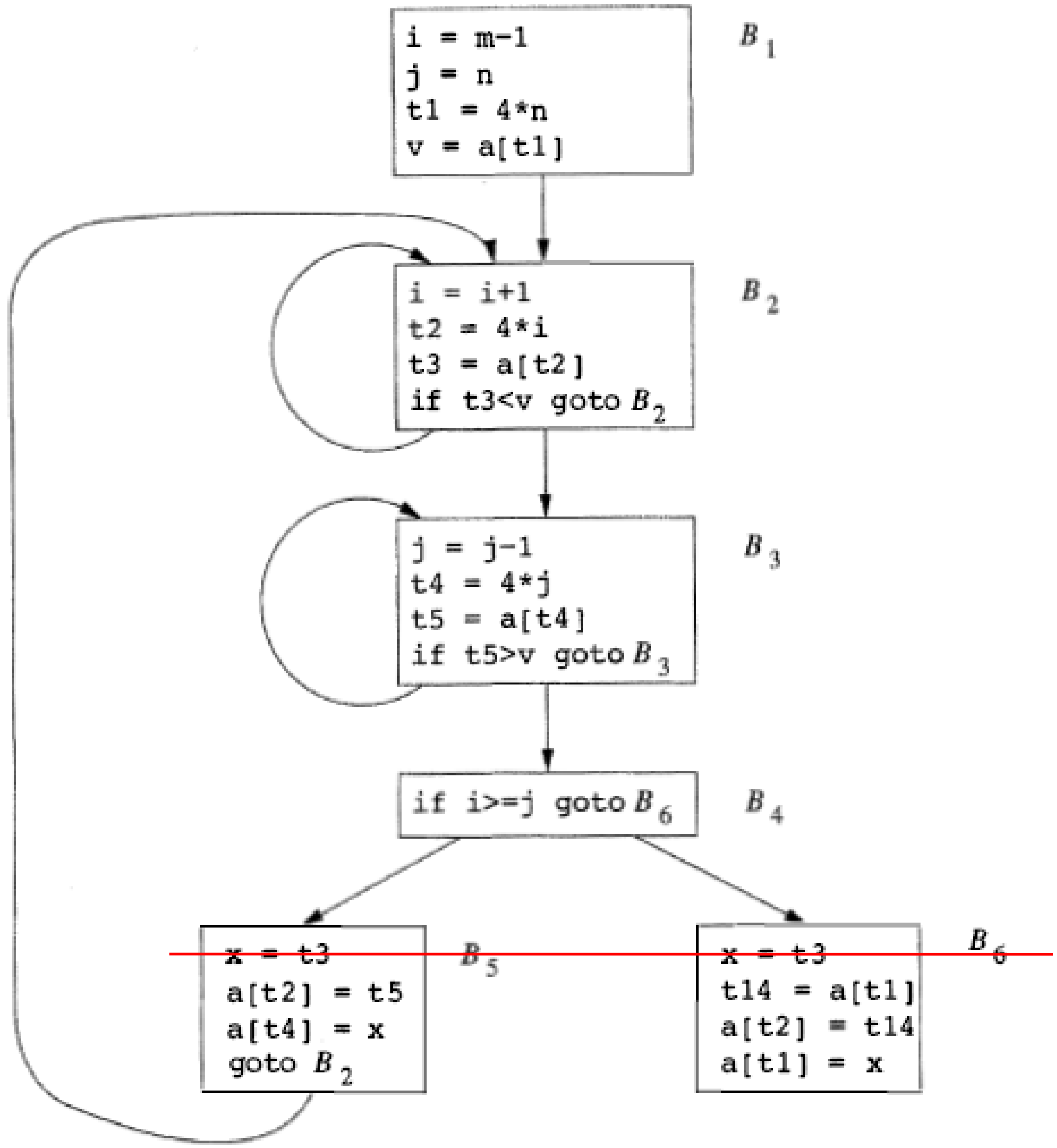
Useless code:

```
 a := b + c;
 d := b – c;
 f := b / c;
```

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

→

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


i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
~~x = t3~~
a[t2] = t5
a[t4] = x
goto B2
B5
~~x = t3~~
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6

Code Motion

- Moving code from one part of the program to other without modifying the algorithm
  - Reduce size of the program
  - Reduce execution frequency of the code subjected to movement

Code Motion

1. *Code Space reduction*: Similar to common sub-expression elimination but with the objective to reduce code size.

Example: Code hoisting

```
if (a< b) then
    z := x ** 2
else
    y := x ** 2 + 10
```

→

```
temp : = x ** 2
if (a< b) then
   z := temp
else
  y := temp + 10
```

"x ** 2" is computed once in both cases, but the code size in the second case reduces.

Code Motion

- *Execution frequency reduction*: reduce execution frequency of partially available expressions (expressions available atleast in one path)

Example:

```
if (a<b) then
        z = x * 2

else
y = 10

g = x * 2
```

→

```
if (a<b) then
   temp = x * 2
   z = temp
else
   y = 10
   temp = x * 2
              g = temp;
```

Code Motion

- Move expression out of a loop if the evaluation does not change inside the loop.

Example:

```
while ( i < (max-2) ) …
```

Equivalent to:

```
t :=  max - 2
while ( i < t ) …
```

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

Code Motion

- Safety of Code movement

  Movement of an expression e from a basic block b_i to another block b_j, is safe if it does not introduce any new occurrence of e along any path.

Example: Unsafe code movement

```
if (a<b) then                  temp = x * 2
  z = x * 2                    if (a<b) then
else                 →           z = temp
  y = 10                       else
                           y = 10
```

Strength Reduction

- Replacement of an operator with a less costly one.

Example:

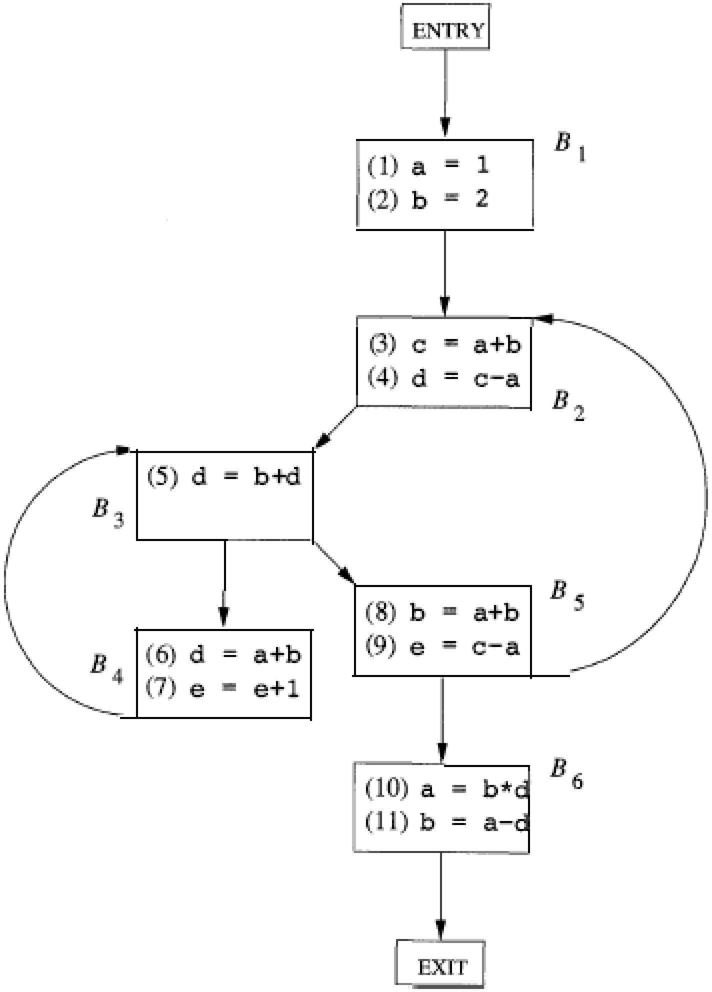
```
for i=1 to 10 do                     temp = 5;
  …                                  for i=1 to 10 do
  x = i * 5          →                 …
  …                                    x = temp
                                       …
end                                    temp = temp + 5
                                     end
```

- Typical cases of strength reduction occurs in address calculation of array references.
- Applies to integer expressions involving induction variables (loop optimization)

TEST YOURSELF

- E-9.1.1

a) Identify the loops of the flow graph.
b) Statements (1) and (2) in B_1 are both copy statements, in which a and b are given constant values. For which uses of a and b can we perform copy propagation and replace these uses of variables by uses of a constant? Do so, wherever possible.
c) Identify any global common sub expressions for each loop.
d) Identify any induction variables for each loop. Be sure to take into account any constants introduced in (b).
e) Identify any loop-invariant computations for each loop.



Peephole Optimization

Peephole Optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognizing sets of instructions that can be replaced by shorter or faster sets of instructions.

Goals:

- improve performance
- reduce memory footprint
- reduce code size

Peephole Optimization

- Pass over generated code to examine a few instructions, typically 2 to 4
  - Redundant instruction Elimination: Use algebraic identities
  - Flow of control optimization: removal of redundant jumps
  - Use of machine idioms

Algebraic identities

- Worth recognizing single instructions with a constant operand:

  `A * 1 = A`

  `A * 0 = 0`

  `A / 1 = A`

  `A * 2 = A + A`

  More delicate with floating-point

- Strength reduction:

  `A ^ 2 = A * A`

Objective

- Why would anyone write **`X * 1`**?
- Why bother to correct such obvious junk code?
- In fact one might write

```
#define MAX_TASKS  1
...
a = b * MAX_TASKS;
```

- Also, seemingly redundant code can be produced by other optimizations. This is an important effect.

Replace Multiply by Shift

- `A := A * 4;`
  - Can be replaced by 2-bit left shift (signed/unsigned)
  - But must worry about overflow if language does
- `A := A / 4;`
  - If unsigned, can replace with shift right
  - But shift right arithmetic is a well-known problem
  - Language may allow it anyway (traditional C)

Addition chains for multiplication

- If multiply is very slow (or on a machine with no multiply instruction like the original SPARC), decomposing a constant operand into sum of powers of two can be effective:

```
X * 125   =    x * 128 - x*4 + x
```

  - two shifts, one subtract and one add, which may be faster than one multiply
  - Note similarity with efficient exponentiation method

Folding Jumps to Jumps

- A jump to an unconditional jump can copy the target address

```
        JNE lab1
         ...
  lab1: JMP lab2
```

Can be replaced by:

```
        JNE lab2
```

As a result, lab1 may become dead (unreferenced)

Jump to Return

- A jump to a return can be replaced by a return

```
           JMP lab1
           ...
lab1: RET
```

  - Can be replaced by

```
RET
```

lab1 may become dead code

Usage of Machine idioms

- Use machine specific hardware instruction which may be less costly.

i := i + 1

ADD i, #1 $\longrightarrow$ INC i

Loop Optimizations

- Most important set of optimizations
  - Programs are likely to spend more time in loops
- Presumption: Loop has been identified
- Optimizations:
  - Loop invariant code removal
  - Induction variable strength reduction
  - Induction variable reduction

LOOP OPTIMIZATION

- **Loop interchange**: exchange inner loops with outer loops
- **Loop splitting**: attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range.
  - A useful special case is ***loop peeling*** - simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop.

Loop Optimization

- **Loop fusion**: two adjacent loops would iterate the same number of times, their bodies can be combined as long as they make no reference to each other's data
- **Loop fission**: break a loop into multiple loops over the same index range but each taking only a part of the loop's body.
- **Loop unrolling**: duplicates the body of the loop multiple times

An Example

Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

Algebraic simplification:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

Algebraic simplification:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

An Example

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example

Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

Dead code elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

Dead code elimination:

```
a := x * x


f := a + a
g := 6 * f
```

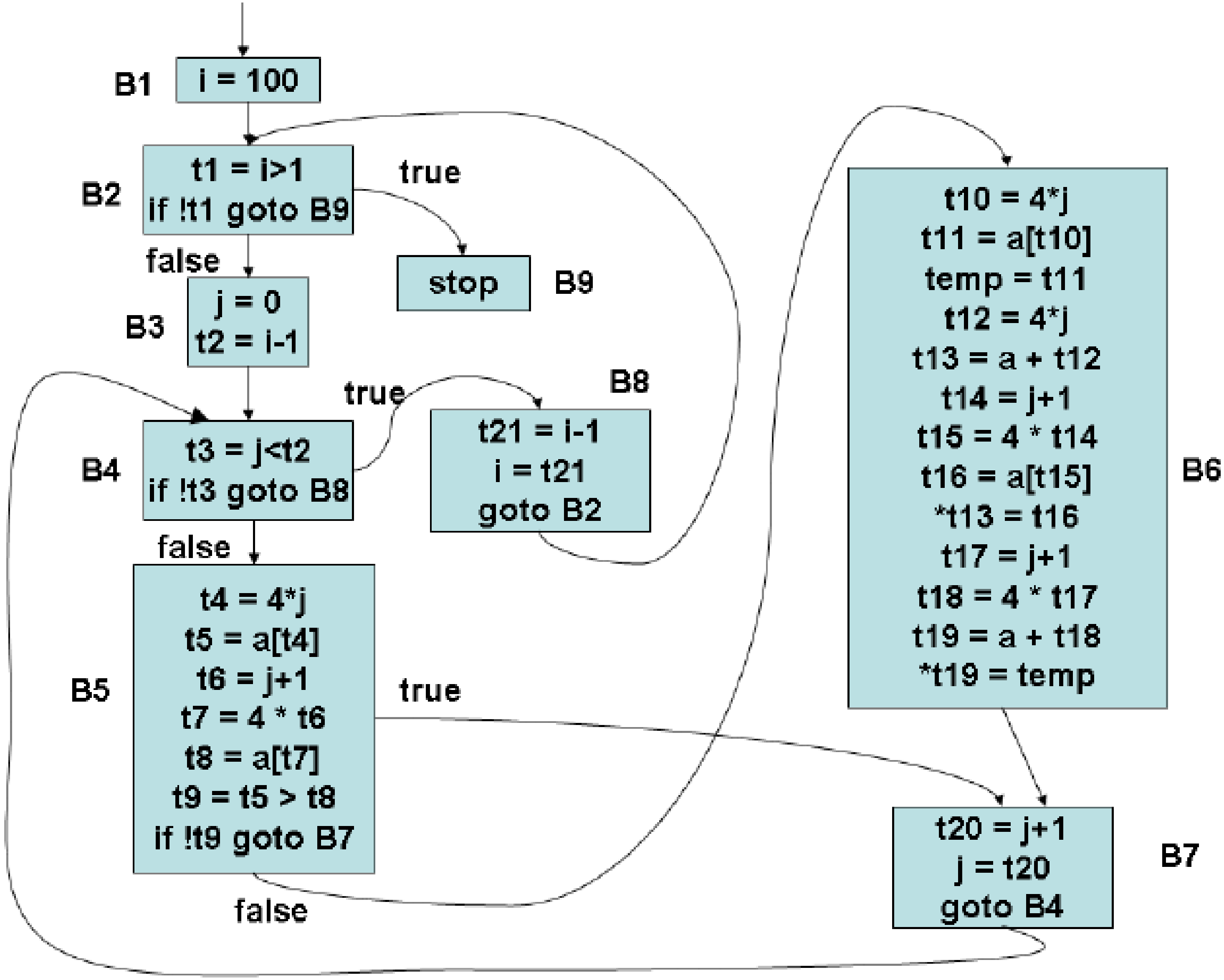
This is the final form

Bubble Sort Running Example

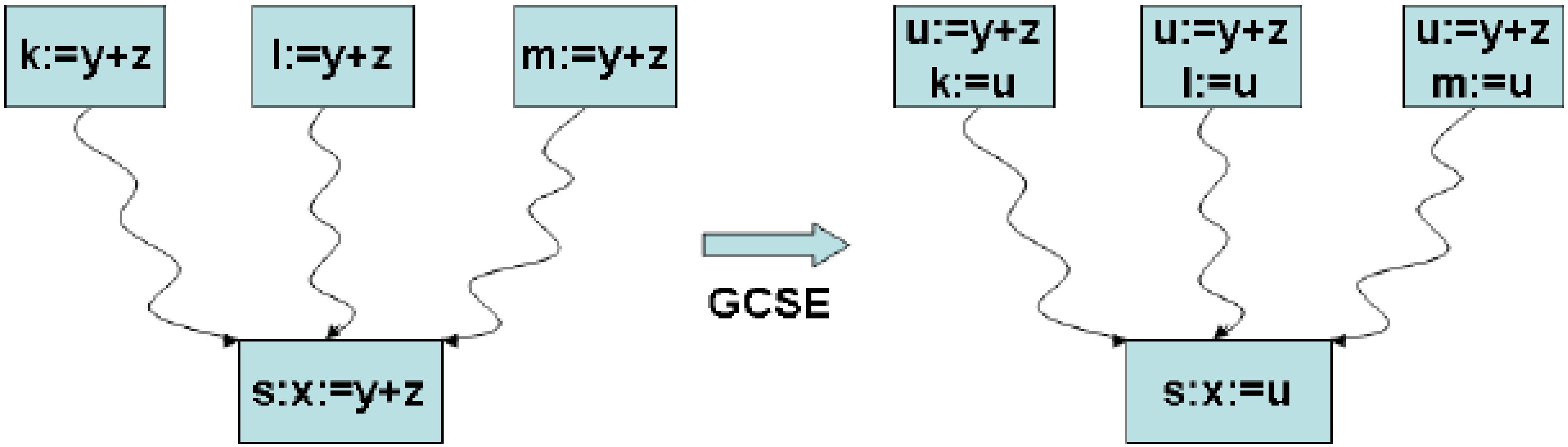
```
for (i=100; i>1; i--) {
    for (j=0; j<i-1; j++) {
        if (a[j] > a[j+1]) {
            temp = a[j];
            a[j+1] = a[j];
            a[j] = temp;
        }
    }
}
```

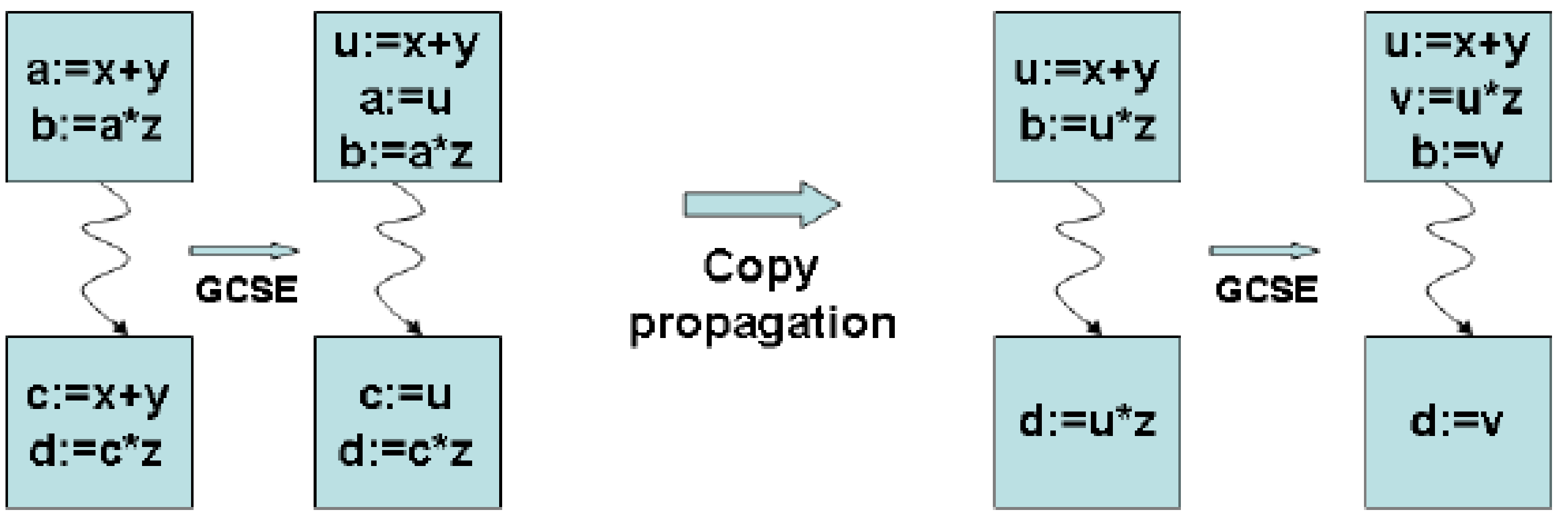
- int a[100]
- array a runs from 0 to 99
- No special jump out if array is already sorted

Control Flow Graph of Bubble Sort

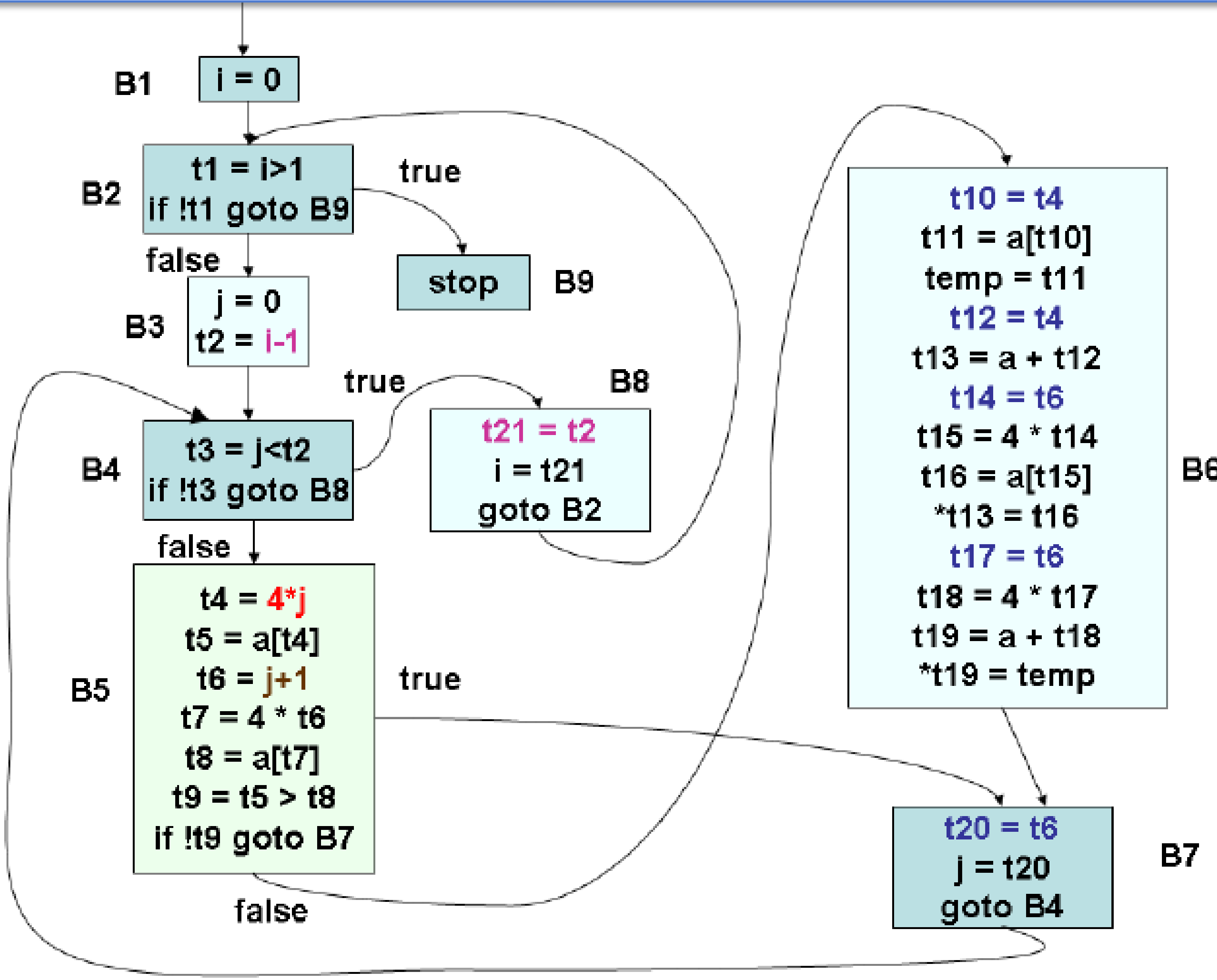


Global common sub-expression elimination Conceptual Example

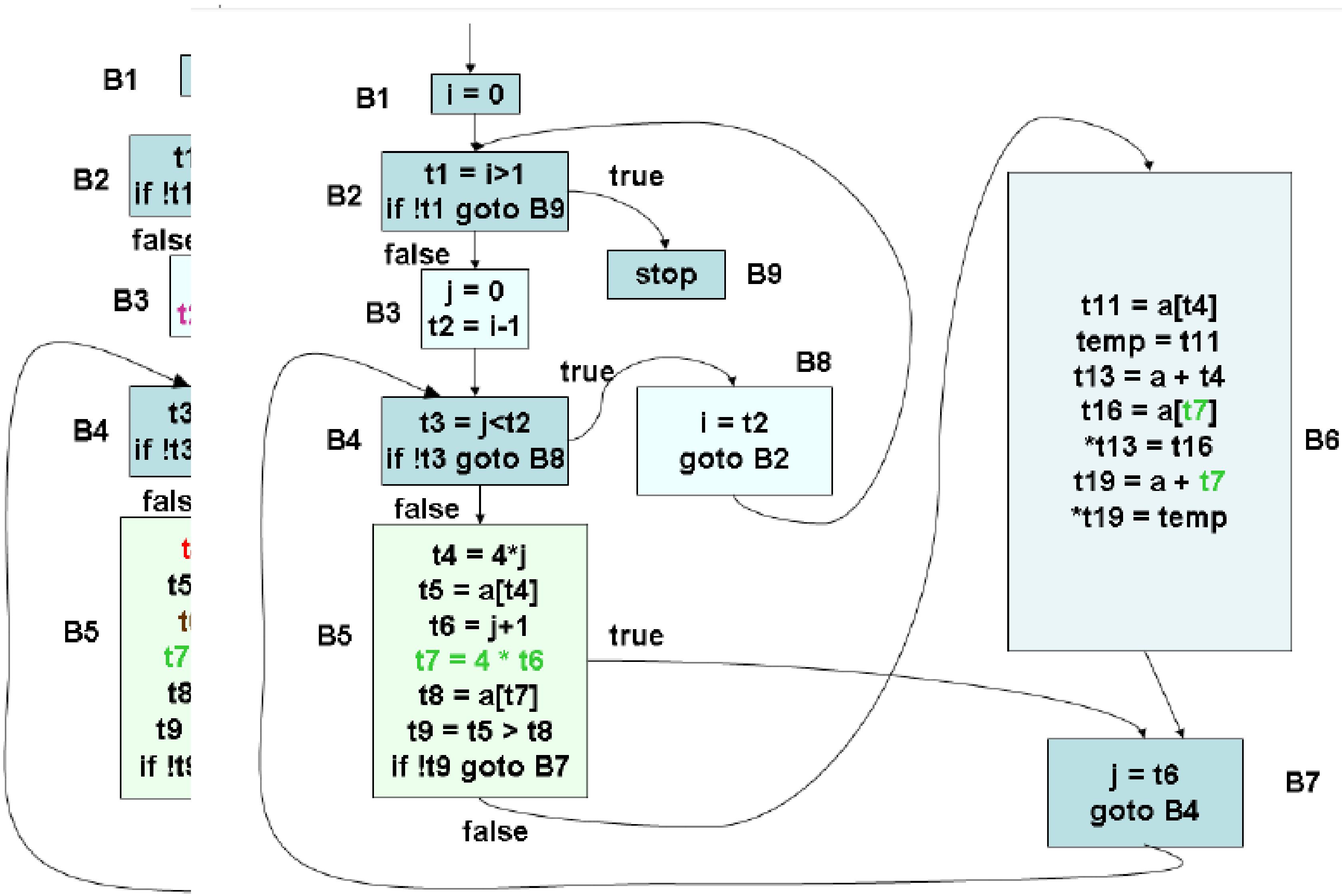




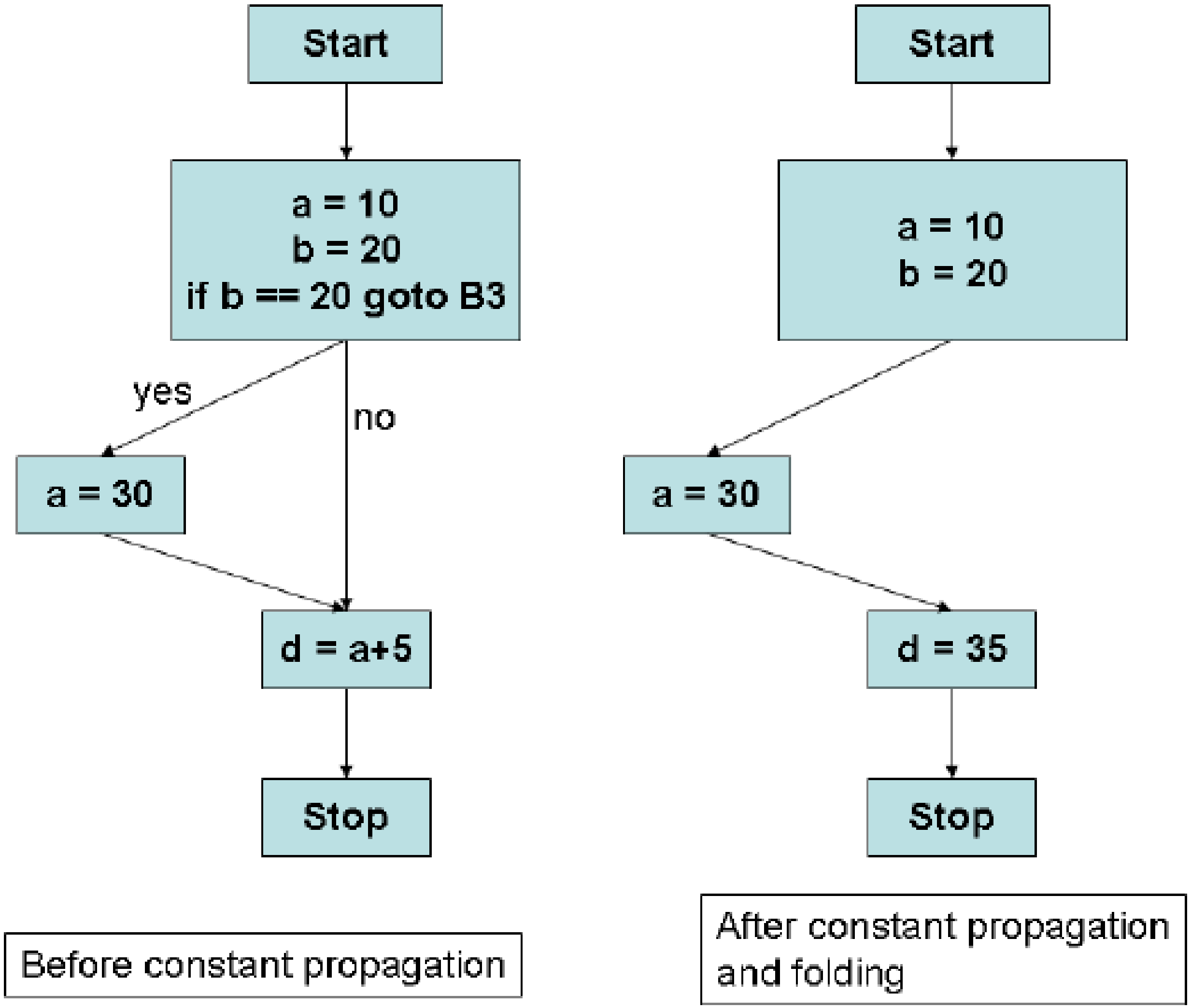
Global Common Sub-expression Elimination Example



Copy Propagation on Running Example



Constant Propagation and Folding Example



Loop Invariant Code motion

```
      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

Strength Reduction

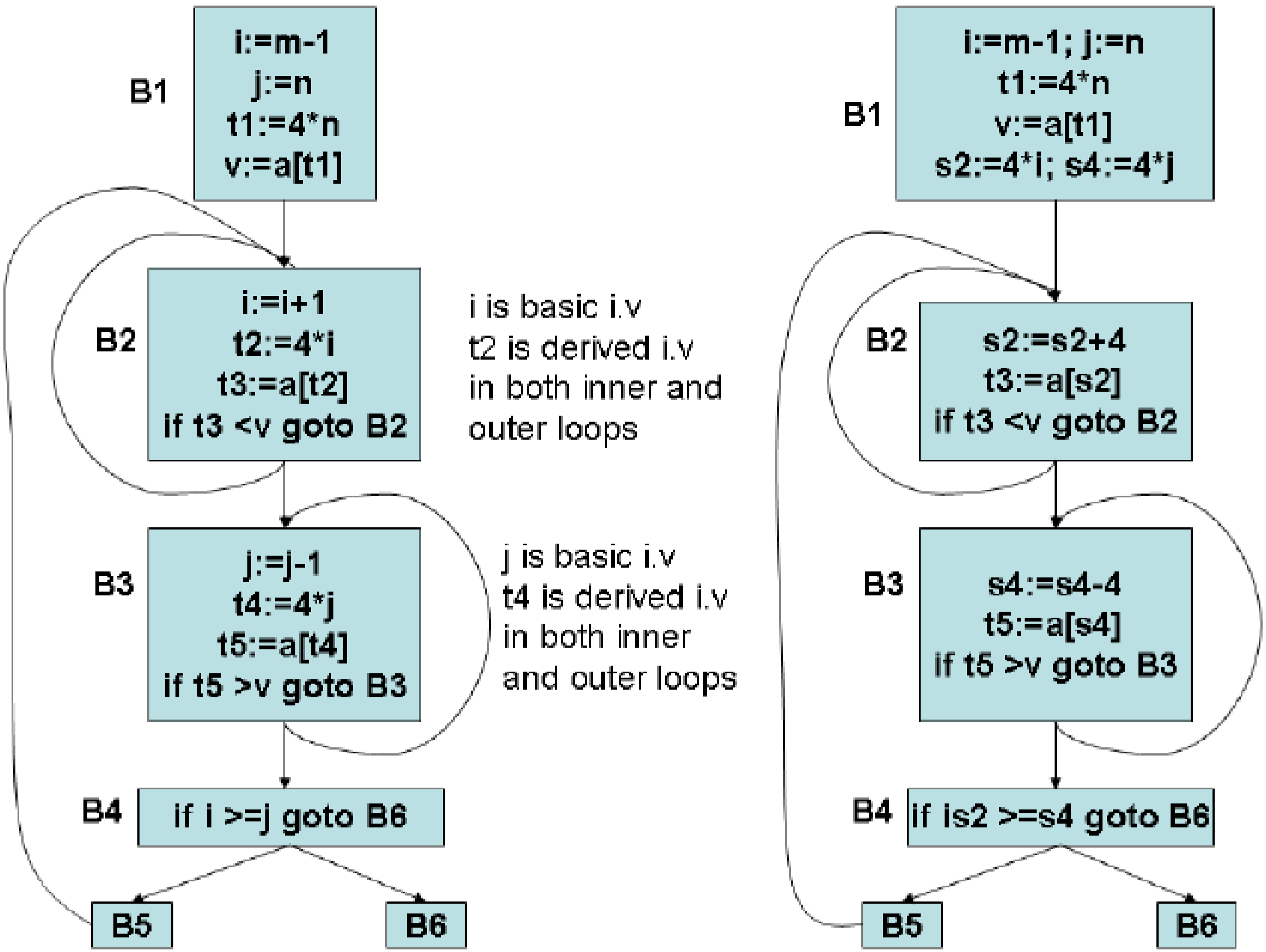
```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

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 and Strength Reduction



Any Questions?