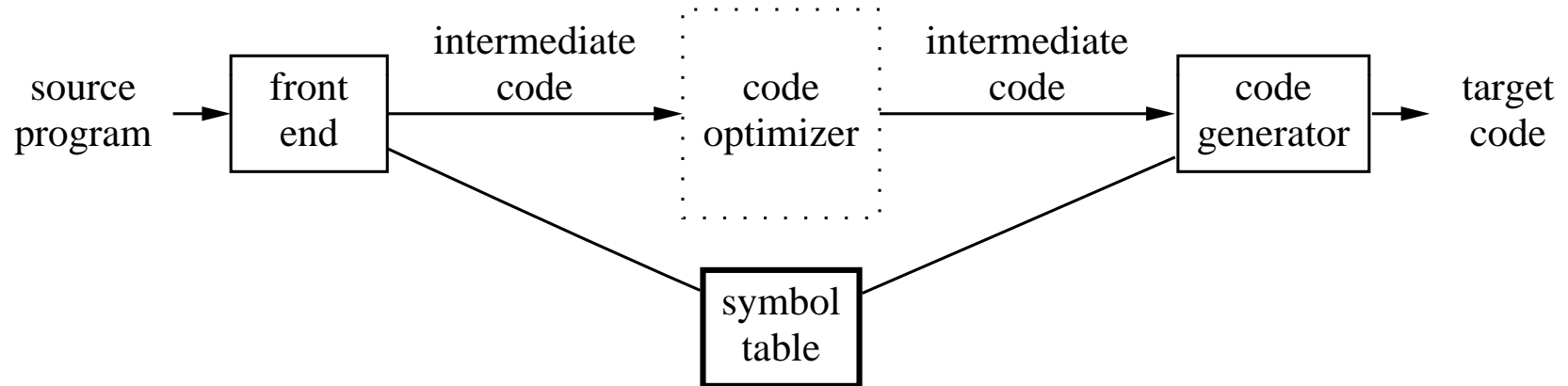


# Code Optimization

# Code Optimization and Phases



# Optimization

- Code produced by standard algorithms can often be made to run faster, take less space or both
- These improvements are achieved through transformations called optimization
- Compilers that apply these transformations are called optimizing compilers
- It is especially important to optimize frequently executed parts of a program
  - The 90/10 rule
  - Profiling can be helpful, but compilers do not typically have sample input data
  - Inner loops tend to be good candidates for optimization

# Criteria for Transformations

- A transformation must preserve the meaning of a program
  - Can not change the output produced for any input
  - Can not introduce an error
- Transformations should, on average, speed up programs
- Transformations should be worth the effort

# Beyond Optimizing Compilers

- Really improvements can be made at various phases
- Source code:
  - Algorithmic transformations can produce spectacular improvements
  - Profiling can be helpful to focus a programmer's attention on important code
- Intermediate code:
  - Compiler can improve loops, procedure calls, and address calculations
  - Typically only optimizing compilers include this phase
- Target code:
  - Compilers can use registers efficiently
  - Peephole transformation can be applied

# Peephole Optimizations

- A simple technique for locally improving target code (can also be applied to intermediate code)
- The peephole is a small, moving window on the target program
- Each improvement replaces the instructions of the peephole with a shorter or faster sequence
- Each improvement may create opportunities for additional improvements
- Repeated passes may be necessary

# Redundant-Instruction Elimination

- Redundant loads and stores:

```
(1) MOV R0, a
```

```
(2) MOV a, R0
```

- Unreachable code:

```
#define debug 0
```

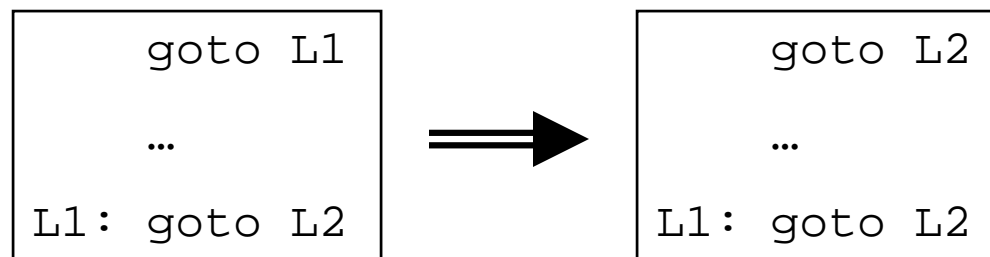
```
if (debug) {
```

```
    /* print debugging information */
```

```
}
```

# Flow-of-Control Optimizations

- Jumps to jumps, jumps to conditional jumps, and conditional jumps to jumps are not necessary
- Jumps to jumps example:
  - The following replacement is valid:



- If there are no other jumps to L1 and L1 is preceded by an unconditional jump, the statement at L1 can be eliminated
- Jumps to conditional jumps and conditional jumps to jumps lead to similar transformations



# Other Peephole Optimizations

- A few algebraic identities that occur frequently (such as  $x := x + 0$  or  $x := x * 1$ ) can be eliminated
- Reduction in strength replaces expensive operations with cheaper ones
  - Calculating  $x * x$  is likely much cheaper than  $x^2$  using an exponentiation routine
  - It may be cheaper to implement  $x * 5$  as  $x * 4 + x$
- Some machines may have hardware instructions to implement certain specific operations efficiently
  - For example, auto-increment may be cheaper than a straight-forward  $x := x + 1$
  - Auto-increment and auto-decrement are also useful when pushing into or popping off of a stack

# Optimizing Intermediate Code

- This phase is generally only included in optimizing compilers
- Offers the following advantages:
  - Operations needed to implement high-level constructs are made explicit (i.e. address calculations)
  - Intermediate code is independent of target machine; code generator can be replaced for different machine
- We are assuming intermediate code uses three-address instructions

# QuickSort in C

```
void quicksort(int m, int n) {
    int i, j, v, x;

    if (n <= m) return;

    /* Start of partition code */
    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;
    /* End of partition code */

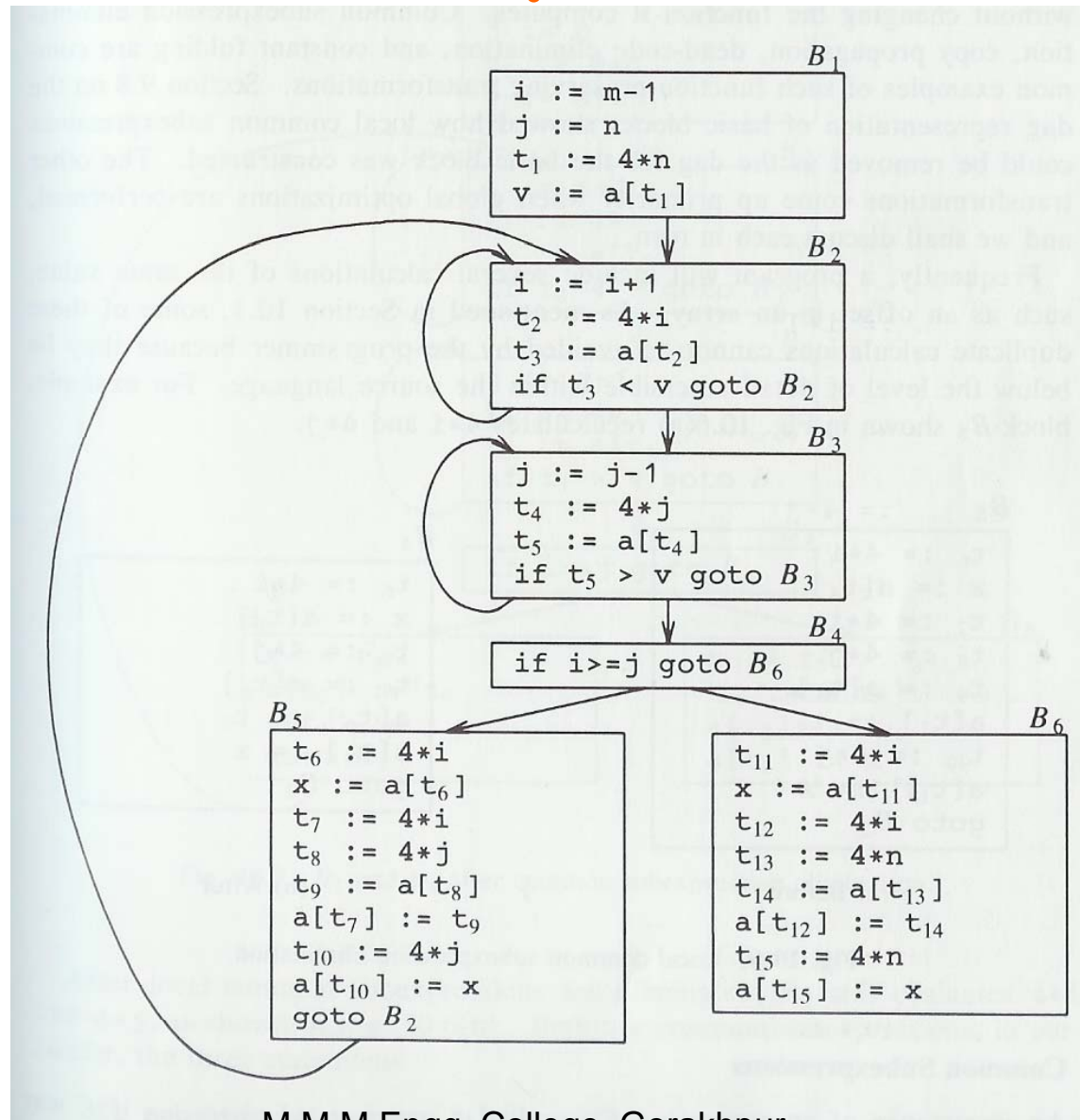
    quicksort(m, j); quicksort(i+1, n);
}
```

# Partition in Three-Address Code

```
(1) i := m-1
(2) j := n
(3) t1 := 4*n
(4) v := a[t1]
(5) i := i+1
(6) t2 := 4*i
(7) t3 := a[t2]
(8) if t3 < v goto (5)
(9) j := j-1
(10) t4 := 4*j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*i
(15) x := a[t6]
```

```
(16) t7 := 4*i
(17) t8 := 4*j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4*j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4*i
(24) x := a[t11]
(25) t12 := 4*i
(26) t13 := 4*n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4*n
(30) a[t15] := x
```

# Partition Flow Graph



# Local vs. Global Transformations

- Local transformations involve statements within a single basic block
- All other transformations are called global transformations
- Local transformations are generally performed first
- Many types of transformations can be performed either locally or globally

# Code Optimizations

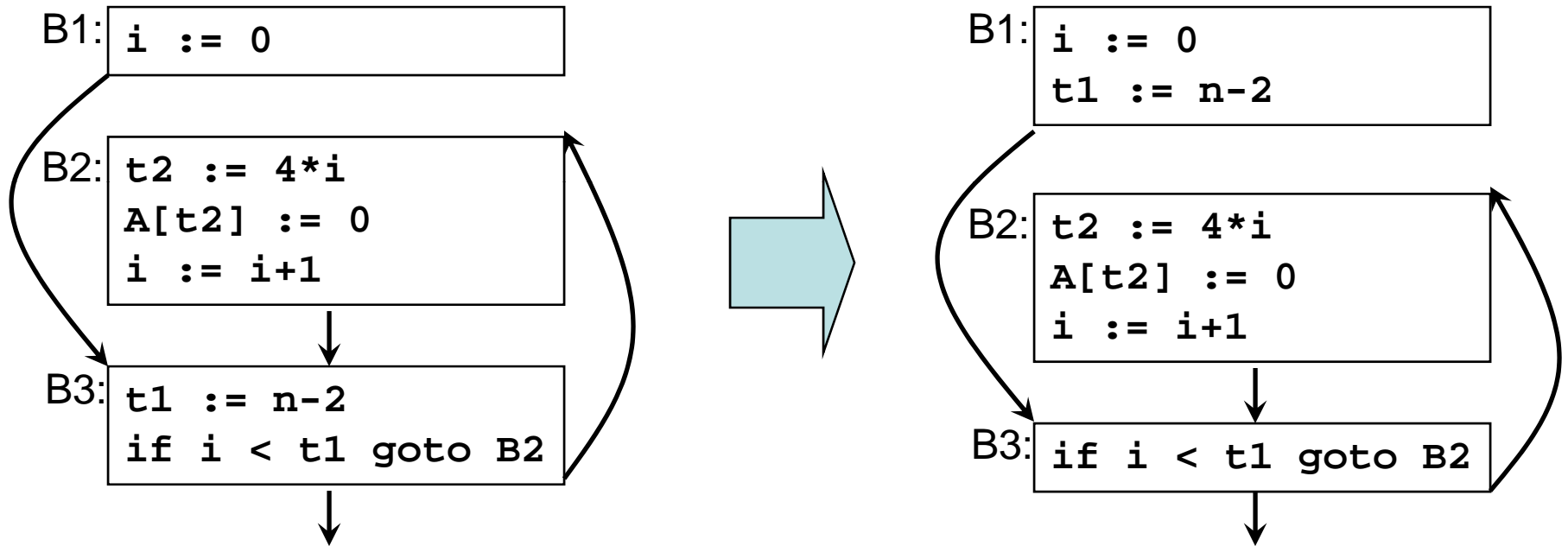
- Local/global common subexpression elimination
- Dead-code elimination
- Instruction reordering
- Constant folding
- Algebraic transformations
- Copy propagation
- *Loop optimizations*

# Loop Optimizations

- Code motion
- Induction variable elimination
- Reduction in strength
- ... lots more

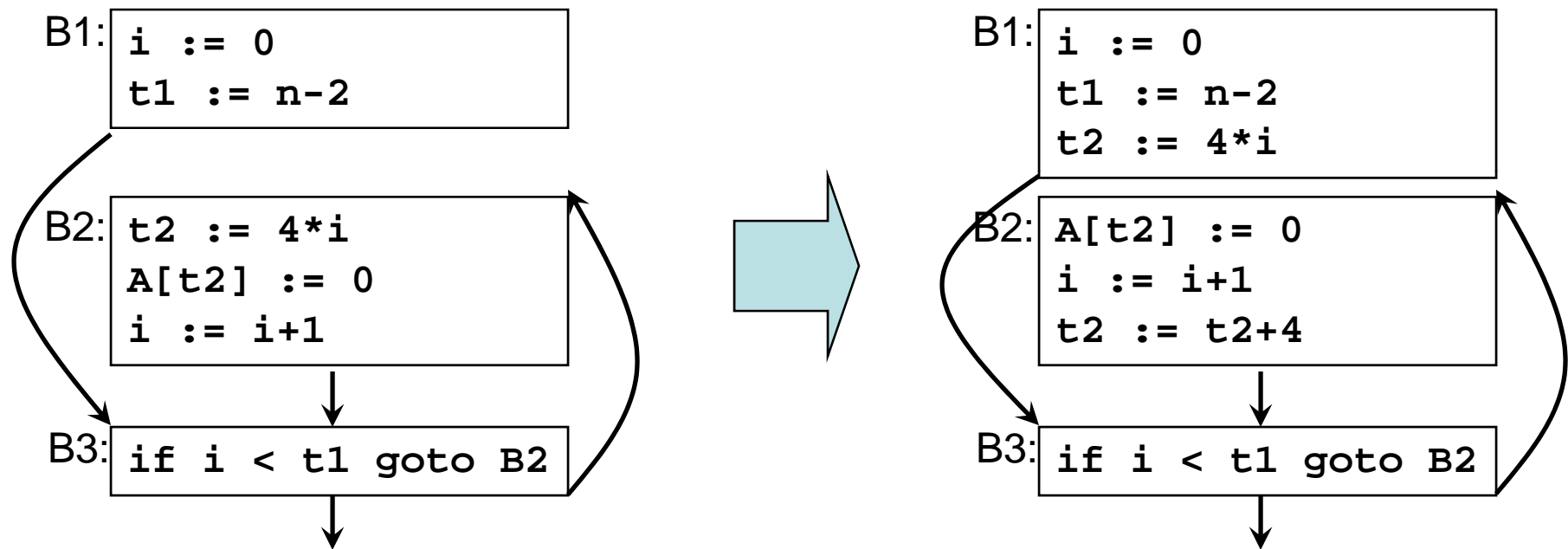


# Code Motion



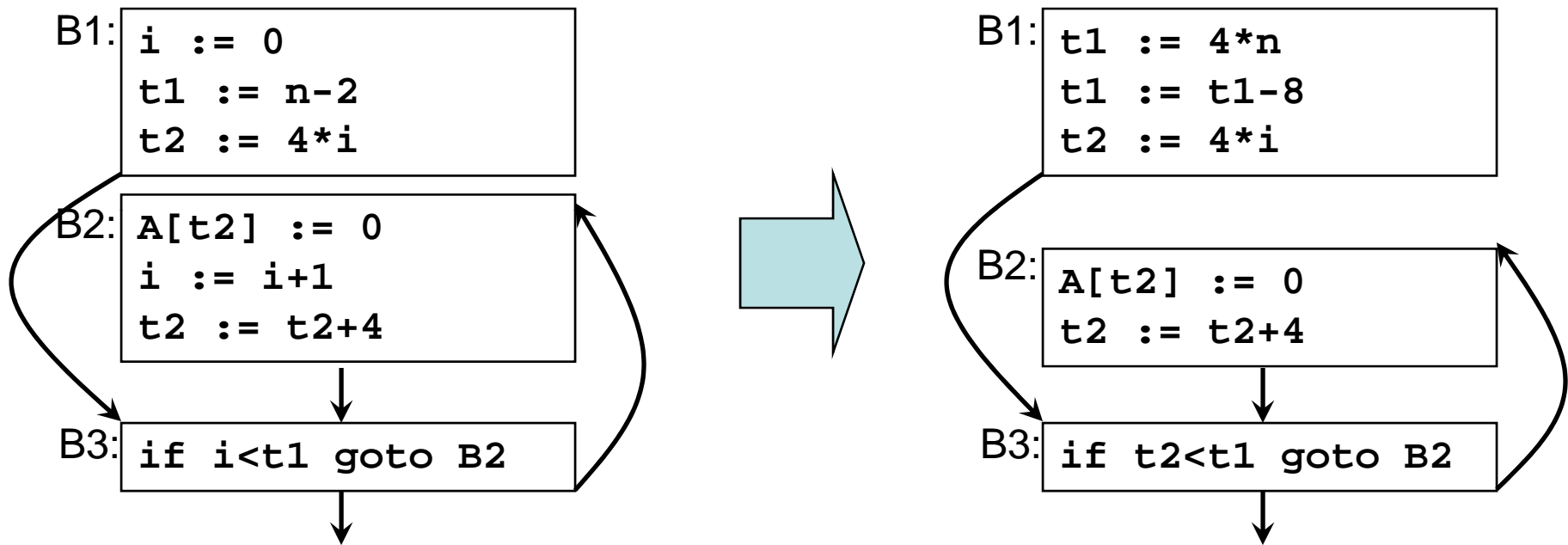
Move *loop-invariant computations* before the loop

# Strength Reduction



Replace expensive computations with *induction variables*

# Reduction Variable Elimination



Replace induction variable in expressions with another

# Common Subexpressions

- $E$  is a common subexpression if:
  - $E$  was previously computed
  - Variables in  $E$  have not changed since previous computation
- Can avoid recomputing  $E$  if previously computed value is still available
- Dags are useful to detect common subexpressions

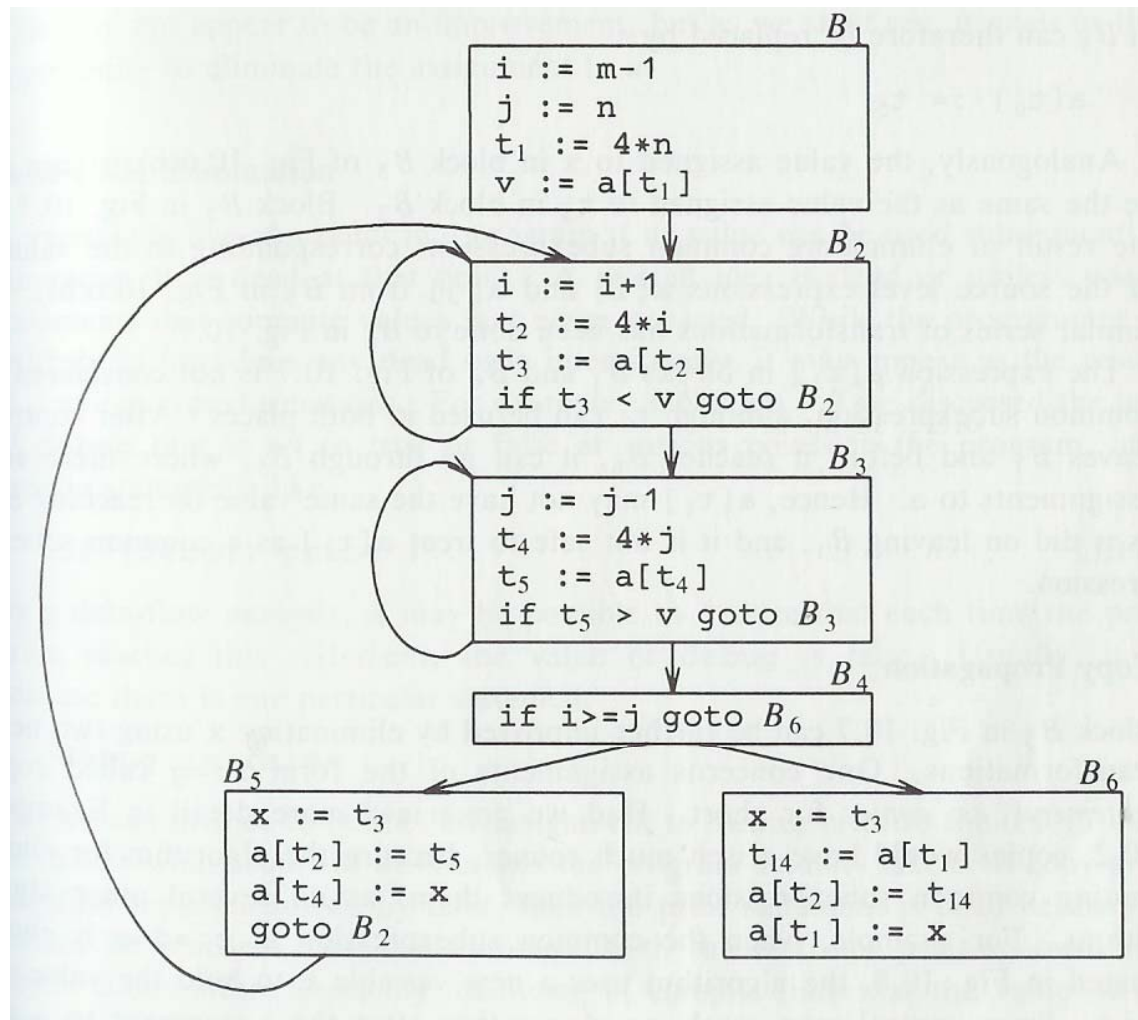
# Local Common Subexpressions

```
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
```



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

# Global Common Subexpressions



# Copy Propagation

- Assignments of the form  $f := g$  are called copy statements (or copies)
- The idea behind copy propagation is to use  $g$  for  $f$  whenever possible after such a statement
- For example, applied to block B5 of the previous flow graph, we obtain:

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

- Copy propagation often turns the copy statement into "dead code"

# Dead-Code Elimination

- Dead code includes code that can never be reached and code that computes a value that never gets used
- Consider: `if (debug) print ...`
  - It can sometimes be deduced at compile time that the value of an expression is constant
  - Then the constant can be used in place of the expression (constant folding)
  - Let's assume a previous statement assigns `debug := false` and value never changes
  - Then the print statement becomes unreachable and can be eliminated
- Consider the example from the previous slide
  - The value of `x` computed by the copy statement never gets used after the copy propagation
  - The copy statement is now dead code and can be eliminated



# Loop Optimizations (1)

- The running time of a program may be improved if we do both of the following:
  - Decrease the number of statements in an inner loop
  - Increase the number of statements in the outer loop
- Code motion moves code outside of a loop
  - For example, consider:

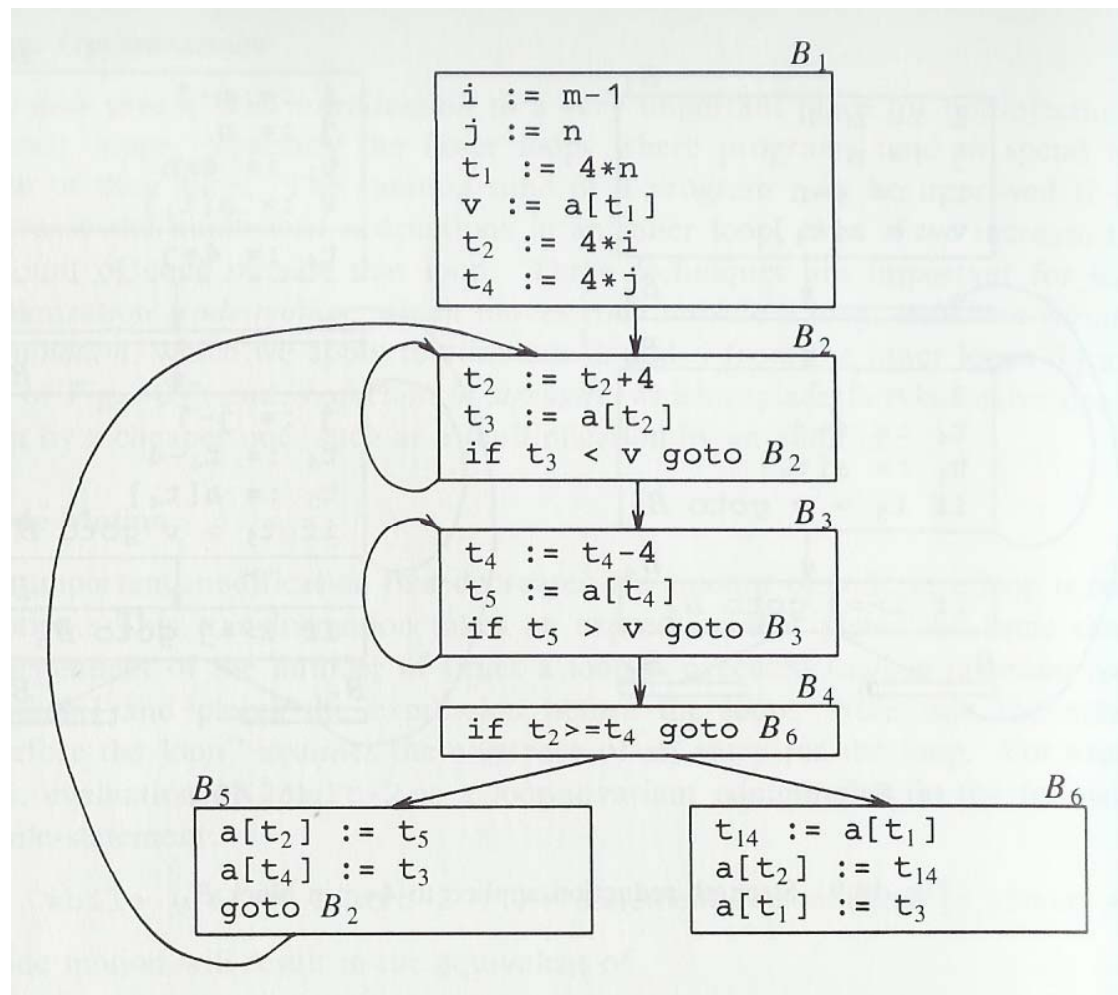
```
while (i <= limit-2) ...
```
  - The result of code motion would be:

```
t = limit - 2  
while (i <= t) ...
```

# Loop Optimizations (2)

- Induction variables: variables that remain in "lock-step"
  - For example, in block B3 of previous flow graph,  $j$  and  $t_4$  are induction variables
  - Induction-variable elimination can sometimes eliminate all but one of a set of induction variables
- Reduction in strength replaces a more expensive operation with a less expensive one
  - For example, in block B3,  $t_4$  decreases by four with every iteration
  - If initialized correctly, can replace multiplication with subtraction
  - Often application of reduction in strength leads to induction-variable elimination
- Methods exist to recognize induction variables and apply appropriate transformations automatically

# Loop Optimization Example



# Example

- Given the following code segment, obtain:
  - The 3AC statements for this computation
  - The basic blocks and the flow graph

```
prod = 0; i = 1;  
do {  
    prod = prod + a[i] * b[i];  
    i = i + 1;  
} while ( i <= 20 );
```

# Example ...contd

- 3AC for the given code segment

(1)  $\text{prod} := 0$

(2)  $i := 1$

(3)  $t1 := 4 * i$

(4)  $t2 := a[t1]$

(5)  $t3 := 4 * i$

(6)  $t4 := b[t3]$

(7)  $t5 := t2 * t4$

(8)  $t6 := \text{prod} + t5$

(9)  $\text{prod} := t6$

(10)  $t7 := i + 1$

(11)  $i := t7$

(12) if  $i \leq 20$  goto (3)

# Example ...contd

## 2 Basic blocks

(1) prod := 0

(2) i := 1

**B1**

(3) t1 := 4 \* i

(4) t2 := a [t1]

(5) t3 := 4 \* i

(6) t4 := b [t3]

(7) t5 := t2 \* t4

(8) t6 := prod + t5

(9) prod := t6

(10) t7 := i + 1

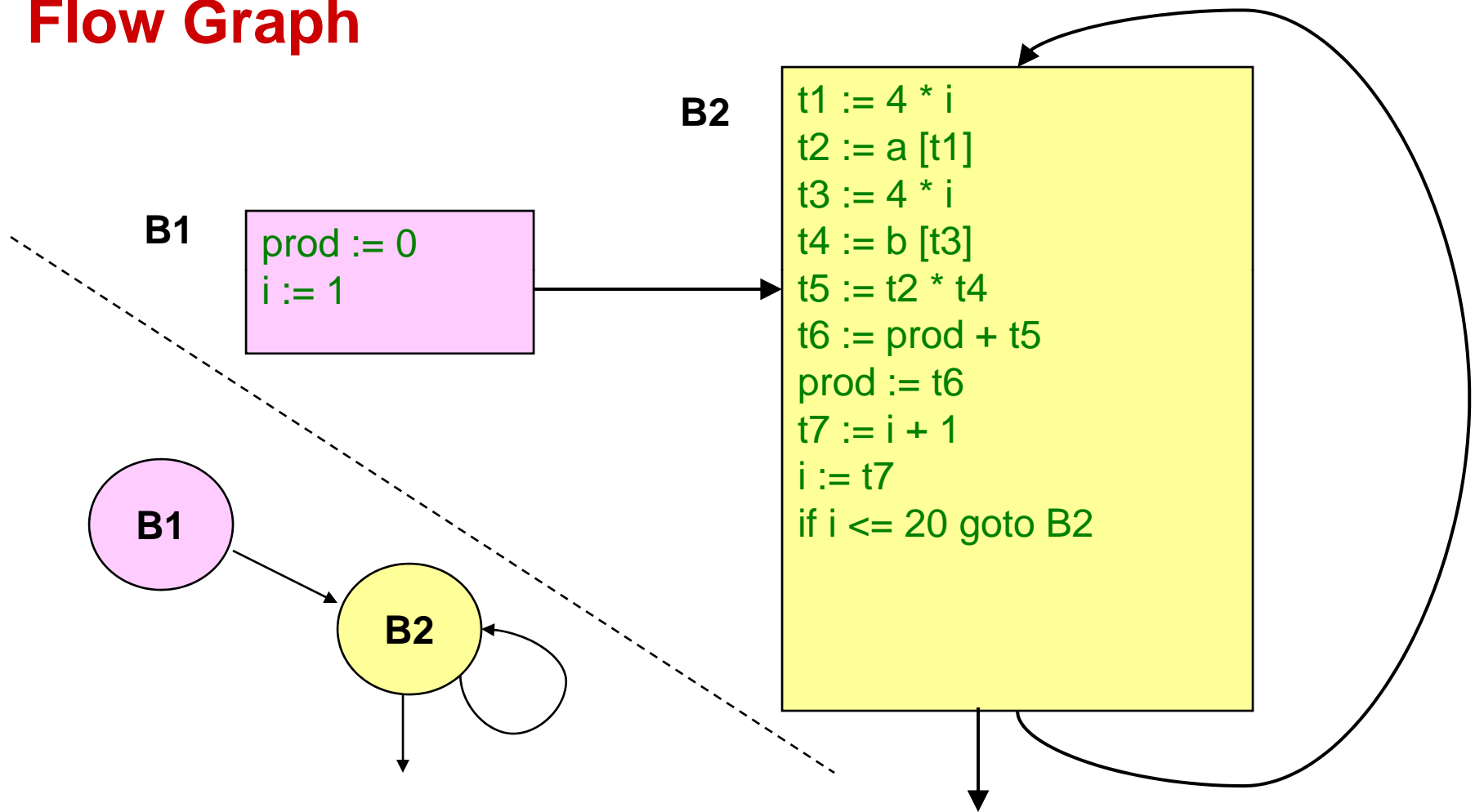
(11) i := t7

(12) if i <= 20 goto (3)

**B2**

# Example ...contd

## Flow Graph



# Implementing a Code Optimizer

- Organization consists of control-flow analysis, then data-flow analysis, and then transformations
- The code generator is applied to the transformed intermediate code
- Details of code optimization are beyond the scope of this course

