Unit 5
Code optimization

RUN-TIME ENVIRONMENT, CODE OPTIMIZATION AND GENERATION

Source Language Issues - Storage Organization - Storage Allocation - Symbol Tables. Principal Sources of Optimization - Optimization of Basic Blocks - Global Optimization - Global Data Flow Analysis - Issues in Design of A Code Generator - A Simple Code Generator Algorithm.

Code optimization

Goals of code optimization:

- remove redundant code without changing the meaning of program.

code optimization : **-** Elimination of unnecessary instructions

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

Objective:

1. The optimization must be correct, it must not, in any way, change the meaning of the program.
2. Optimization should increase the speed and performance of the program.
3. The compilation time must be kept reasonable.
4. The optimization process should not delay the overall compiling process.

Achieved through code transformation while preserving semantics.

- A very hard problem + non-undecidable, i.e., an optimal program cannot be found in most general case.
- Many complex optimization techniques exist.

Trade offs: Effort of implementing a technique + time taken during compilation vs. optimization achieved.

For instance, lexical/semantic/code generation phases require linear time in terms of size of programs, whereas certain optimization techniques may require quadratic or cubic order.

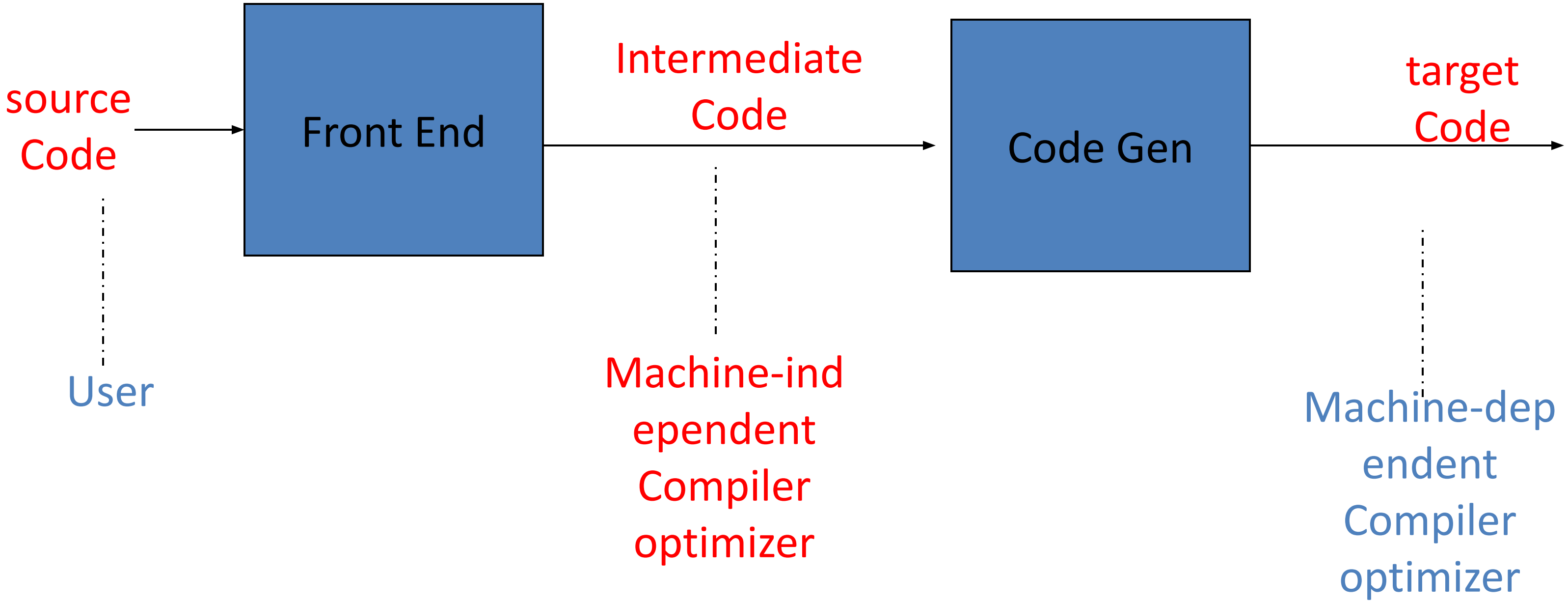
` In many cases simple techniques work well enough.

Issues:

- What are principal sources of optimization?
- When are these optimizations applied?

Code Optimization

- Intermediate Code undergoes various transformations—called Optimizations—to make the resulting code running faster and taking less space
- Optimization never guarantees that the resulting code is the best possible


source Code
Front End
Intermediate Code
Code Gen
target Code
User
Machine-independent Compiler optimizer
Machine-dependent Compiler optimizer

Types of Code Optimization

- **Types of Code Optimization –**The optimization process can be broadly classified into two types :
- **Machine Independent Optimization –** This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.(i.e., they don't take into consideration any properties of the target mach)
- **Machine Dependent Optimization –** Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

Machine Independent Optimization

- The techniques used are a combination of Control-Flow and Data-Flow analysis.

– Control-Flow Analysis. Identifies loops in the flow graph of a program since such loops are usually good candidates for improvement.

– Data-Flow Analysis. Collects information about the way variables are used in a program

Criteria for Code-Improving Transformations

- The best transformations are those that yield the most benefit for the least effort.

1. A transformation must preserve the meaning of a program.
2. A transformation must, on the average, speed up a program by a measurable amount.
3. Avoid code-optimization for programs that run occasionally or during debugging.
4. Remember! Dramatic improvements are usually obtained by improving the source code: The programmer is always responsible in finding the best possible data structures and algorithms for solving a problem

Basic Blocks and Flow Graphs

- The Machine-Independent Code-Optimization phase consists of control-flow and data-flow analysis followed by the application of transformations.
- During Control-Flow analysis, a program is represented as a Flow Graph where:
  - Nodes represent Basic Blocks: Sequence of consecutive statements in which flow-of-control enters at the beginning and leaves at the end without halt or branches;
  - Edges represent the flow of control

Basic Blocks

- Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code.
- These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

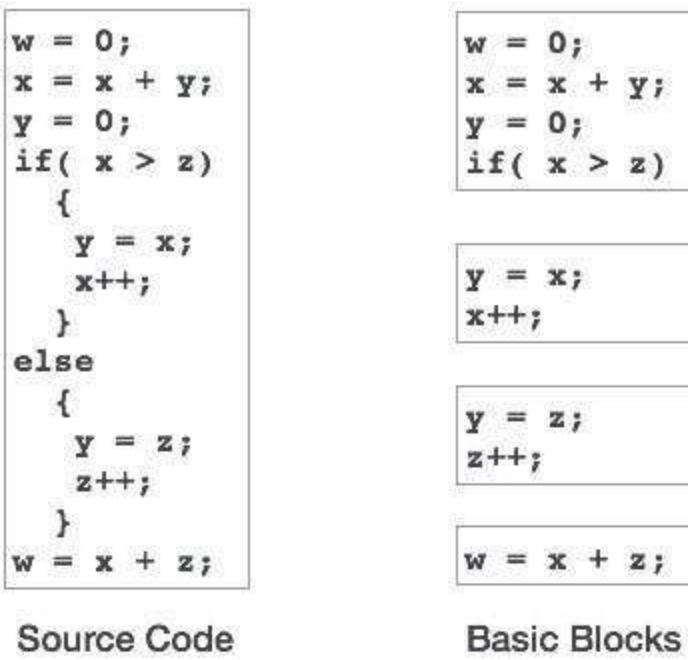
Basic block identification

Basic blocks are important concepts from both code generation and optimization point of view.

Algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
  - First statement of a program.
  - Statements that are target of any branch (conditional/unconditional).
  - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

Example



```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Code

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

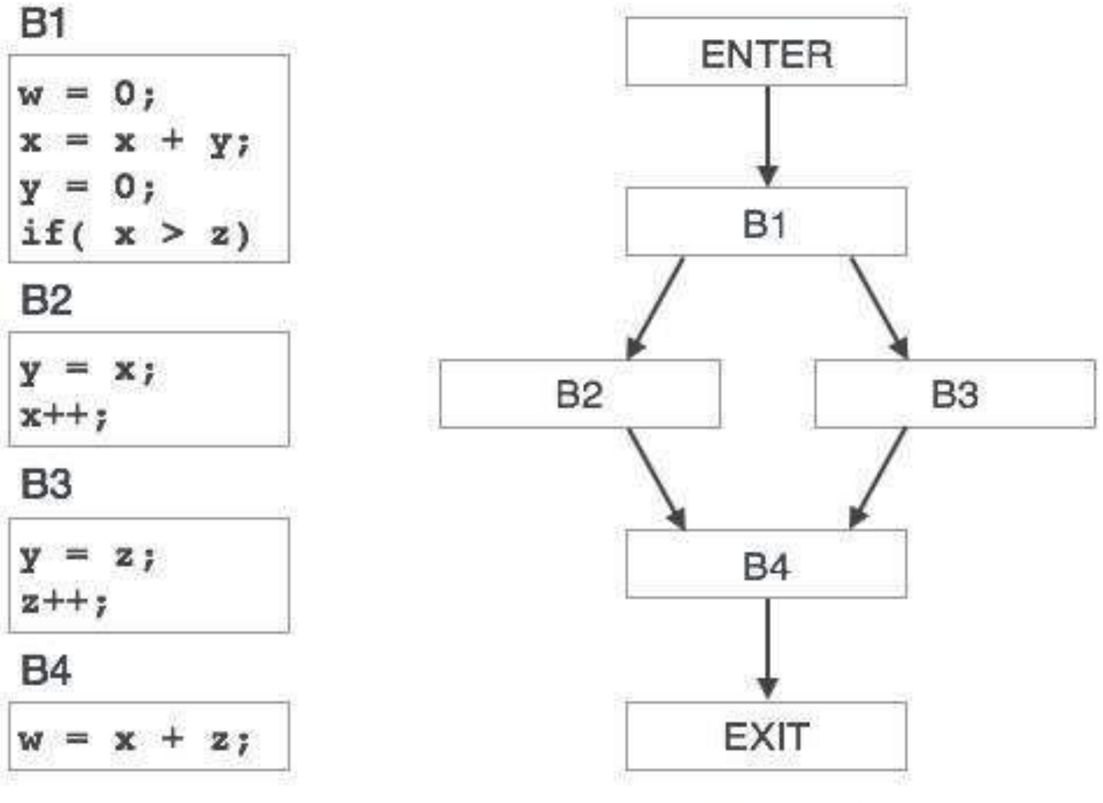
```
y = z;
z++;
```

```
w = x + z;
```

Basic Blocks

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization, by locating any unwanted loops in the program.



```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

```
w = x + z;
```

Basic Blocks

Flow Graph

Principle sources of optimization

- Preserves the semantics of the original program
- Applies relatively low-level semantic transformations

- **Local optimization**: if it can be performed by looking only at the statements in a block.
- **Global optimization** : if it can be performed by looking at on entire program.
- Many optimization Techniques can be performed at both local and global levels.
- The local transformations are usually performed first.

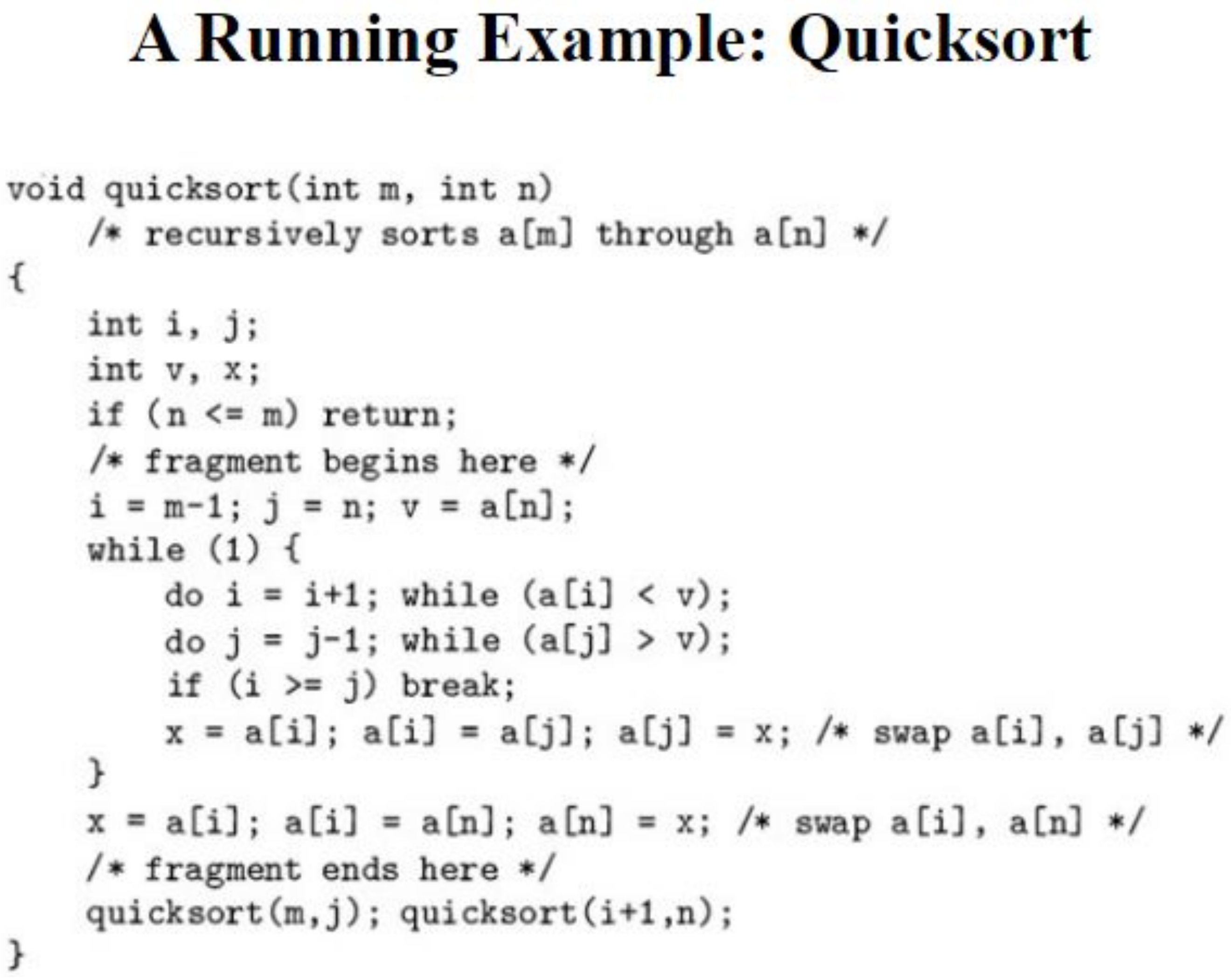
Function(Semantic)-Preserving optimization

- A basic block computes a set of expressions: A number of transformations can be applied to a basic block without changing the expressions computed by the block
- **Function-Preserving optimization**: It can improve the code optimization without changing the function it computes.

Examples include

1. Common sub-expression elimination
2. Copy propagation
3. Dead-code elimination
4. Constant folding
5. Loop Optimization

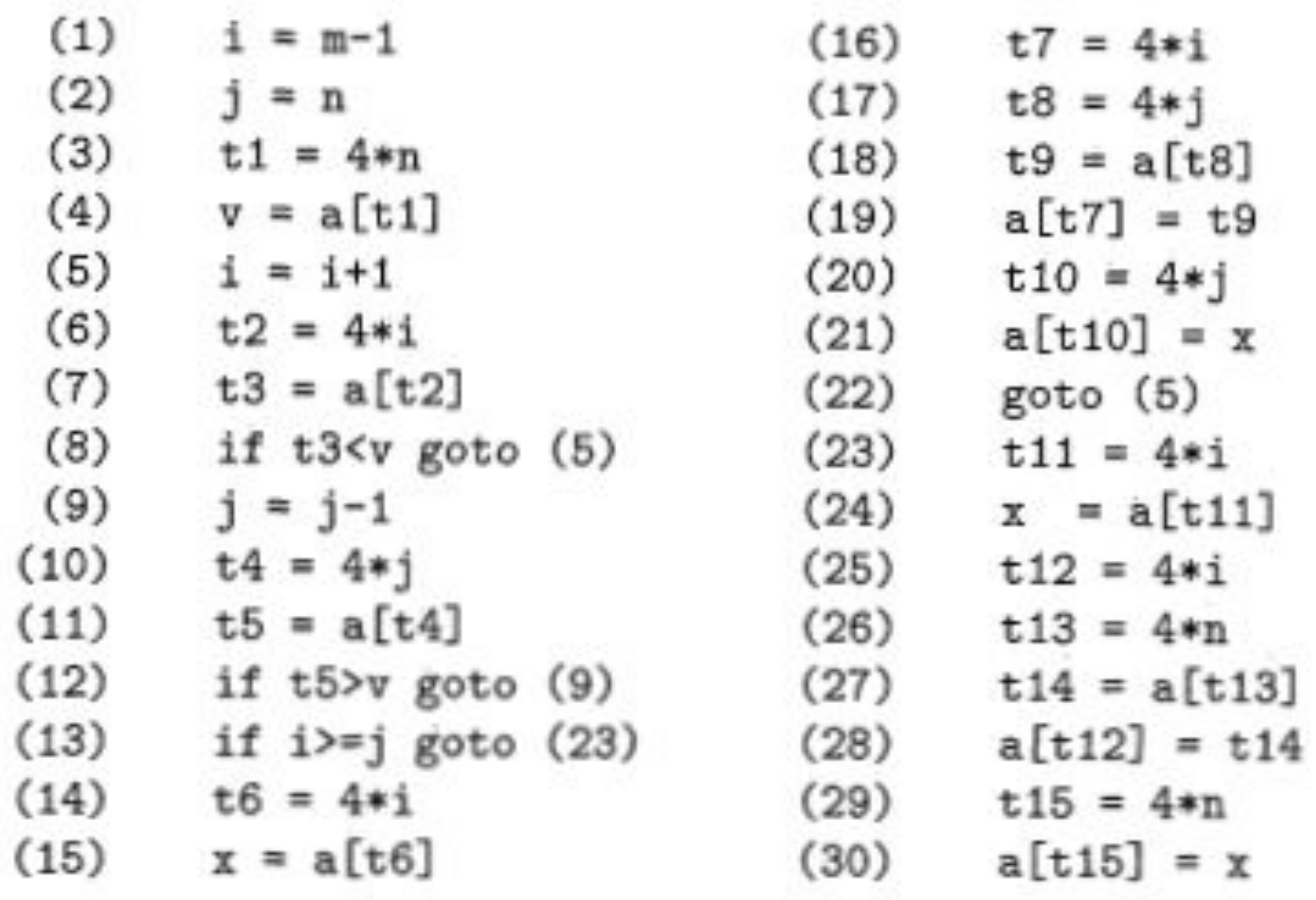
Example: Quicksort



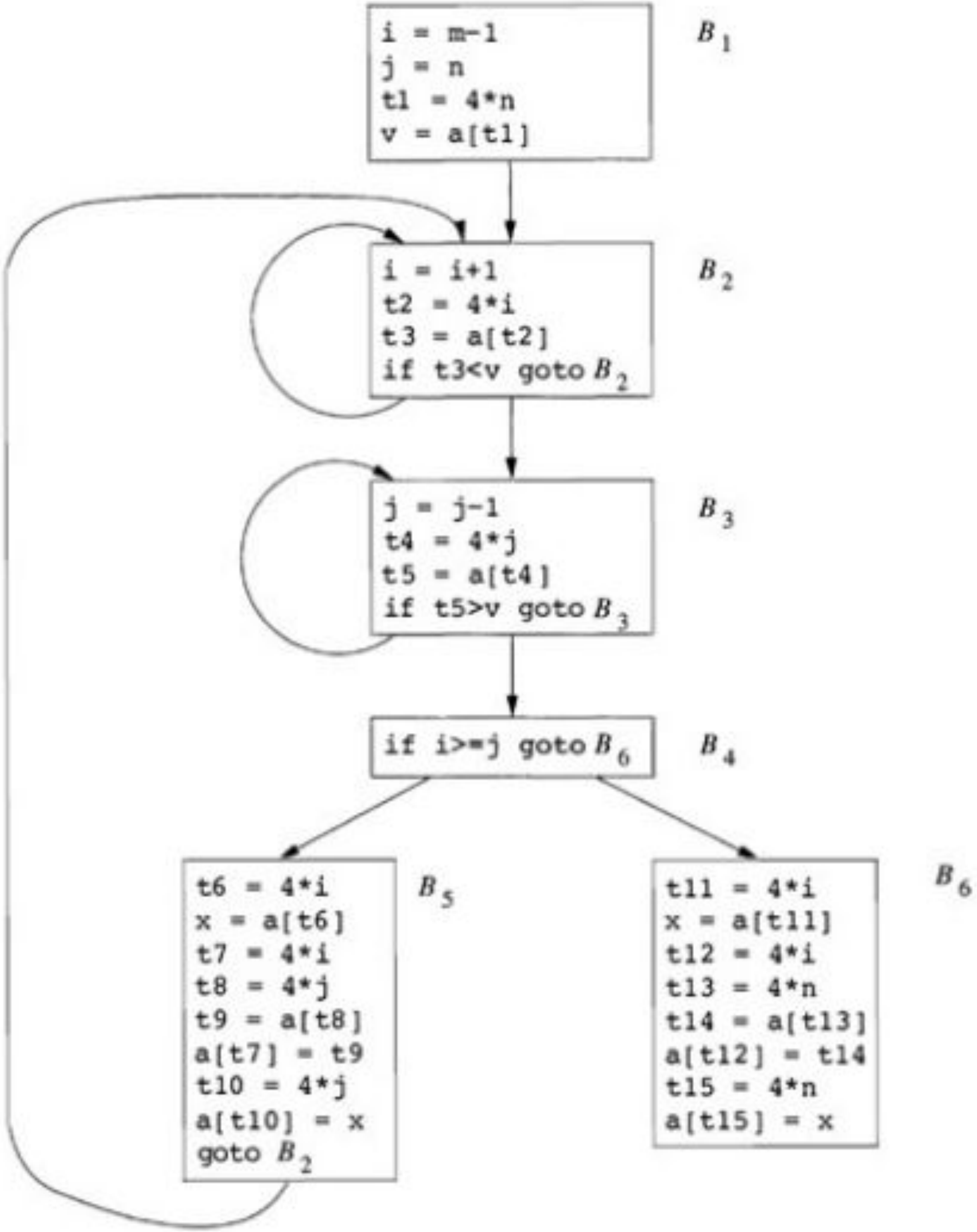
A Running Example: Quicksort

```
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);
}
```

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
```



```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

```
if i>=j goto B6
```

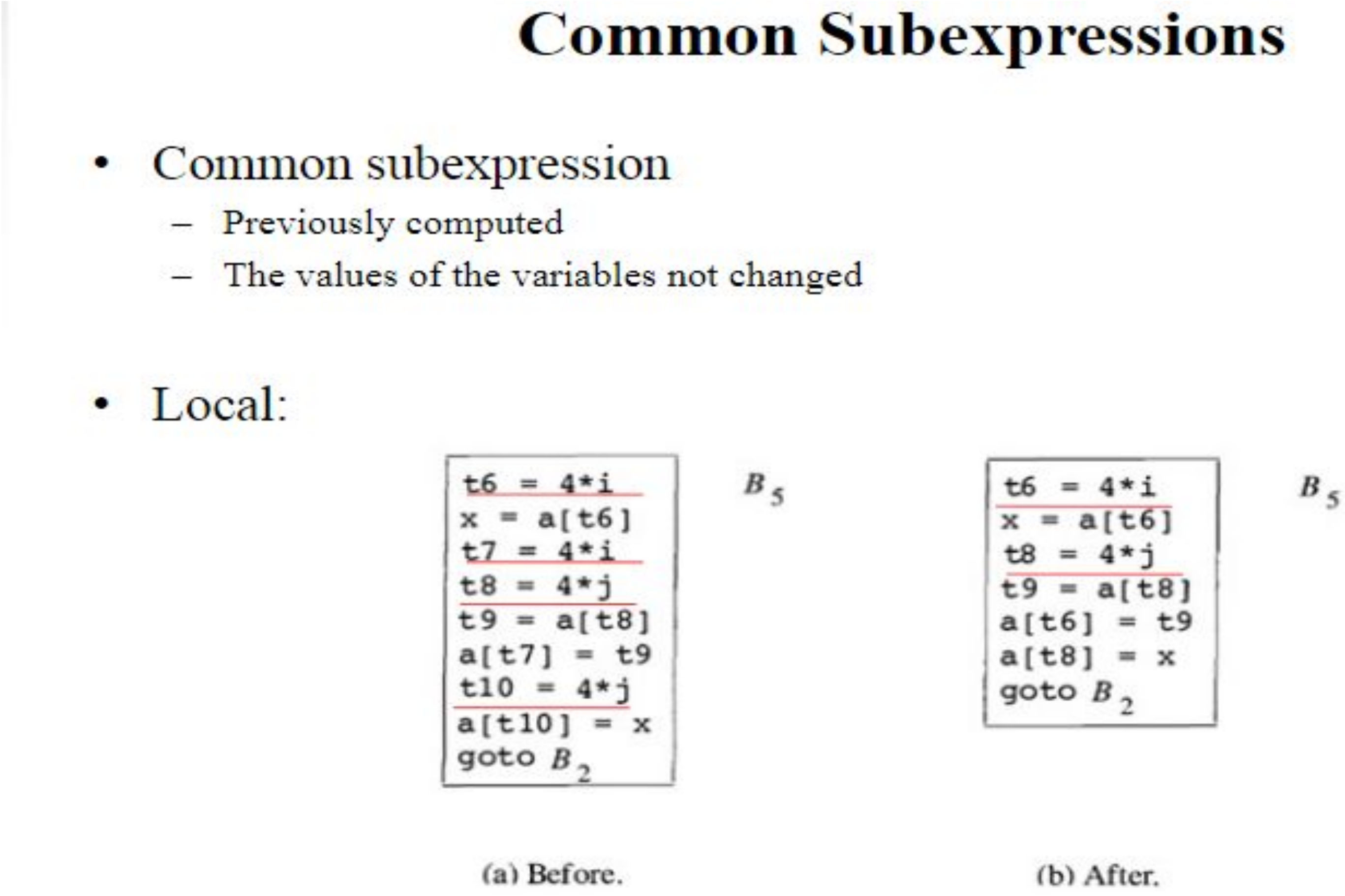
```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

```
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

Common sub-expression elimination

- Frequently a program will include calculations of the same value.
- An occurrence of an expression E is called a common subexpression if E was previously computed, and the values of variables in E have not changed since the previous computation.

Example. Consider the basic block B5. The assignments to both t7 and t10 have common subexpressions and can be eliminated.



Common Subexpressions

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

B_5

```
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
```

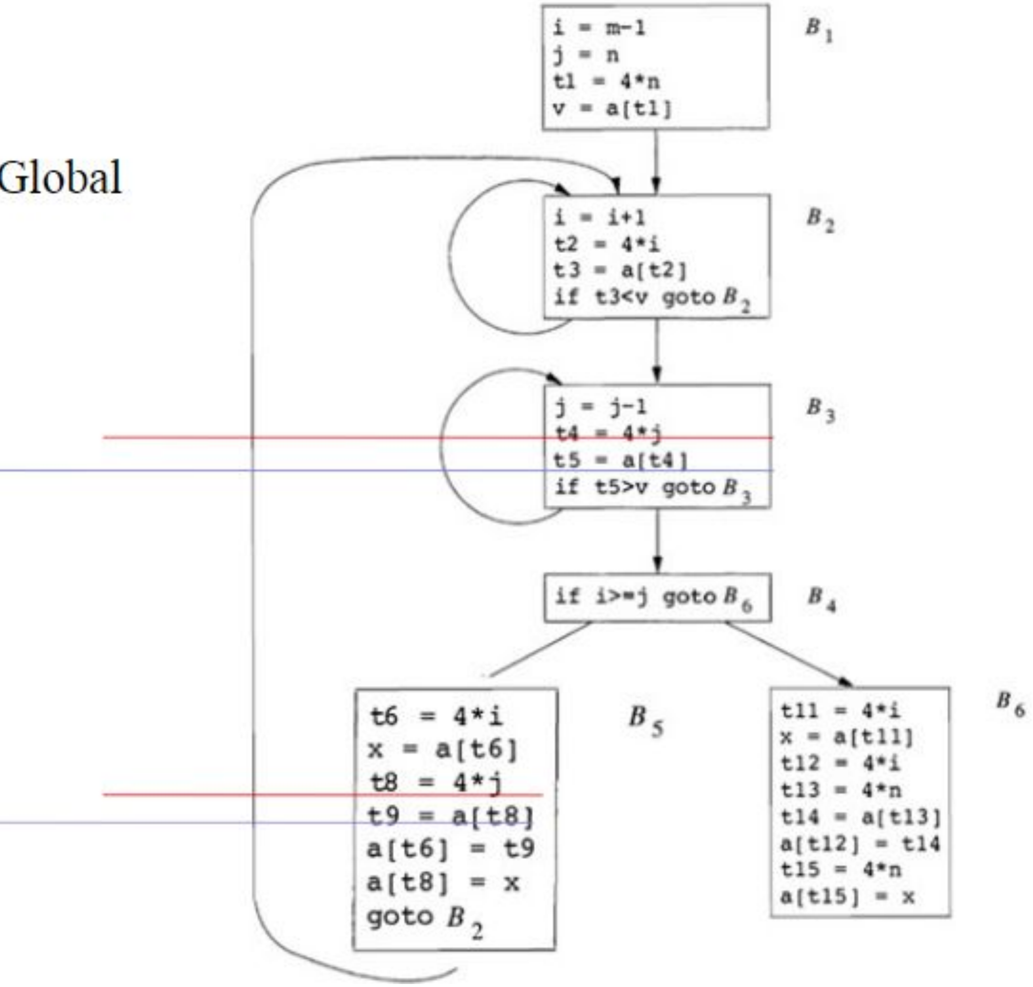
(a) Before.

B_5

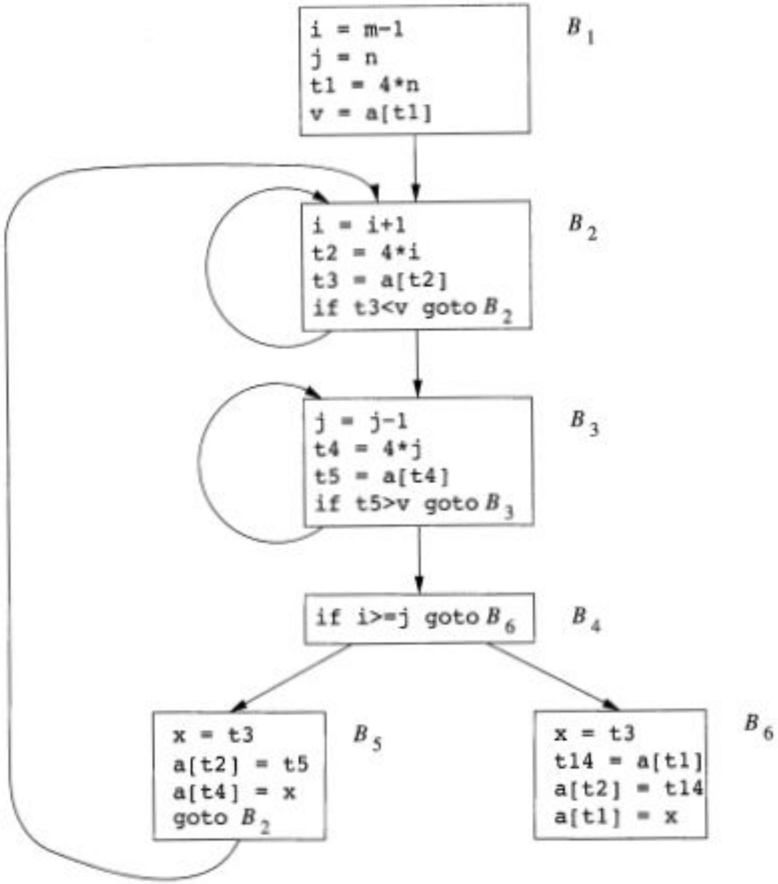
```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

(b) After.

B5 and B6 after common-subexpression elimination



```
B1:
i = m-1
j = n
t1 = 4*n
v = a[t1]

B2:
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2

B3:
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3

B4:
if i>=j goto B6

B5:
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2

B6:
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

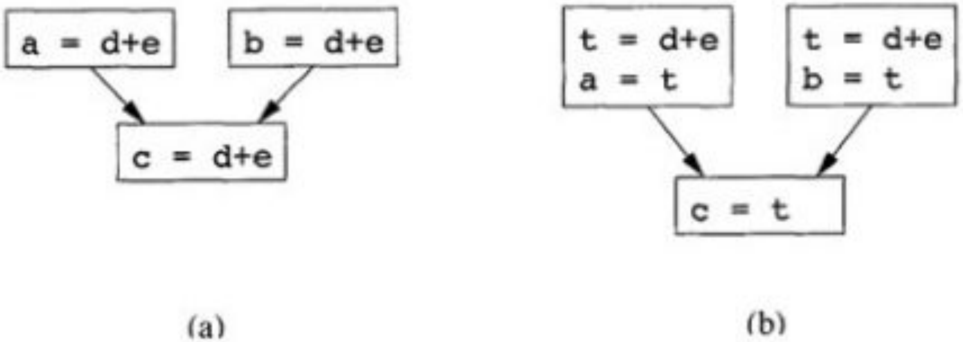


```
B1:
i = m-1
j = n
t1 = 4*n
v = a[t1]

B2:
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2

B3:
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3

B4:
if i>=j goto B6

B5:
x = t3
a[t2] = t5
a[t4] = x
goto B2

B6:
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
```

Copy Propagation

An idea behind this technique is to use g for f whenever possible after the copy of
f := g

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



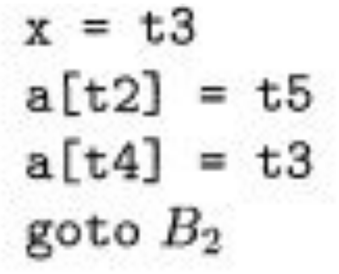
(a)

(b)

In order to eliminate the common subexpression from the statement c = d+e in Fig.(a), we must use a new variable t to hold the value of d + e. The value of variable t, instead of that of the expression d + e, is assigned to c in Fig. (b). Since control may reach c = d+e either after the assignment to a or after the assignment to b, it would be incorrect to replace c = d+e by either c = a or by c = b

Copy Propagation

- The idea behind the copy-propagation transformation is to use v for u, wherever possible after the copy statement u = v.
- For example, the assignment x = t3 in block B5 of is a copy. Copy propagation applied to B5 yields the code. This change may not appear to be an improvement, but, as we shall see in, it gives us the opportunity to eliminate the assignment to x.



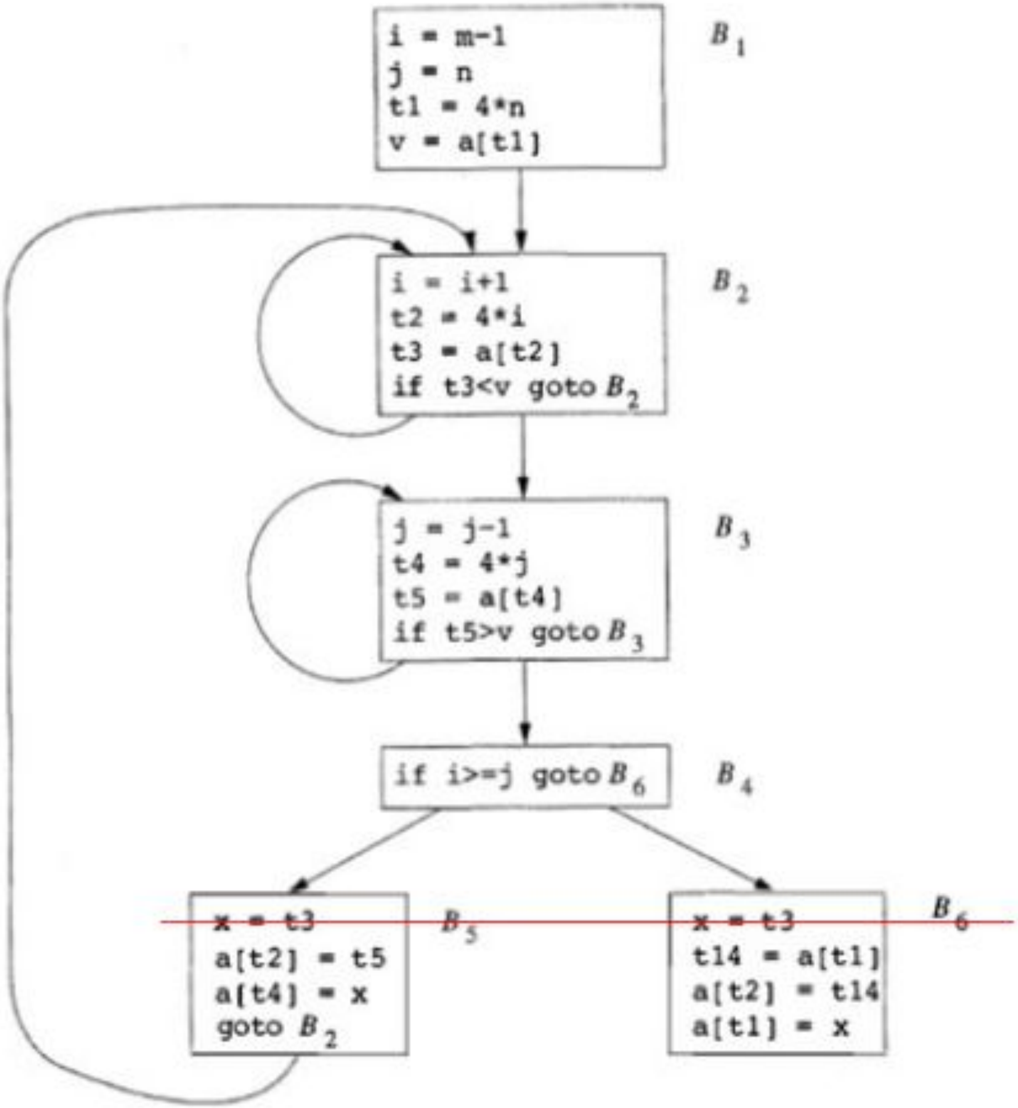
```
x = t3
a[t2] = t5
a[t4] = t3
goto $B_2$
```

Dead-Code Elimination

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
- A related idea is dead (or useless) code - statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.
- If (debug) print …
- Many times, debug := false
- Considering the Block B5 after Copy Propagation we can see that x is never reused all over the code. Thus, x is a dead variable and we can eliminate the assignment **x=t3**

Constant folding

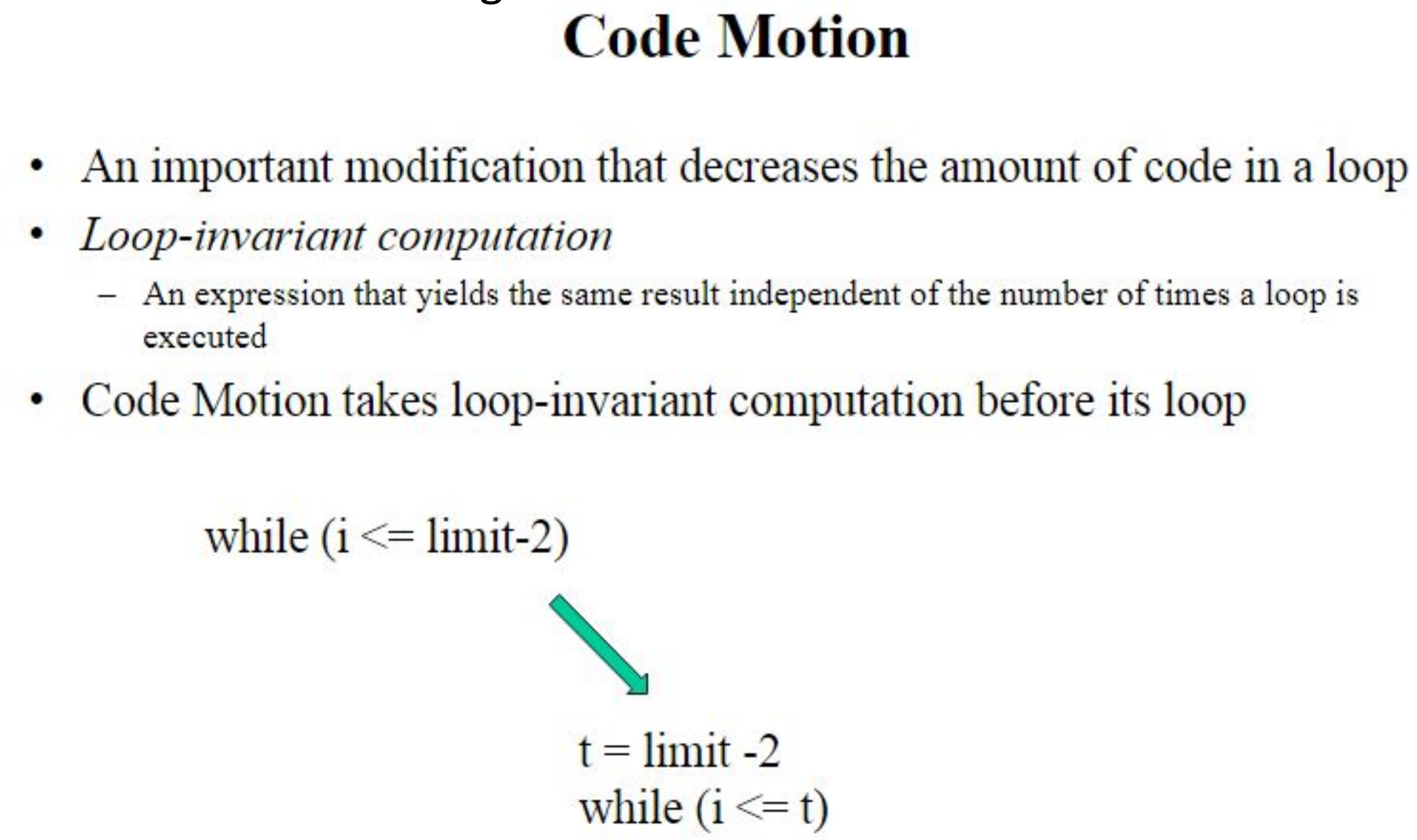
- Deducing at compile time that the value of an expression is a constant and using the constant instead
- Constant Folding is the transformation that substitutes an expression with a constant.
- Constant Folding is useful to discover Dead-Code



```
B1:
i = m-1
j = n
t1 = 4*n
v = a[t1]

B2:
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2

B3:
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3

B4:
if i>=j goto B6

B5:
~~x = t3~~
a[t2] = t5
a[t4] = x
goto B2

B6:
~~x = t3~~
t14 = a[t1]
a[t2] = t14
a[t1] = x
```

Loop Optimization

- Code motion
- Induction-variable elimination
- Reduction in strength



Code Motion

- An important modification that decreases the amount of code in a loop
- *Loop-invariant computation*
  - An expression that yields the same result independent of the number of times a loop is executed
- Code Motion takes loop-invariant computation before its loop

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

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

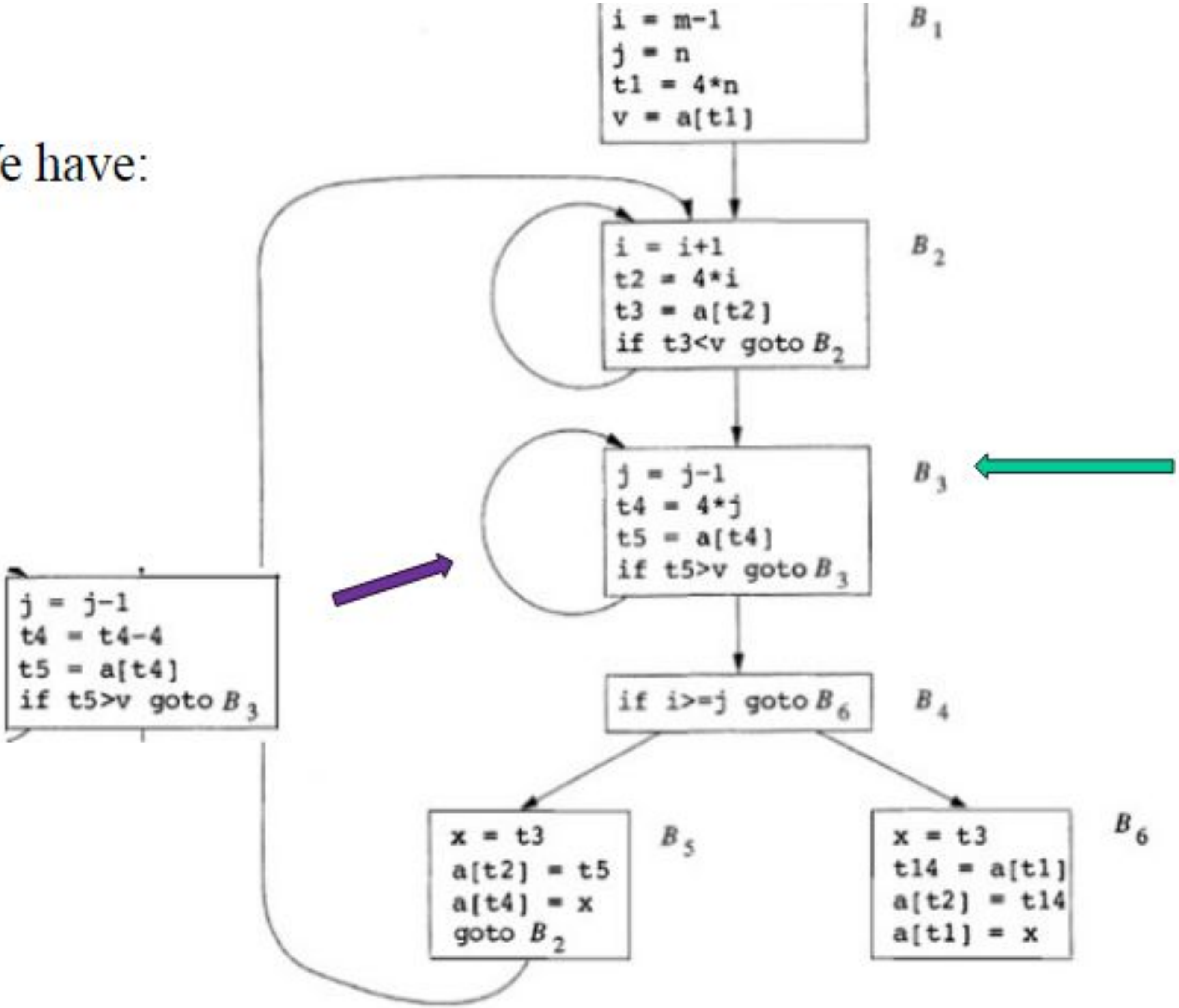
Induction Variables and Reduction in Strength

- Induction variable
  - For an induction variable *x*, there is a positive or negative constant *c* such that each time assigned, its value increases by *c*
- Induction variables can be computed with a single increment (addition or subtraction) per loop iteration

- Strength reduction
  - The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition

- Induction variables lead to
  - strength reduction
  - eliminate computation

Now We have:



```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

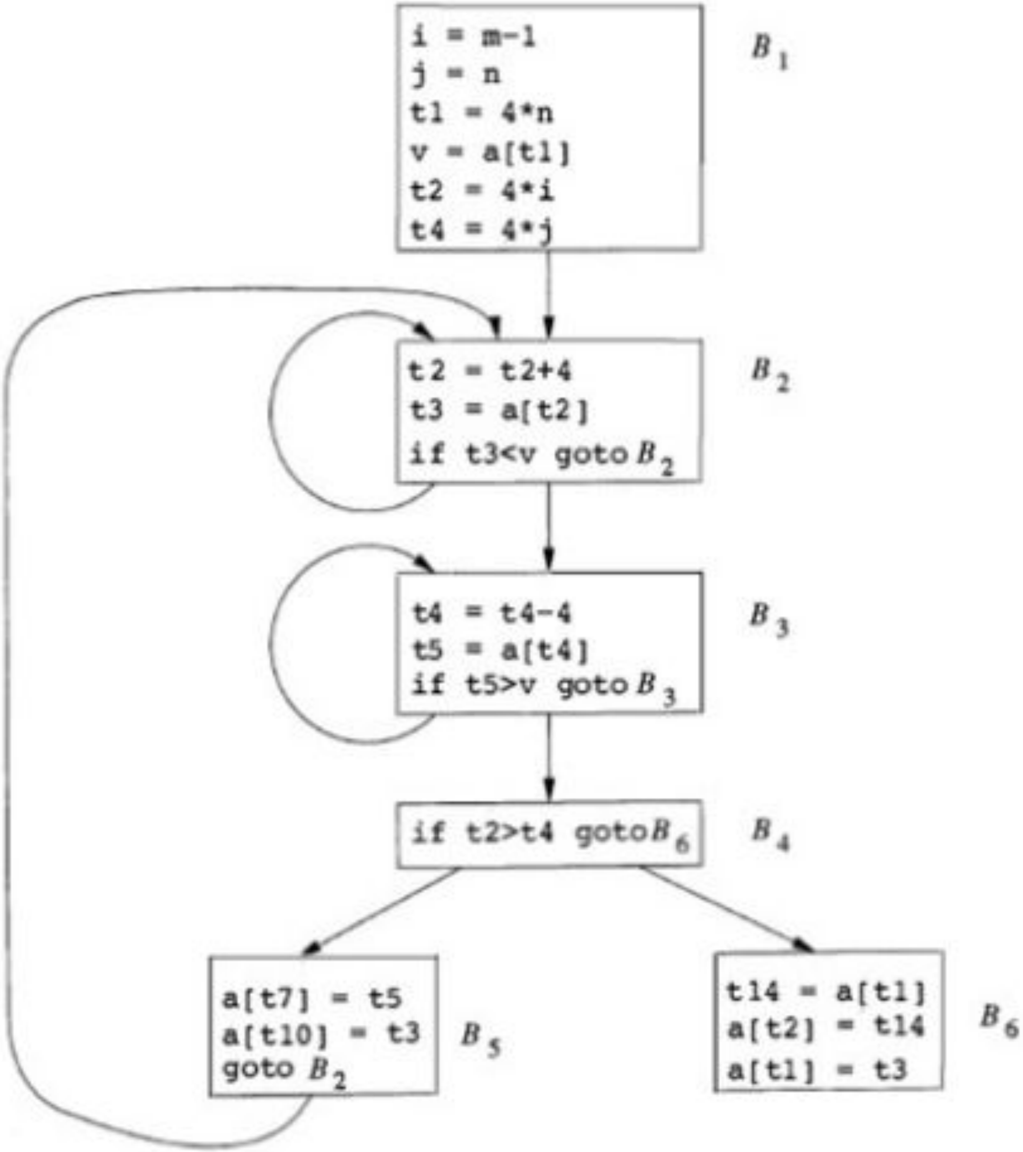
```
j = j-1
t4 = t4-4
t5 = a[t4]
if t5>v goto B3
```

```
if i>=j goto B6
```

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

```
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
```

Inside-out



```
i = m-1
j = n
t1 = 4*n
v = a[t1]
t2 = 4*i
t4 = 4*j
```

```
t2 = t2+4
t3 = a[t2]
if t3<v goto B2
```

```
t4 = t4-4
t5 = a[t4]
if t5>v goto B3
```

```
if t2>t4 gotoB6
```

```
a[t7] = t5
a[t10] = t3
goto B2
```

```
t14 = a[t1]
a[t2] = t14
a[t1] = t3
```

- Machine-independent optimizations

  Code motion

  Reduction in strength

  Common subexpression sharing

- Tuning: Identifying performance bottlenecks
- Machine-dependent optimizations

  Pointer code

  Loop unrolling

  Enabling instruction-level parallelism

- Understanding processor optimization

  Translation of instructions into operations

  Out-of-order execution

Optimizations for Basic blocks

- Reducible flow graph
- Global Data flow analysis
- Machine dependent Optimizations

Principal sources of optimization

- **Register allocation:**
  - Good usage of registers important. Reduces the time it takes to go to memory to pick up information (whether on stack/heap etc.)
  - Problem: fixed number of registers vs. large number of variables. An optimization problem.
  - **Two techniques used when designing microprocessors:**
  - Define efficient memory operations. Do not need to depend on a very efficient register allocator.
  - Define large collection of registers (32, 64, 128) so that register allocation problem is easier. (Example: RISC chips).
- **Unnecessary Operations:**
  - Avoid generating expressions that will not be needed.
  - Approaches differ from simple local searches to searches across all programs.

1/

https://slideplayer.com/slide/5270116/

- https://slideplayer.com/slide/3250106/
- Choice of instructions
- Moving code
- Reordering code
- Strength reduction
- Must be faithful to original program