COMPILER DESIGN

HANDLED BY
DIVYA B
DEPT. OF CSE
VJEC, CHEMPERI

CODE OPTIMIZATION

- The code generated by the compiler can be made faster or made to take less space or both.
- Some transformations can be applied on this code called **optimization** or **optimization transformations**.
- Compilers that can apply optimizing transformations are called **optimizing compilers.**
- Code optimization is an optional phase and it must not change the meaning of the program.

CODE OPTIMIZATION

- CO aims to improve a program, rather than improving the algorithm used in the program. Thus replacement of an algorithm by a more efficient algorithm is beyond the scope of CO.
- Efficient code generation for a specific target machine (eg: by fully exploiting its instruction set) is also beyond the scope of CO.
- Compiler was found to consume 40% extra compilation time due to optimization. The optimized program occupied 25% less storage and executed 3 times faster than an unoptimized program.

NEED FOR CODE OPTIMIZATION

- Code produced by a compiler may not be perfect in terms of execution speed and memory space.
- Optimization by hand takes much more effort and time.
- Machine level details like instructions and addresses are not known to the programmer.
- Advanced architecture features like instruction pipeline requires optimized code.
- Structure reusability and maintainability of the code are improved.

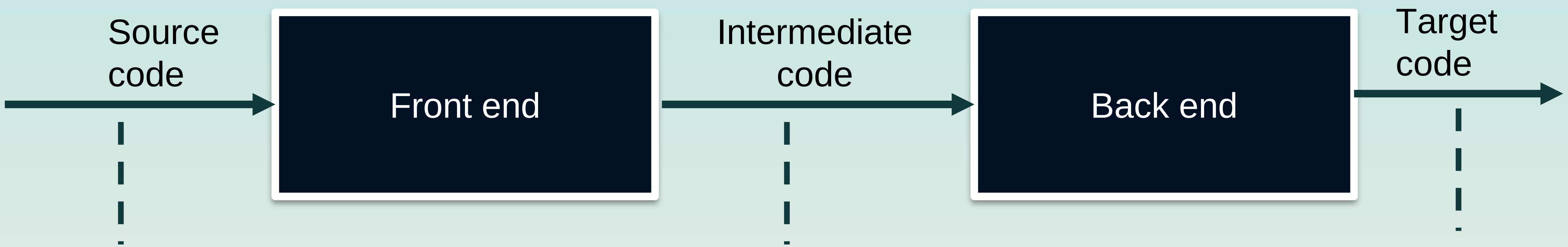
CRITERIA FOR CODE OPTIMIZATION

- It should preserve the meaning of the program ie, it should not change the output or produce error for a given input. This approach is called **safe approach**.
- Eventually it should improve the efficiency of the program by a reasonable amount. Sometimes, it may increase the size of the code or may slow the program slightly but it should improve the overall efficiency.
- It must be worth the effort, i.e., the effort put on optimization must be worthy when compared with the improvement.

CODE OPTIMIZATION

- Optimizations can be applied in 3 places
  - Source Code
  - Intermediate Code
  - Target Code
- Optimization can be done in two ways
  - Machine dependent optimization – runs only on a particular machine.
  - Machine independent optimization – used for any machine.

CODE OPTIMIZATION

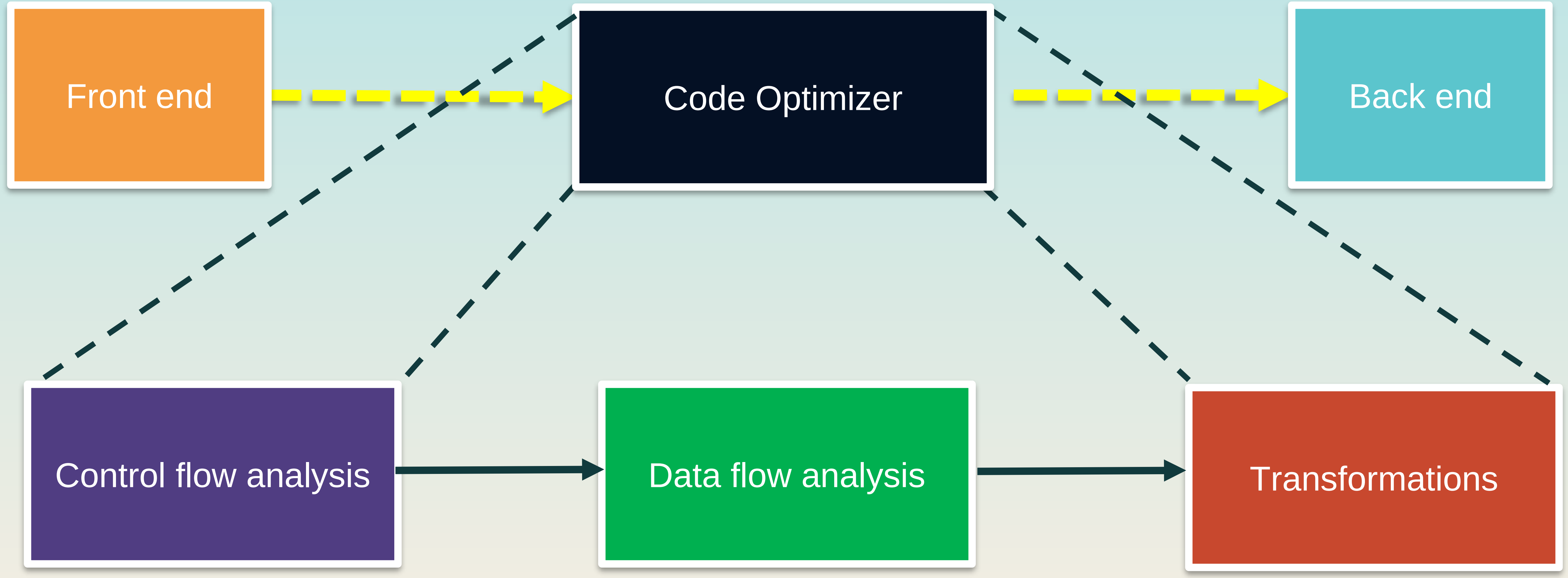


user can
change algorithm
transform loops

compiler can
improve loops
procedure calls
address calculations

compiler can
use registers
select instructions
do peephole transformations

CODE OPTIMIZATION



CODE OPTIMIZATION

- **Local Optimization**
  - Transformations are applied over a small segment of the program called basic block, in which statements are executed in sequential order. Speed-up factor for local optimization is 1.4.
- **Global Optimization**
  - Transformations are applied over a large segment of the program like loop, procedures, functions etc.
  - Local optimization must be done before applying global optimization. Speed-up factor is 2.7.

BASIC BLOCK

- Basic block is a sequence of consecutive 3 address statements which may be entered only at the beginning and when entered statements are executed in sequence without halting or branching.
- To identify basic blocks, we have to find **leader** statements.

BASIC BLOCK

- Rules for finding leader statements
  - Input : A sequence of three address statements
  - Output : A list of basic blocks with each three address statement in exactly one block.

BASIC BLOCK

- **Rules for finding leader statements**

1. Determine the set of leaders
   - ✓ The first statement is the leader.
   - ✓ Any statement that is the target of a goto is a leader.
   - ✓ Any statement that immediately follows a goto is a leader.
2. For each leader, its basic block consist of the leader and all statements up to, but not including, the next leader, or the end of the program.

FLOW GRAPHS

- It is the pictorial representation of control flow analysis in a program.
- It shows the relationship among basic blocks.
- Nodes are basic blocks and edges represent control flow.
- If there is a directed edge from B1 to B2, the control transfers from the last statement of B1 to the first statement of B2. B1 is called **predecessor** of B2 and B2 is **successor** of B1.

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 go to (3)
10) i = i+1
11) if i <=10 go to (2)
12) i = 1

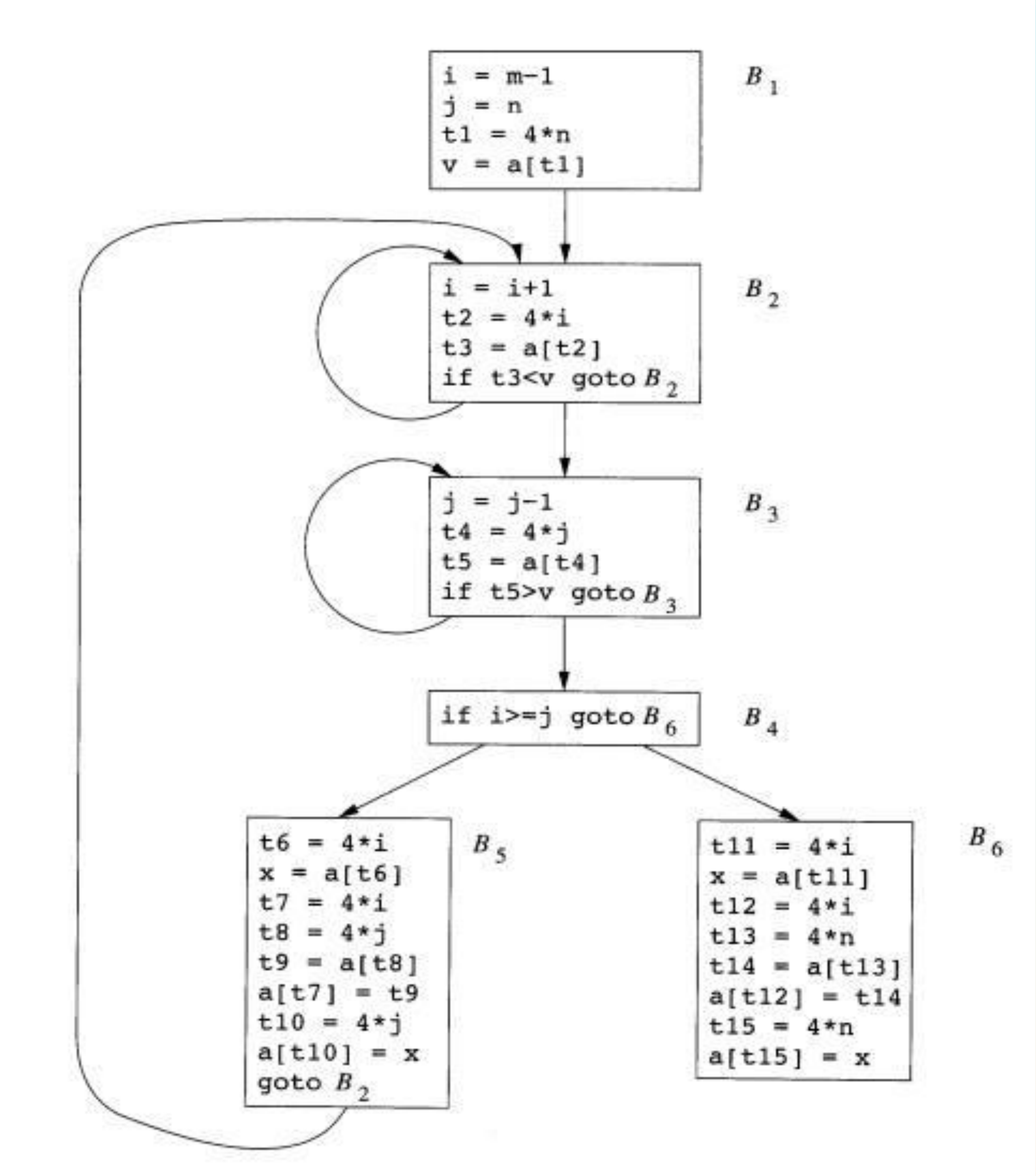
13) t5 = i -1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 go to (13)

```
void quicksort(m,n)
int m,n;
{
int i,j,v,x;
if (n <= m) return;
i = m-1; j = n; v = a[n]; /* fragment begins here */
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; /* fragment ends here */
quicksort(m,j); quicksort(i+1,n);
}
```

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]
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

PRINCIPAL SOURCES OF OPTIMIZATION

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels.
- Local transformations are usually performed first.

PRINCIPAL SOURCES OF OPTIMIZATION

- **Function preserving transformations**
- There are a number of ways in which a compiler can improve a program without changing the function it computes.
  - ✓ Common subexpression elimination
  - ✓ Copy propagation
  - ✓ Dead code elimination
  - ✓ Constant folding

PRINCIPAL SOURCES OF OPTIMIZATION

- **Loop Optimization**
  - ✓ Code Motion
  - ✓ Induction Variable Elimination
  - ✓ Reduction in Strength

PRINCIPAL SOURCES OF OPTIMIZATION

- Function preserving transformations
  - ✓ **Common subexpression elimination**
    - ✓ An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation.

PRINCIPAL SOURCES OF OPTIMIZATION

- *Function preserving transformations*
  - ✓ **Common subexpression elimination**
    - ✓ We can avoid recomputing the expression if we can use the previously computed value.
    - ✓ Two types
      - **Local common sub expression elimination**
      - **Global common sub expression elimination**

PRINCIPAL SOURCES OF OPTIMIZATION

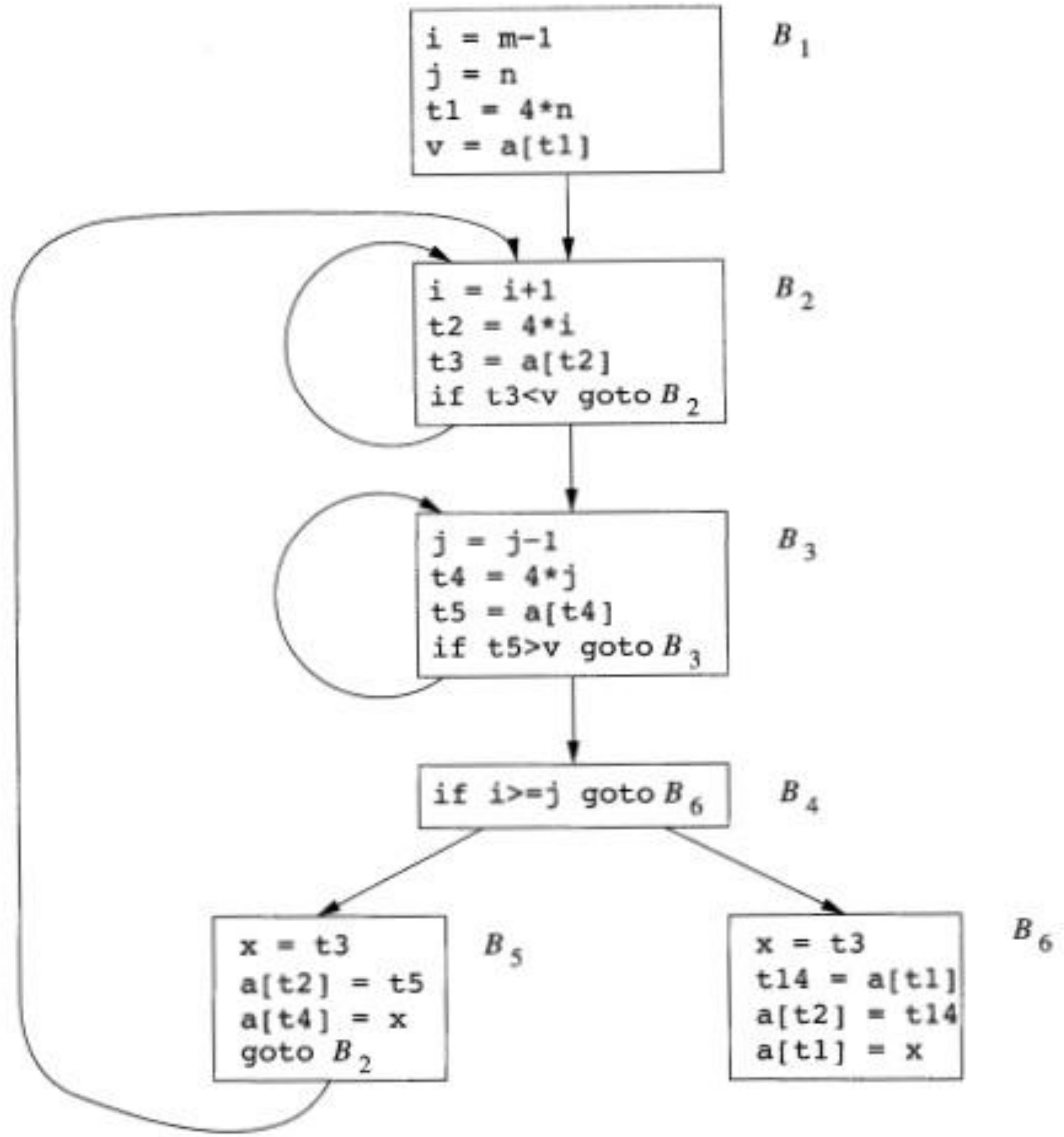
- **Function preserving transformations**
  - ✓ **Local Common subexpression elimination**

```
t6 = 4*i                B5
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.

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

(b) After.


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

PRINCIPAL SOURCES OF OPTIMIZATION

- Function preserving transformations
  - ✓ **Copy propagation**
    - ✓ Assignments of the form f = g called copy statements, or copies for short.
    - ✓ The idea behind the copy-propagation. transformation is to use g for f, whenever possible after the copy statement f = g.

PRINCIPAL SOURCES OF OPTIMIZATION

- Function preserving transformations
  - ✓ **Copy propagation**

B5

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

→

B5

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

PRINCIPAL SOURCES OF OPTIMIZATION

- **Function preserving transformations**
  - ✓ **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.

PRINCIPAL SOURCES OF OPTIMIZATION

- Function preserving transformations
  - ✓ **Dead Code Elimination**

```
i=0;
if(i==1)
{
a=b+5;
}
```

Dead code

PRINCIPAL SOURCES OF OPTIMIZATION

- **Function preserving transformations**
  - ✓ **Constant Folding**
    - ✓ If all operands are constants in an expression, then it can be evaluated at compile time itself.
    - ✓ The result of the operation can replace the original evaluation in the program.
    - ✓ This will improve the run time performance and reduce size of the code by avoiding evaluation at compile time.

E.g. a=3.14157/2 can be replaced by a=1.570 thereby eliminating a division operation.

PRINCIPAL SOURCES OF OPTIMIZATION

- **Loop Optimizations**
  - ✓ The running time of a program may be improved if the number of instructions in a loop is decreased, even if we increase the amount of code outside the loop.

PRINCIPAL SOURCES OF OPTIMIZATION

- **Loop Optimizations**
  - ✓ **Code Motion**
    - ✓ An important modification that decreases the amount of code in a loop is code motion.
    - ✓ This transformation takes an expression that yields the same result independent of the number of times a loop is executed and places the expression before the loop.

PRINCIPAL SOURCES OF OPTIMIZATION

- Loop Optimizations
  - ✓ **Code Motion**
    - ✓ A fragment of code that resides in the loop and computes the same value of each iteration is called loop invariant code.

PRINCIPAL SOURCES OF OPTIMIZATION

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

→

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

PRINCIPAL SOURCES OF OPTIMIZATION

- Loop Optimizations
  - ✓ **Induction Variable Elimination**
    - ✓ Any two variables are said to be induction variables, if a change in any one of the variables leads to a corresponding change in the other variable.

PRINCIPAL SOURCES OF OPTIMIZATION

Induction variables are i1, i2 and i3

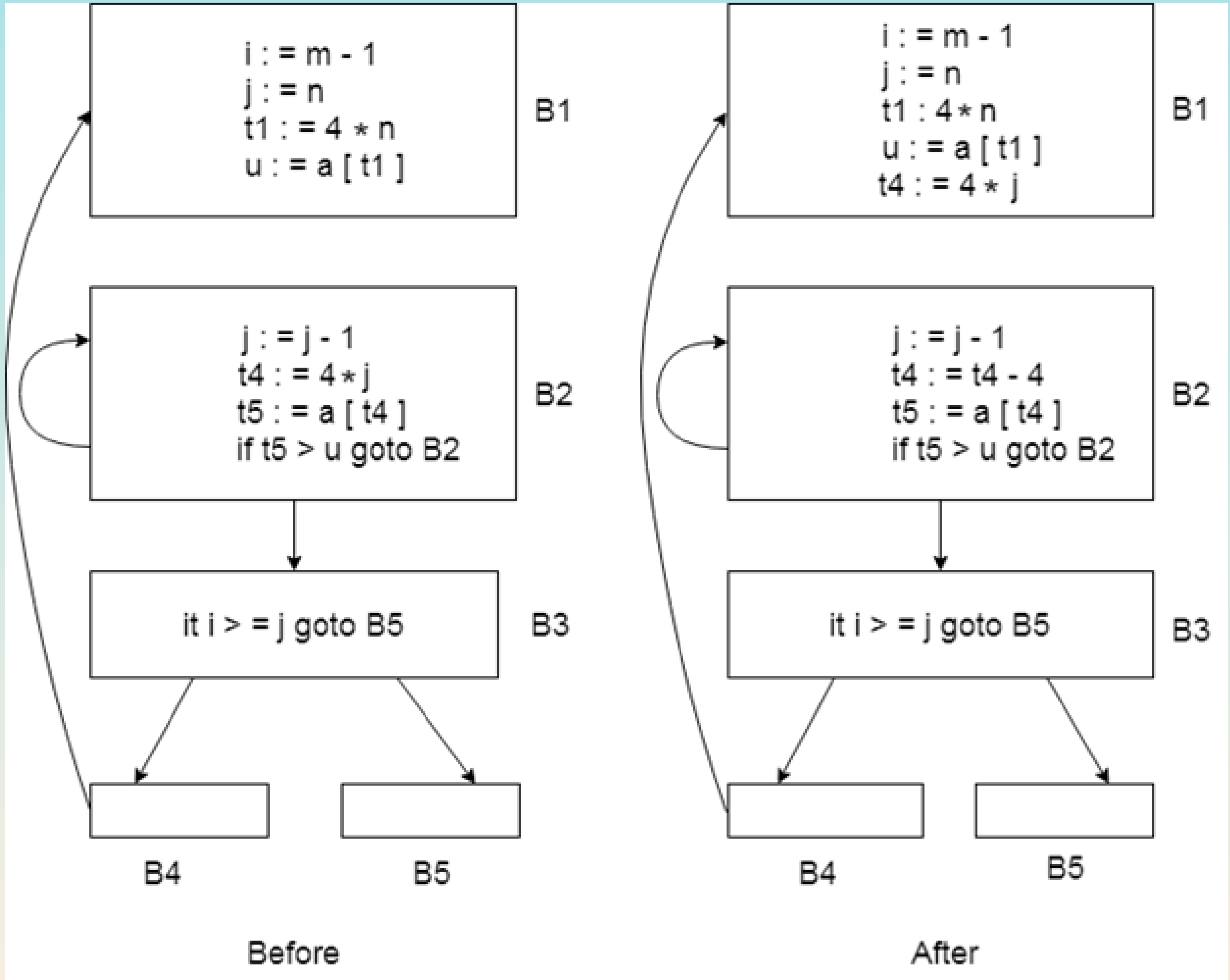
```
int a[SIZE];
int b[SIZE];
void f (void)
{
int i1, i2, i3;
for (i1 = 0, i2 = 0, i3 = 0; i1 < SIZE; i1++)
a[i2++] = b[i3++];
return; }
```

After induction variable elimination

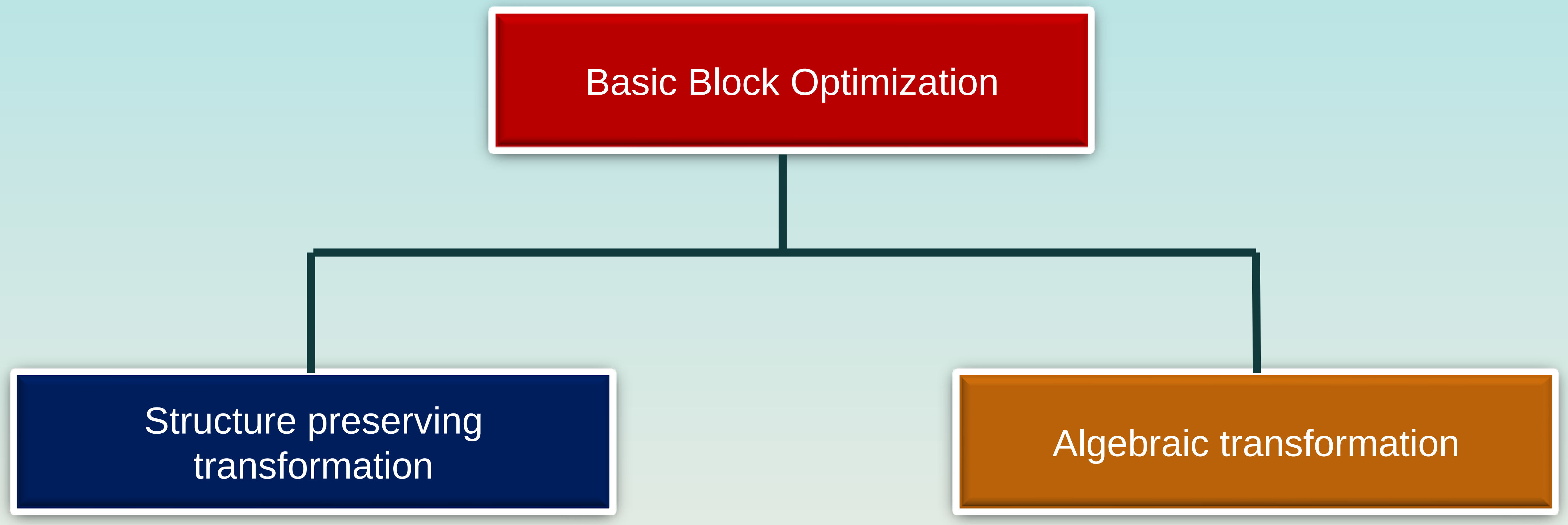
```
int a[SIZE];
int b[SIZE];
void f (void)
{
int i1; for (i1 = 0; i1 < SIZE; i1++)
a[i1] = b[i1];
return;
}
```

PRINCIPAL SOURCES OF OPTIMIZATION

- **Loop Optimizations**
  - ✓ **Strength Reduction**
    - ✓ There are expressions that consume more CPU cycles, time, and memory.
    - ✓ These expressions should be replaced with cheaper expressions without compromising the output of expression.
    - ✓ For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.



Divys-Compiler Design PPT


Basic Block Optimization
Structure preserving transformation
Algebraic transformation

OPTIMIZATION OF BASIC BLOCKS

- The structure-preserving transformation on basic blocks includes
  - ✓ Dead code elimination
  - ✓ Common subexpression elimination
  - ✓ Renaming of temporary variables
  - ✓ Interchange of two independent adjacent statements

OPTIMIZATION OF BASIC BLOCKS

- Many of the structure preserving transformations can be implemented by constructing a DAG for a block.
- There is a node ***n*** associated with each statement ***s*** within the block.
- The children of ***n*** are those nodes corresponding to statement that are the last definitions prior to ***s*** of the operands used by ***s***.

OPTIMIZATION OF BASIC BLOCKS

- DAG
  - ✓ DAG is an useful data structure for implementing transformation on basic blocks.
  - ✓ DAG is constructed from three address code.
  - ✓ Common subexpression can be detected by noticing, as a new node m is about to added, whether there is an existing node n with the same children, in the same order, and with the same operator. If so, n computes the same value as m and may be used in its place.

OPTIMIZATION OF BASIC BLOCKS

- **DAG – Applications**
  - ✓ Determine the common subexpression.
  - ✓ Determine which names are used in the block and compute outside the block.
  - ✓ Determine which statement of the block could have their computed value outside the block.
  - ✓ Simplify the list of quadruples by eliminating common subexpression and not performing the assignment of the form x = y and unless it is a must.

OPTIMIZATION OF BASIC BLOCKS

- **Rules for constructing DAG**
  - ✓ In a DAG Leaf node represents identifiers, names, constants. Interior node represents operators.
  - ✓ While constructing DAG, there is a check made to find if there is an existing node with same children. A new node is created only when such a node does not exist. This action allows us to detect common subexpression and eliminate the same.
  - ✓ Assignments of the form x = y must not be performed until unless it is a must.

OPTIMIZATION OF BASIC BLOCKS

- **Algebraic Transformations**
  - ✓ Using algebraic identities

$$x + 0 = 0 + x = x$$
$$x - 0 = x$$
$$x * 1 = 1 * x = x$$
$$x / 1 = x$$

OPTIMIZATION OF BASIC BLOCKS

- *Algebraic Transformations*

  ✓ Another class of algebraic optimization includes reduction in strength.

  x ** 2 = x * x

  2 * x = x + x

  x / 2 = x * 0.5

OPTIMIZATION OF BASIC BLOCKS

- Algebraic Transformations
  - ✓ Associative laws can be applied to expose common subexpression.

    a = b + c

    e = c + d + b

    Intermediate code will be

    a = b + c

    t = c + d

    e = t + b

If t is not needed outside this block, the sequence can be

a = b + c

e = a + d

OPTIMIZATION OF BASIC BLOCKS

- *Algebraic Transformations*
  - ✓ The compiler writer should examine the language specification carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics.
  - ✓ Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.