# Code Optimization

**Md. Khorshed Alam**

**CSE, NDUB**

# Code Optimization

- High-level language constructs can introduce substantial run-time overhead if we naively translate each construct independently into machine code.
- *Elimination of unnecessary instructions* in object code, or the *replacement of one sequence of instructions by a faster sequence* of instructions that does the same thing is usually called "code improvement" or "code optimization."

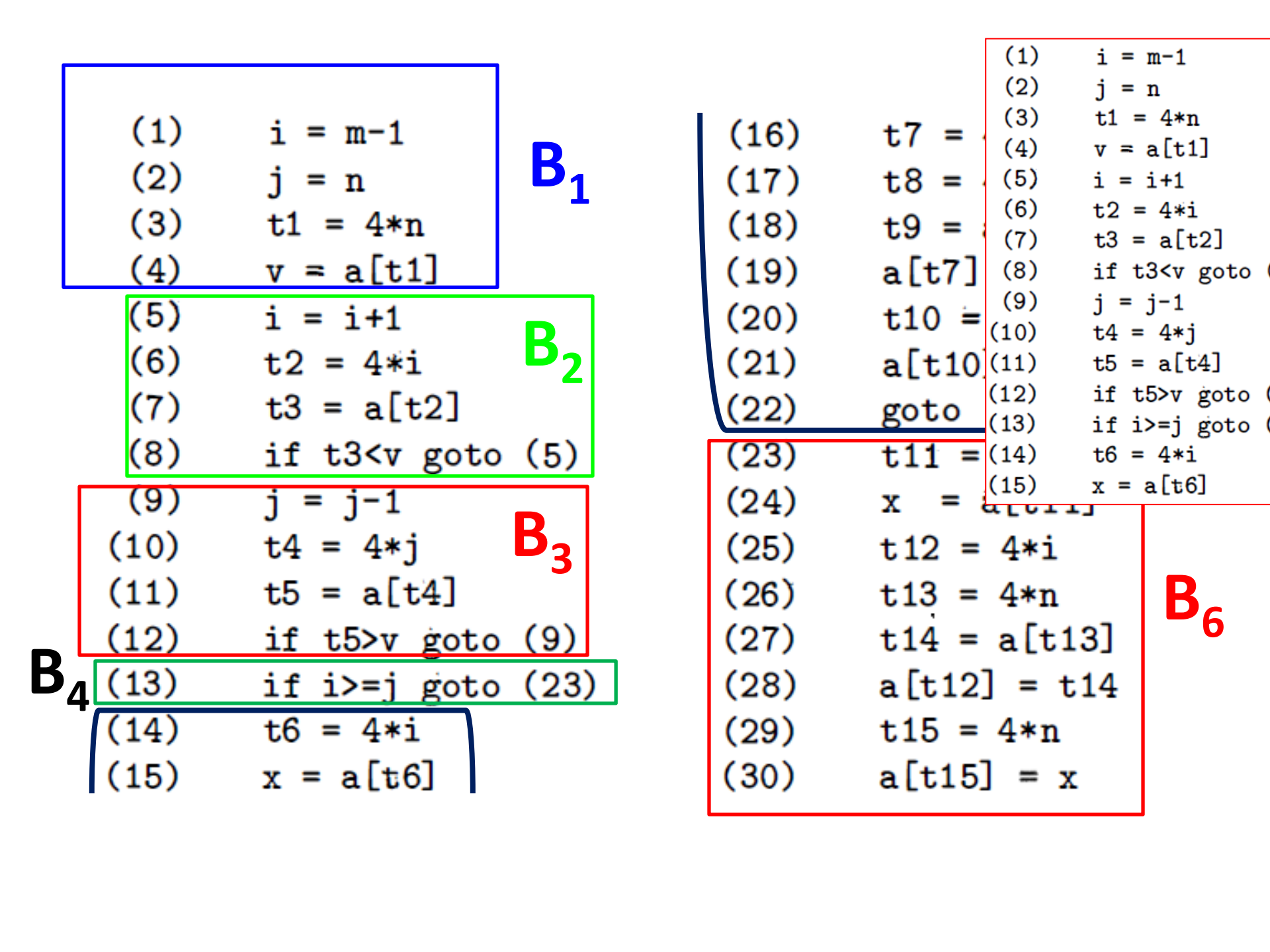
# Quicksort

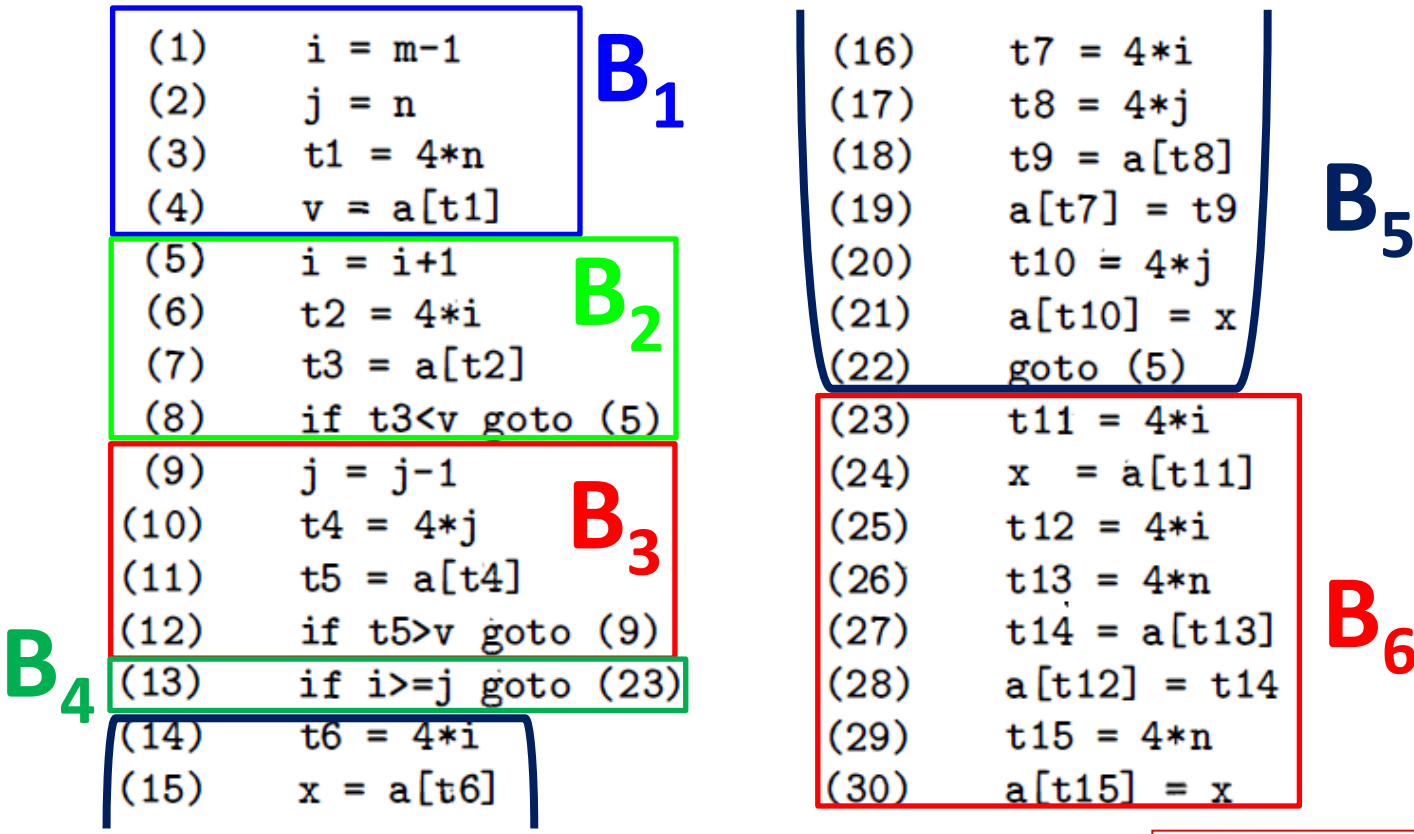
```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >=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

```
(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]
```

**Integers = 04 bytes**

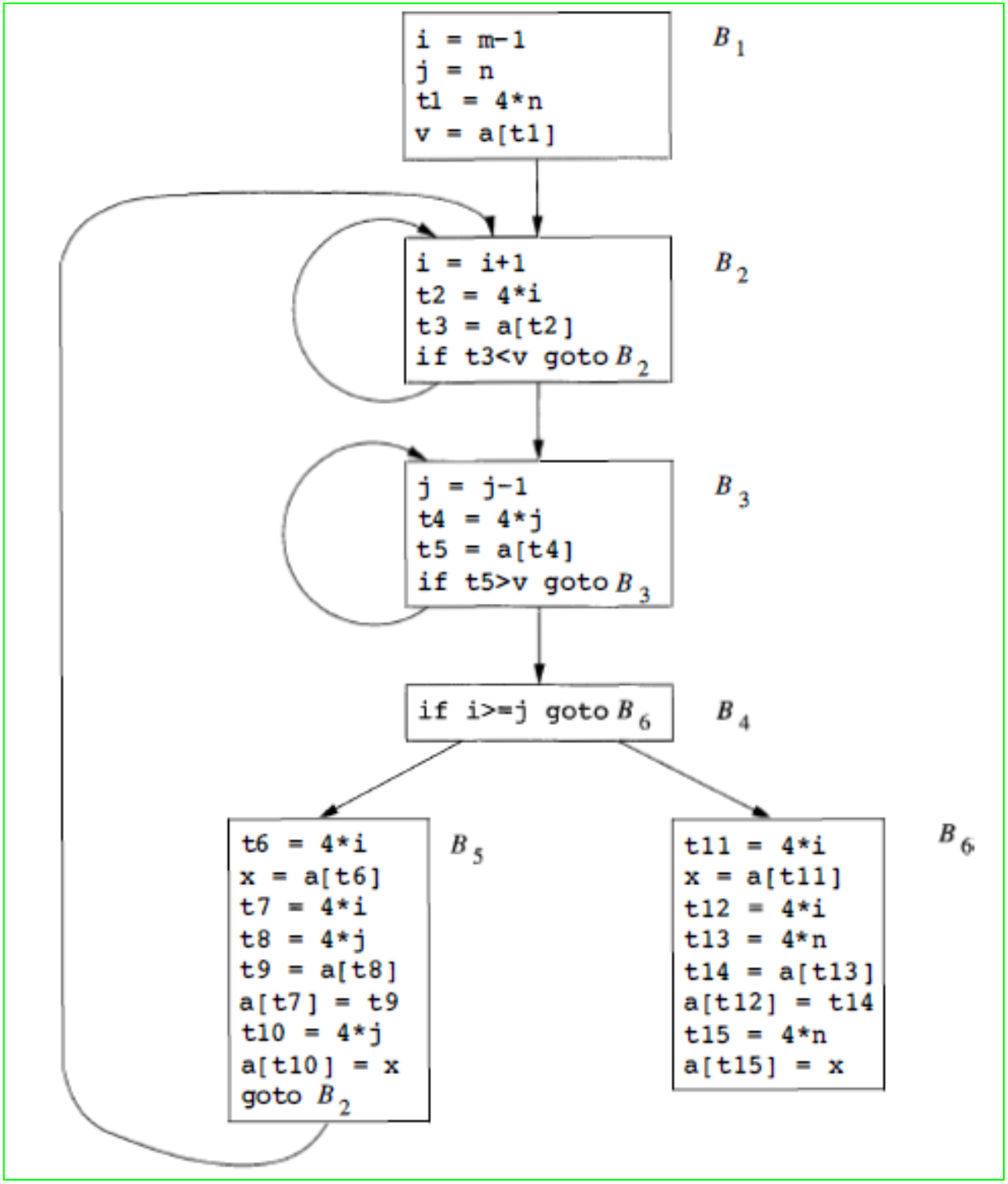
```
(16)    t7 = 4*i
(17)    t8 = 4*j
(18)    t9 = a[t8]
(19)    a[t7] = t9
(20)    t10 = 4*j
(21)    a[t10] = x
(22)    goto (5)
(23)    t11 = 4*i
(24)    x  = a[t11]
(25)    t12 = 4*i
(26)    t13 = 4*n
(27)    t14 = a[t13]
(28)    a[t12] = t14
(29)    t15 = 4*n
(30)    a[t15] = x
```


(1) i = m-1
(2) j = n
(3) t1 = 4*n
(4) v = a[t1]
B1
(5) i = i+1
(6) t2 = 4*i
(7) t3 = a[t2]
(8) if t3<v goto (5)
B2
(9) j = j-1
(10) t4 = 4*j
(11) t5 = a[t4]
(12) if t5>v goto (9)
B3
B4
(13) if i>=j goto (23)
(14) t6 = 4*i
(15) x = a[t6]
(16) t7 =
(17) t8 =
(18) t9 =
(19) a[t7]
(20) t10 =
(21) a[t10
(22) goto
(23) t11 =
(24) x =
(25) t12 = 4*i
(26) t13 = 4*n
(27) t14 = a[t13]
(28) a[t12] = t14
(29) t15 = 4*n
(30) a[t15] = x
B6
(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
(9) j = j-1
(10) t4 = 4*j
(11) t5 = a[t4]
(12) if t5>v goto
(13) if i>=j goto
(14) t6 = 4*i
(15) x = a[t6]




```
(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
```

# Flow Graph


i = m-1
j = n
t1 = 4*n
v = a[t1]
$B_1$
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto $B_2$
$B_2$
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto $B_3$
$B_3$
if i>=j goto $B_6$
$B_4$
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 $B_2$
$B_5$
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
$B_6$

# Optimization Techniques: Semantic Preserving Transformations

- Common-sub expression elimination
- Copy Propagation
- Dead-code elimination
- Constant folding

# Local Common Sub-expression Elimination

- A program will include several calculations of the same value, such as an offset in an array.
- Some of these *duplicate calculations cannot be avoided* by the programmer because they lie below the level of detail accessible within the source language.

❑ **B5 recalculates *4*i & 4*j*, although none of these calculations were requested explicitly by the programmer.**

**Before**

```
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
```

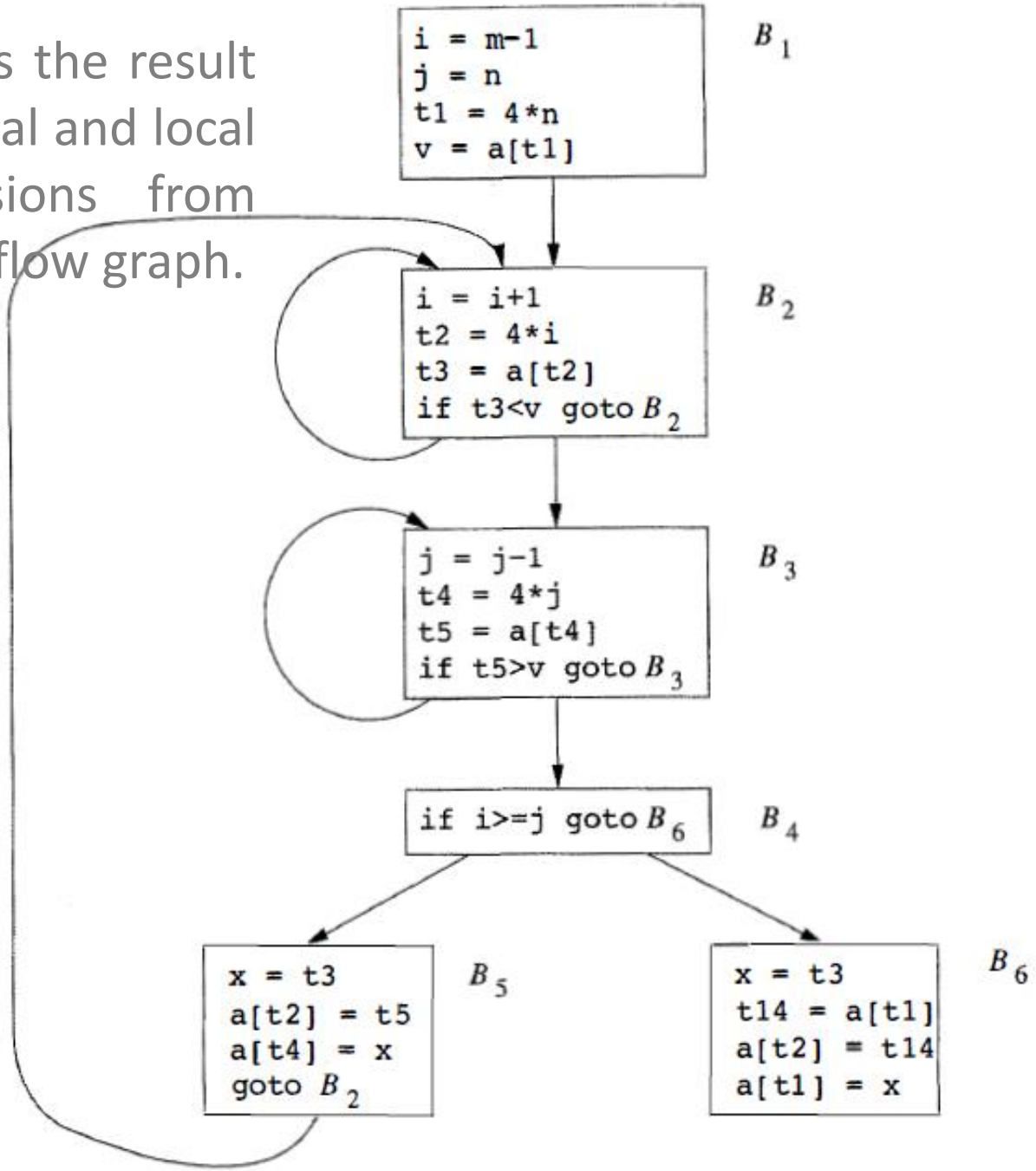
**After**

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

# GLOBAL Common Sub-expression Elimination

- *An occurrence of an expression E is called a common sub-expression if E was previously computed & the values of the variables in E have not changed since the previous computation*
- We avoid recomputing E if we can use its previously computed value;
- variable x to which the previous computation of E was assigned has not changed in the interim.

**Example 9.2**: Fig. shows the result of eliminating both global and local common sub expressions from blocks B5 and B6 in the flow graph.



After local common sub expressions are eliminated, $B_5$ still evaluates 4*i & 4*j.
Both are common subexpressions; in particular, the three statements

```
t8 = 4*j
t9 = a[t8]
a[t8] = x
```

in $B_5$ can be replaced by

```
t9 = a[t4]
a[t4] = x
```

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

$B_5$

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

$B_3$

- using $t_4$ computed in block $B_3$ .
- Observe that as control passes from the evaluation of **4*j** in $B_3$ to $B_5$, there is no change to **j** & no change to $t_4$, so $t_4$ can be used if **4*j** is needed.

- Another common sub expression comes to light in $B_5$ after $t_4$ replaces $t_8$
- The new expression $a[t_4]$ corresponds to the value of a[ j] at the source level.
- Not only does j retain its value as control leaves $B_3$ & then enters $B_5$ , but a[j] , a value computed into a temporary $t_5$, does too, because there are no assignments to elements of the array a in the interim.

**The statements**

```
t9 = a[t4]
a[t6] = t9
```

in $B_5$ therefore can be replaced by

```
a[t6] = t5
```

$B_3$

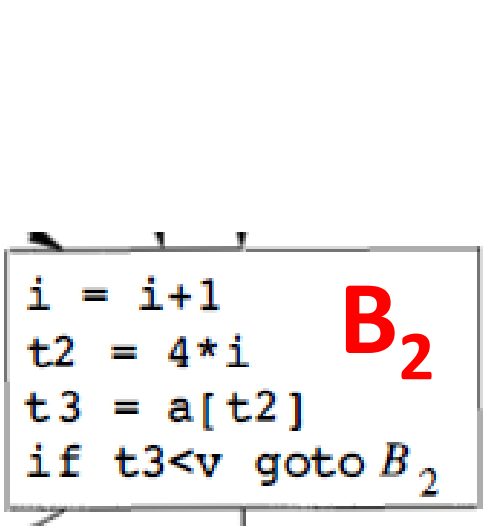
```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

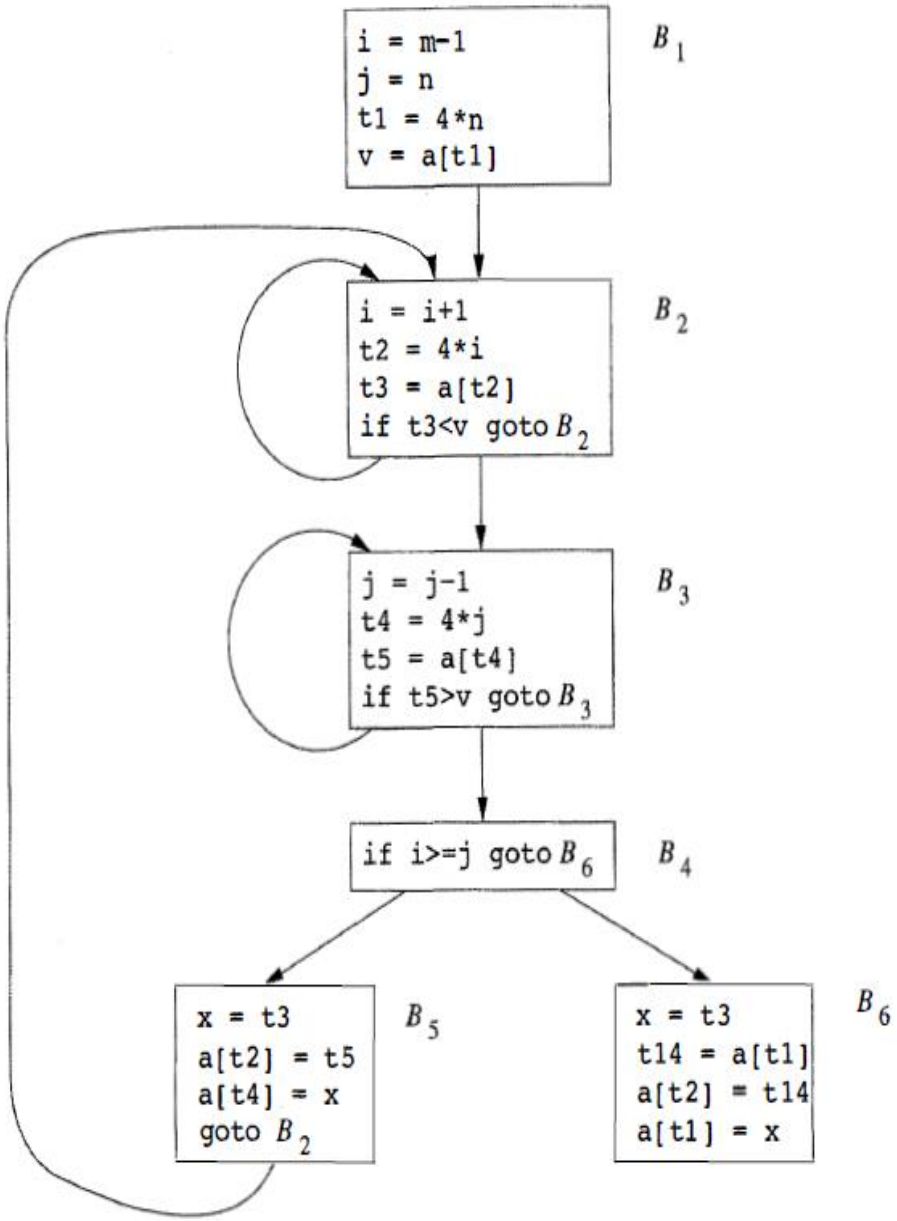
$B_5$

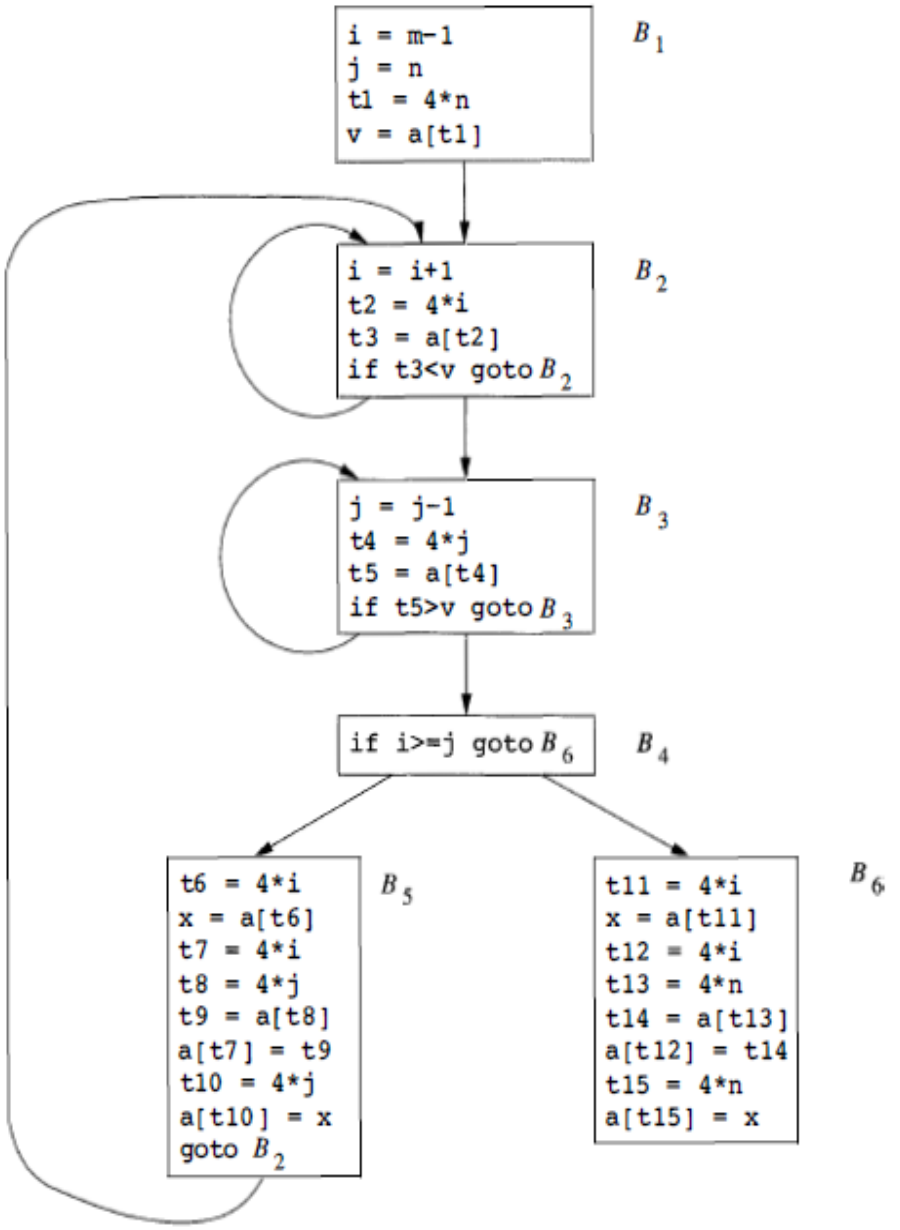
```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

Analogously, the value assigned to $x$ in block $B_5$ of Fig. 9.4(b) is seen to be the same as the value assigned to $t3$ in block $B_2$. Block $B_5$ in Fig. 9.5 is the result of eliminating common subexpressions corresponding to the values of the source level expressions $a[i]$ and $a[j]$ from $B_5$ in Fig. 9.4(b). A similar series of transformations has been done to $B_6$ in Fig. 9.5.

The expression $a[t1]$ in blocks $B_1$ and $B_6$ of Fig. 9.5 is *not* considered a common subexpression, although $t1$ can be used in both places. After control leaves $B_1$ and before it reaches $B_6$, it can go through $B_5$, where there are assignments to $a$. Hence, $a[t1]$ may not have the same value on reaching $B_6$ as it did on leaving $B_1$, and it is not safe to treat $a[t1]$ as a common subexpression.



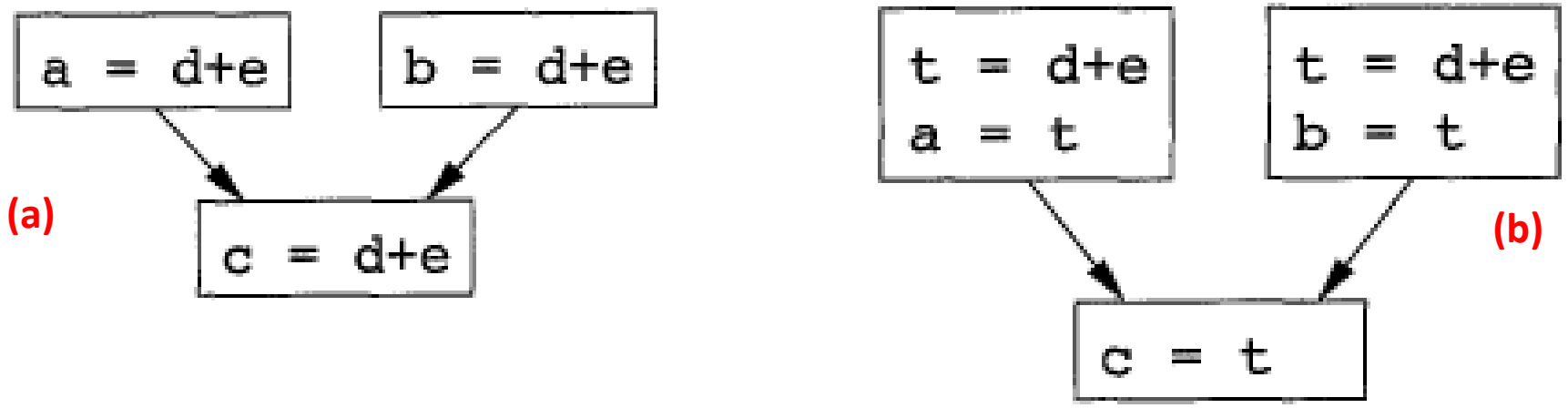

i = m-1
j = n
t1 = 4*n
v = a[t1]
$B_1$
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto $B_2$
$B_2$
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto $B_3$
$B_3$
if i>=j goto $B_6$
$B_4$
x = t3
a[t2] = t5
a[t4] = x
goto $B_2$
$B_5$
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
$B_6$


i = m-1
j = n
t1 = 4*n
v = a[t1]
$B_1$
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto $B_2$
$B_2$
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto $B_3$
$B_3$
if i>=j goto $B_6$
$B_4$
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 $B_2$
$B_5$
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
$B_6$

# Copy Propagation

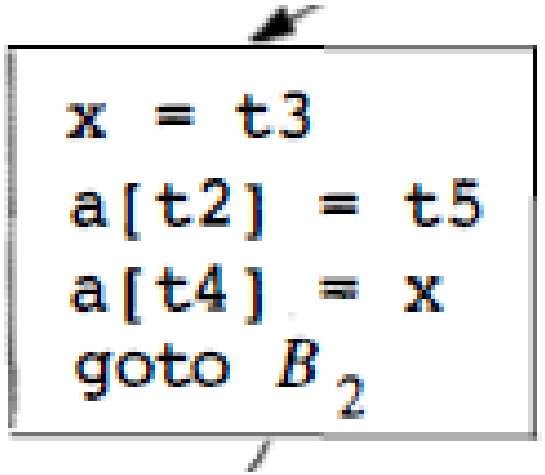
- Block $B_5$ can be further improved by eliminating x using two new transformations.
- One concerns assignments of the form **u = v** called **copy** statements.

**Example 9.3 :** In order to eliminate the common subexpression from the statement `c = d+e` in Fig. 9.6(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. 9.6(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`. □



- The idea behind the copy-propagation transformation is to use **v** for u, wherever possible after the copy statement **u=v**.
- assignment

**x = $t_3$** in block **$B_5$** is a copy.



# 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.

**Example 9.4:** Suppose `debug` is set to `TRUE` or `FALSE` at various points in the program, and used in statements like

```
if (debug) print ...
```

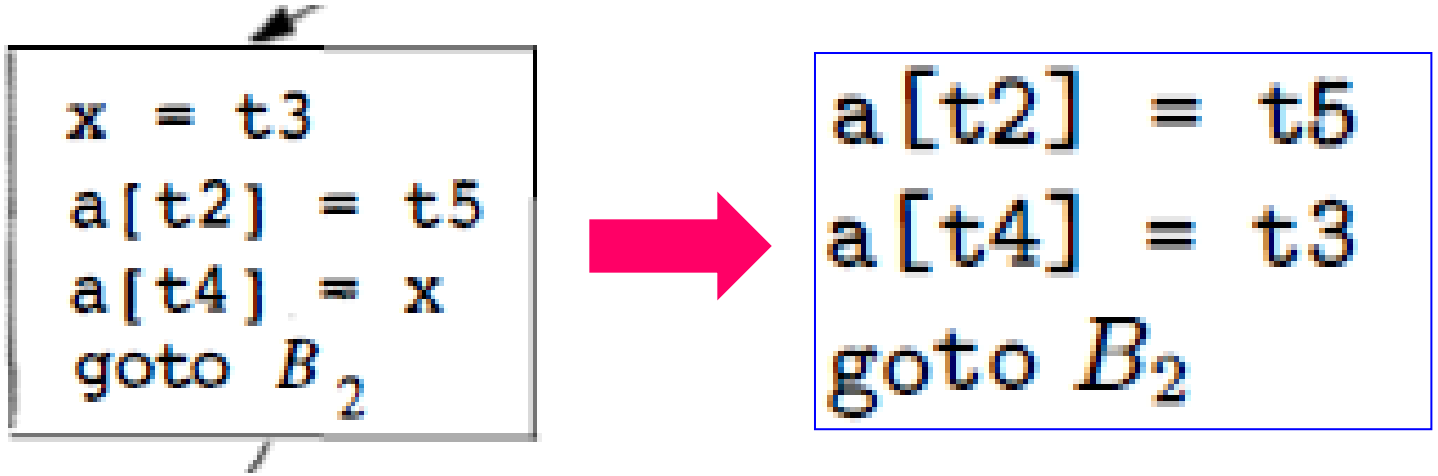
It may be possible for the compiler to deduce that each time the program reaches this statement, the value of `debug` is `FALSE`. Usually, it is because there is one particular statement

```
debug = FALSE
```

that must be the last assignment to `debug` prior to any tests of the value of `debug`, no matter what sequence of branches the program actually takes. If copy propagation replaces `debug` by `FALSE`, then the print statement is dead because it cannot be reached. We can eliminate both the test and the print operation from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as *constant folding*. □

- One advantage of copy propagation is that it often turns copy statement into dead code.

❑Copy propagation followed by dead-code elimination removes the assignment to x and transforms the code



# Problems

- Page 51: 2.2.1, 2.2.2
- Page 125: 3.3.2,
- Page 151: 3.6.3
- Page 166: 3.7.3
- Page 206: 4.2.1, 4.2.2
- Page 216: 4.3.1
- Page 370: 6.2.1
- Page 440: 7.2.3
- Page 516: 8.2.1, 8.2.2, 8.2.6
- Page 531: 8.4.1