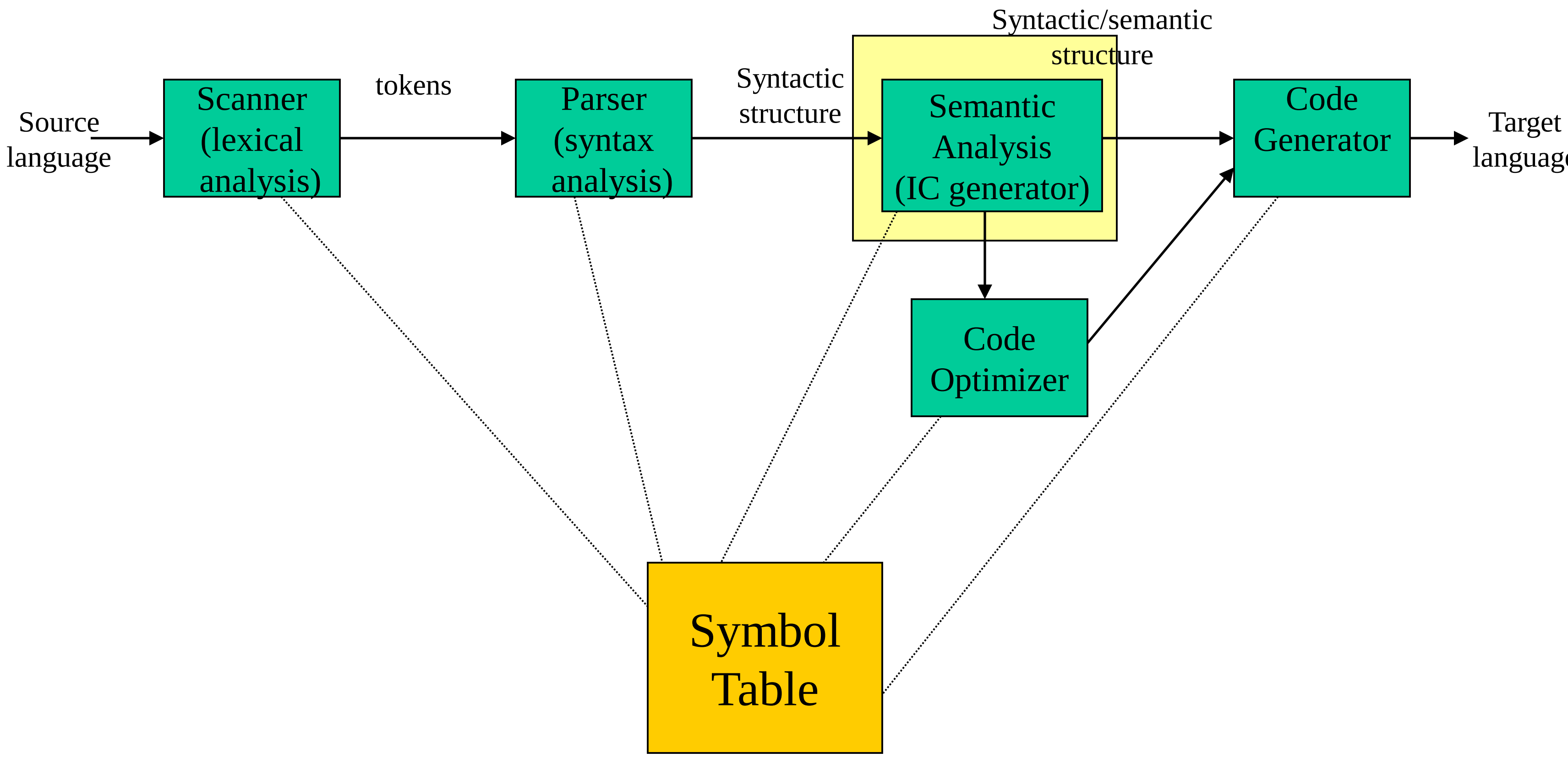
Code Optimization



Code Optimization

REQUIREMENTS:

- Meaning must be preserved (correctness)
- Speedup must occur on average.
- Work done must be worth the effort.

OPPORTUNITIES:

- Programmer (algorithm, directives)
- Intermediate code
- Target code

Optimization

- Refers to the techniques used by the compiler to improve the execution efficiency of the generated object code.
- Machine independent

(1) Loop Optimization

- Code motion/frequency reduction
- Loop unrolling
- Loop jamming

(2) Folding -> constant propogation

(3) Redundancy elimination

(4) Strength reduction

- Machine dependent

(1) Register allocation

(2) Peephole optimization

Peephole Optimizations

- Constant Folding

  `x := 32` becomes **`x := 64`**

  `x := x + 32`

- Unreachable Code

  `goto L2`

  `x := x + 1` ← unneeded

- Flow of control optimizations

  `goto L1` becomes **`goto L2`**

  …

  `L1: goto L2`

Peephole Optimizations

- Algebraic Simplification

  `x := x + 0` ← unneeded

- Dead code

  `x := 32` ← where x not used after statement

  `y := x + y` → `y := y + 32`

- Reduction in strength

  `x := x * 2` → `x := x + x`

Register Allocation

- Its goal is to find a way to map the temporary variables used in a program into physical memory locations (either main memory or machine registers).
- Accessing a register is much faster than accessing memory,therefore one tries to use registers as much as possible.
- Ex. : x = y+z

```
MOV  R0,x
ADD  R0,z
MOV  R0,x
```

Loop optimization

- To eliminate loop invariant computations and induction variables
- Loop invariant computation– that computes the same value every time a loop is executed. So, moving such a computation outside the loop leads to a reduction in the execution time.
- Induction variables – used in loop and their values are in lock step.

Eliminating loop invariant computations

- First identify it
- Move them outside loop – meaning should not be changed
- To detect loops in the program – control flow analysis required
- So partition intermediate code into basic blocks

- Which requires identifying leader statements:

Basic Block : is a sequence of three-address statements that can be entered only at the beginning, and control ends after the execution of the last statement, without a halt or any possibility of branching, except at the end

Basic Blocks : Algorithm

Method to find basic blocks:

- Input: a sequence of three-address statements
- Output: a list of basic blocks

(1) First determine the set of leaders

- The first statement is a leader
- The target of a conditional or unconditional goto is a leader
- A statement that immediately follows a conditional goto is a leader

(2) For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

Flow graphs

- Add the flow control information to the set of basic blocks making up a program by constructing a directed graph called a flow graph
- Nodes of flow graphs are basic blocks

- There is directed edge from B1 to B2 if B2 can immediately follow B1 in some execution sequence; that is if

  1) there is a conditional or unconditional jump from the last statement of B1 to the first statement of B2 or

  2) B2 immediately follows B1 in the order of the program and B1 does no end in an unconditional jump.
- We say B1 is a predecessor of B2, and B2 is a successor of B1.

Example

Fact(x)

```
{
    int f=1;
    for(i=2;i<=x;i++)
    f=f*i;
    return(f);
}
```

<u>Three-address code representations</u>

(1) f=1;

(2) i=2

(3) if i<=x goto(8)

(4) f=f*i

(5) t1=i+1

(6) i=t1

(7) goto(3)

(8) goto calling program

The leader statements are:

- Statement number 1, becoz it's the first statement
- Statement number 3, becoz it's the target of a goto
- Statement number 4, becoz it immediately follows a conditional goto statement
- Statement number 8, becoz it's a garget of a conditional goto statement

Basic blocks and flow graph

- Block B1

```
f=1;
i=2
```

- Block B2

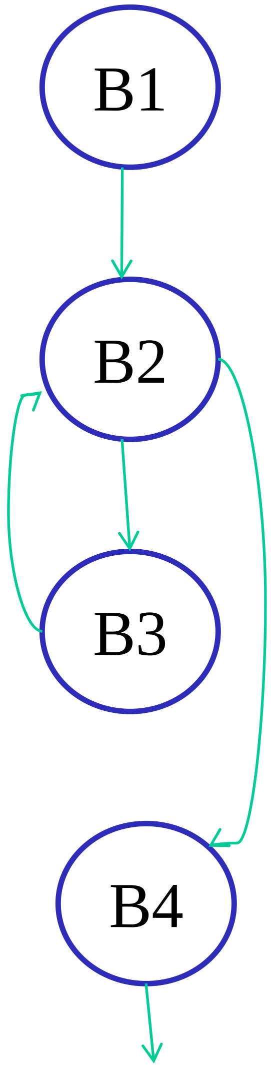
```
If i<=x goto(8)
```

- Block B3

```
f=f*i;
t1=i+1;
i=t1;
goto(3)
```

- Block B4

```
goto calling program
```



Then apply below mentioned optimization techniques at basic block level to every blocks

- Common Subexpression elimination
- Constant Propagation
- Dead code elimination
- Plus many others such as copy propagation, value numbering, partial redundancy elimination, …

Common Expression elimination

- t1 = i+1
- t2 = b[t1]
- t3 = i + 1
- a[t3] = t2

- t1 = i + 1
- t2 = b[t1]
- t3 = i + 1 ← *no longer live*
- a[t1] = t2

Common expression can be eliminated: a[i+1] = b[i+1]

Now, suppose i is a constant:

- i = 4
- t1 = i+1
- t2 = b[t1]
- a[t1] = t2

- i = 4
- t1 = 5
- t2 = b[t1]
- a[t1] = t2

- i = 4
- t1 = 5
- t2 = b[5]
- a[5] = t2

Final Code:

- i = 4
- t2 = b[5]
- a[5] = t2

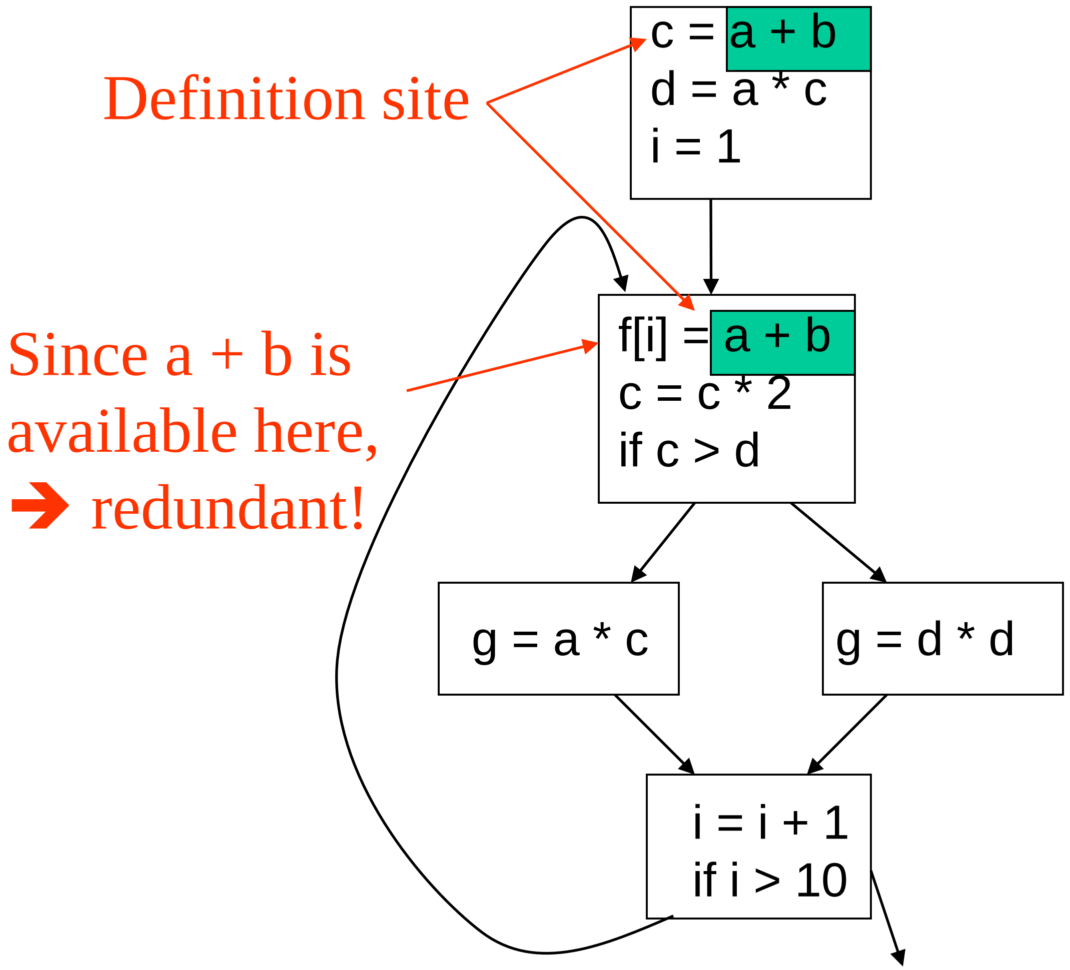
Redundant Expressions

An expression **x op y** is redundant at a point p if it has already been computed at some point(s) and no intervening operations redefine **x** or **y**.

`m = 2*y*z`	`t0 = 2*y`	`t0 = 2*y`
	`m = t0*z`	`m = t0*z`
`n = 3*y*z`	`t1 = 3*y`	`t1 = 3*y`
	`n = t1*z`	`n = t1*z`
`o = 2*y-z`	`t2 = 2*y`	
	`o = t2-z`	`o = t0-z`

redundant

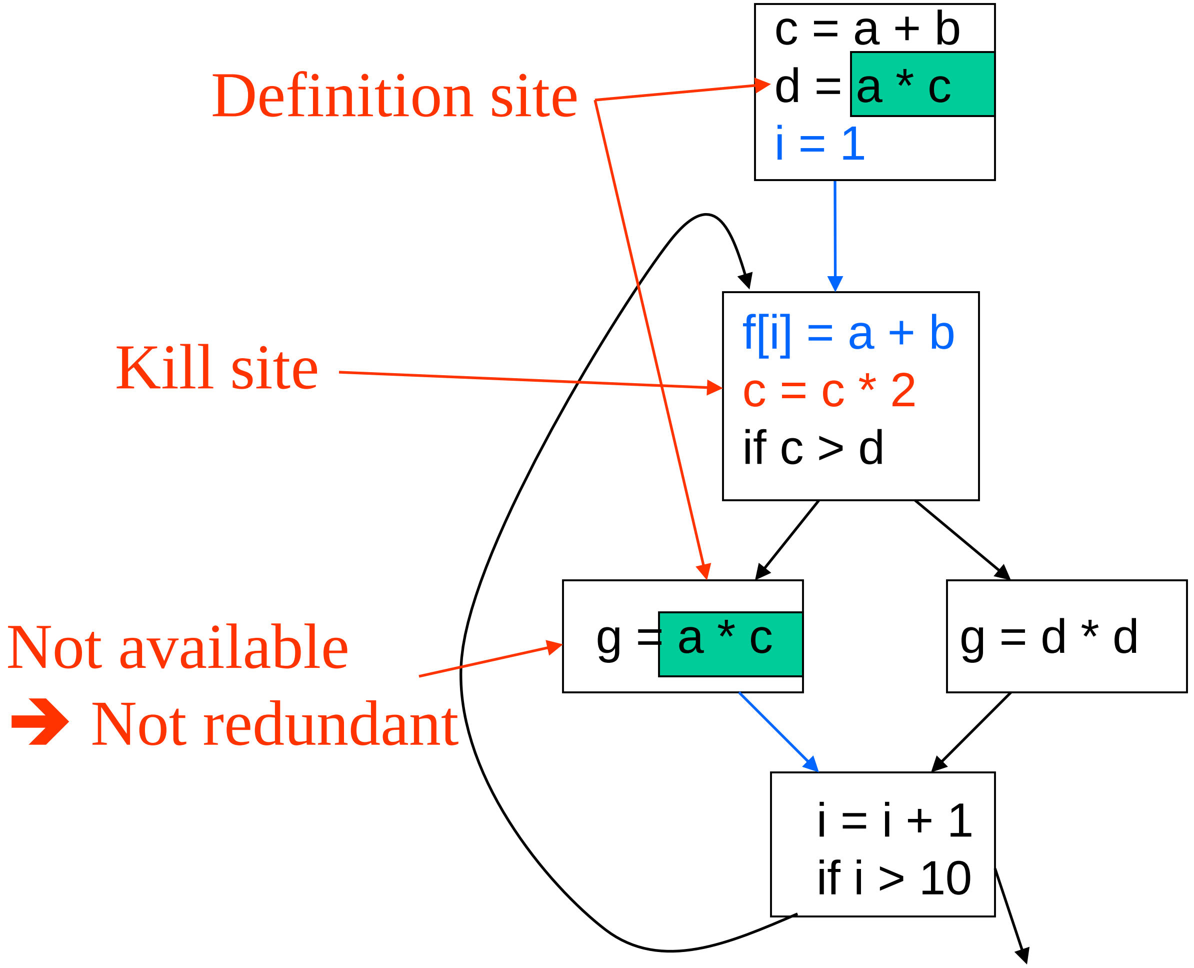
Redundant Expressions



Candidates:

a + b
a * c
d * d
c * 2
i + 1

Redundant Expressions

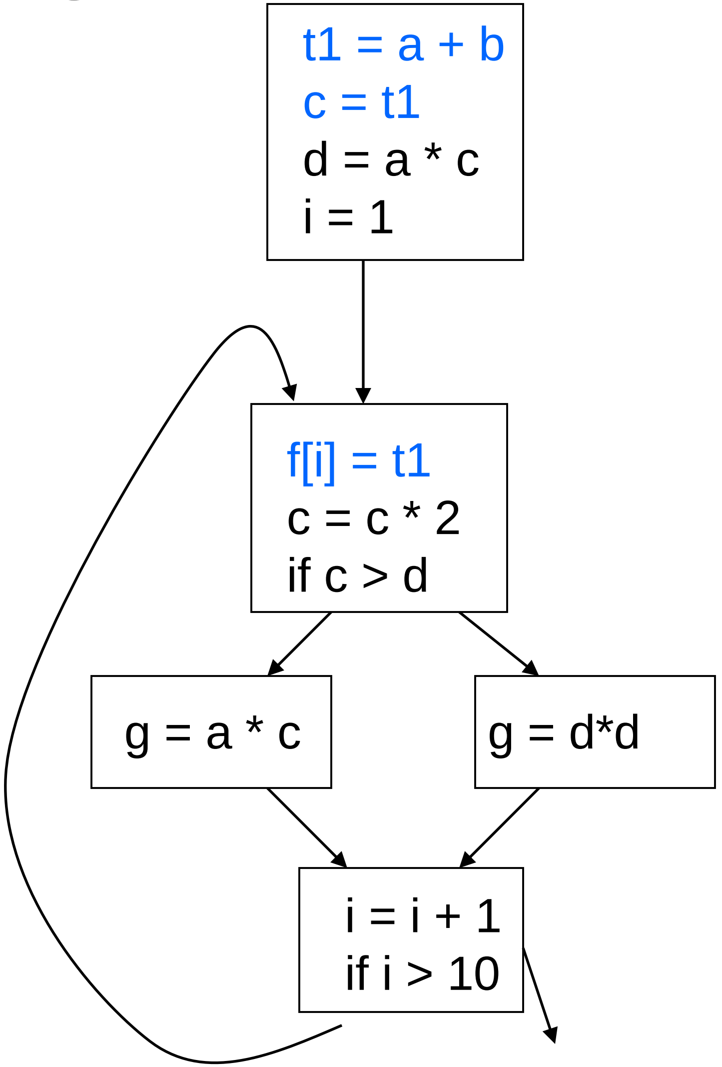


Candidates:
a + b
a * c
d * d
c * 2
i + 1

Redundant Expressions

- An expression *e* is defined at some point *p* in the CFG if its value is computed at *p*. (definition site)
- An expression *e* is killed at point *p* in the CFG if one or more of its operands is defined at *p*. (kill site)
- An expression is ***available*** at point *p* in a CFG if every path leading to *p* contains a prior definition of *e* and *e* is not killed between that definition and *p*.

Removing Redundant Expressions



Candidates:

a + b

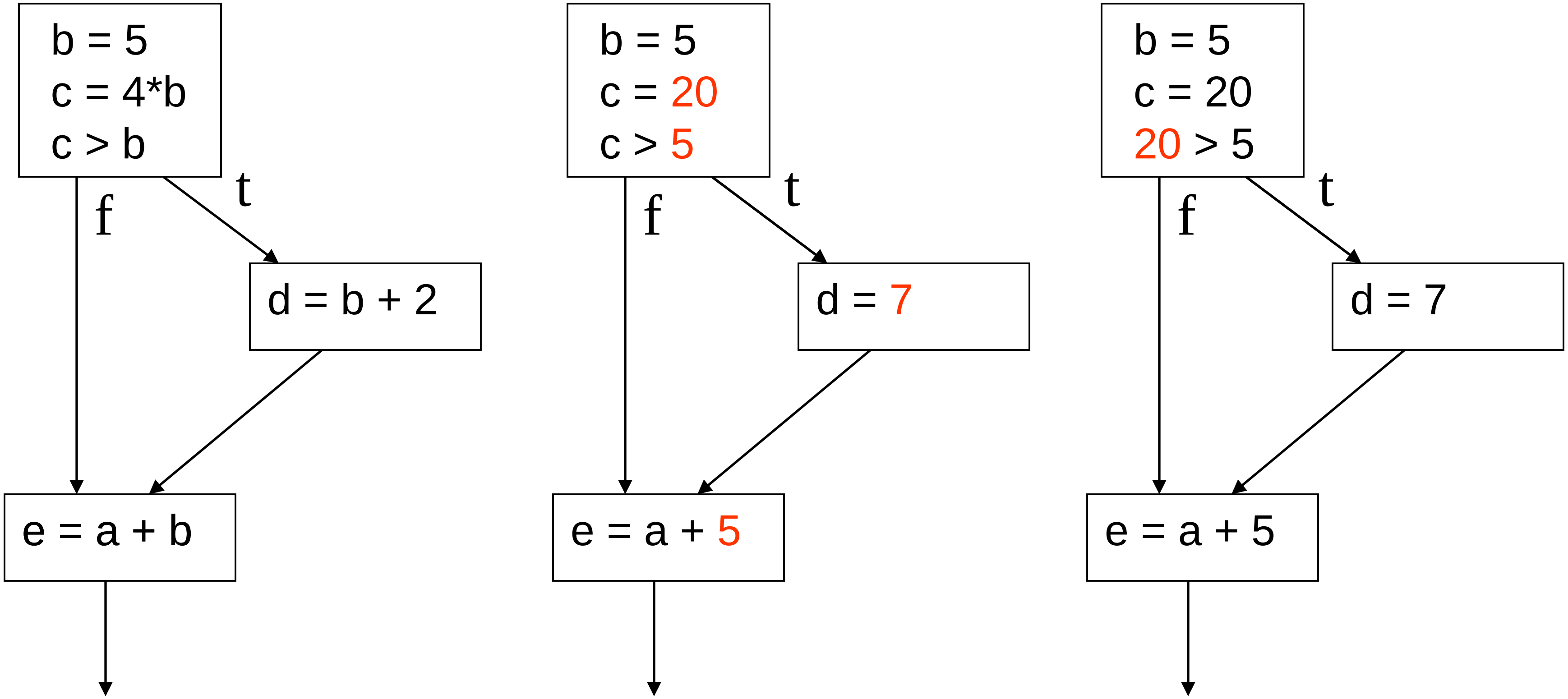
~~a * c~~

~~d * d~~

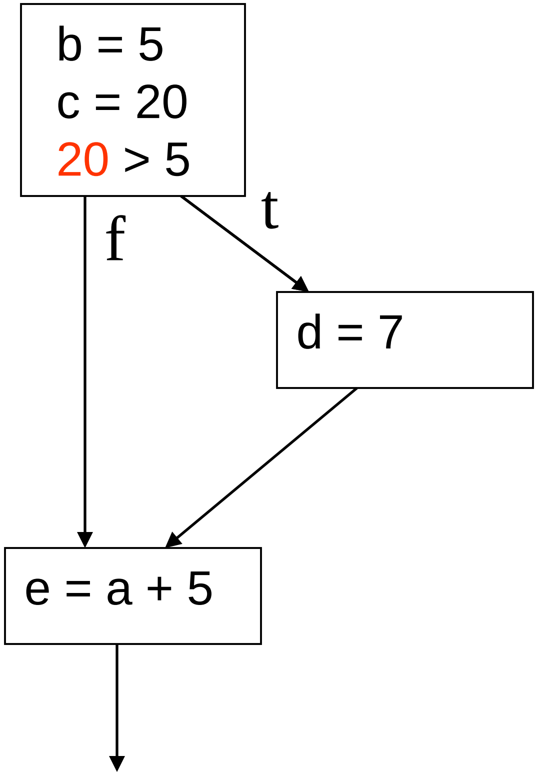
~~c * 2~~

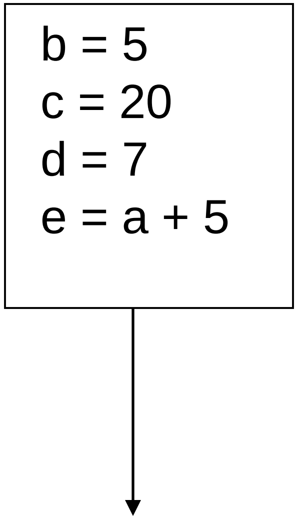
~~i + 1~~

Constant Propagation

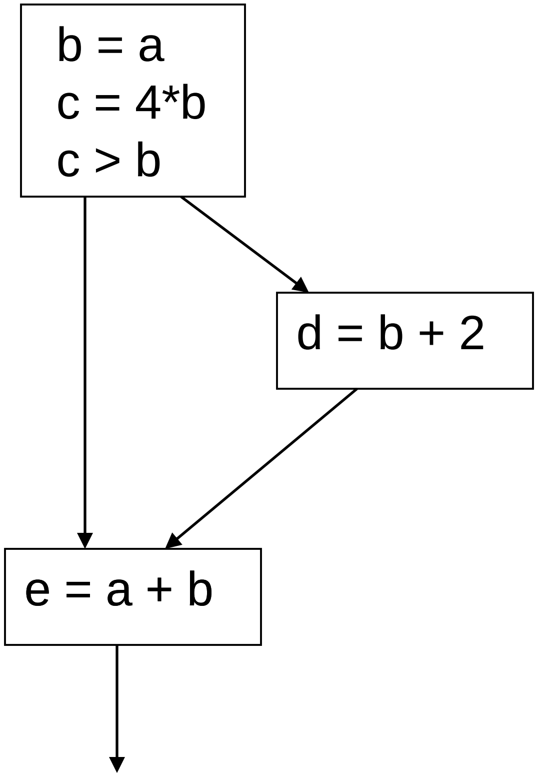


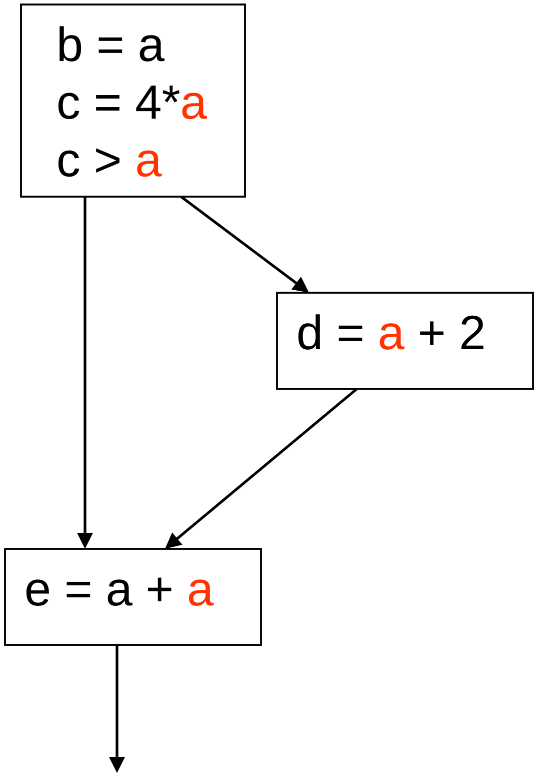
Constant Propagation





Copy Propagation





Code Motion

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

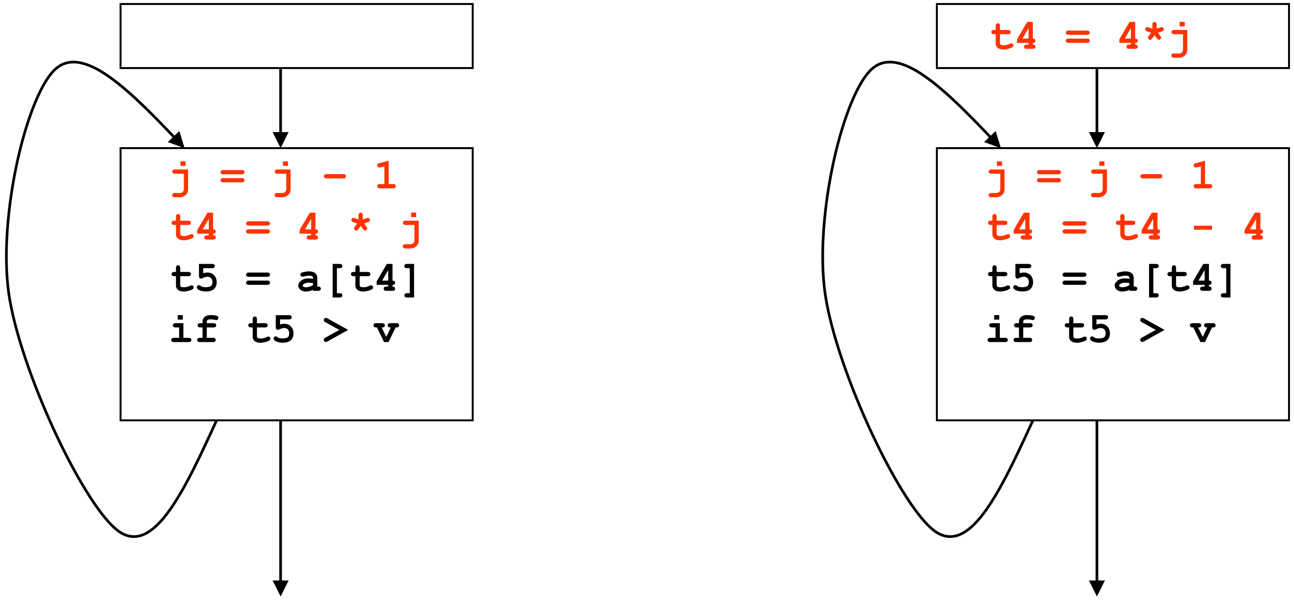
```
L1:
    t1 = limit - 2
    if (i > t1) goto L2
    body of loop
    goto L1
L2:
```

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

```
    t1 = limit - 2
L1:
    if (i > t1) goto L2
    body of loop
    goto L1
L2:
```

Strength Reduction

- Induction Variables control loop iterations



Loop Unrolling

- Replicating the body of the loop to reduce the required no of tests if the no of iterations are constant
- Ex

```
I=1;
while(I<=100)
{
X[I]=0;
I++;
}
```

```
I=1;
while(I<=100)
{
X[I]=0;
I++;
X[I]=0;
I++;
}
```

Loop Jamming

- Loop jamming is a technique that merges the bodies of two loops if the two loops have the same no of iterations and they use the same indices.

```
{
for(I=0;I<10;I++)
    for(J=0;J<10;J++)
            X[I,J]=0;
    for(I=0;I<10;I++)
            X[I,I]=1;
}
```

→

```
{
for(I=0;I<10;I++)
   {
    for(J=0;J<10;J++)
            X[I,J]=0;
     X[I,I]=1;
     }
}
```

Example – quick sort

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

Three Address Code of Quick Sort

1	$i = m - 1$	16	$t_7 = 4 * I$
2	$j = n$	17	$t_8 = 4 * j$
3	$t_1 = 4 * n$	18	$t_9 = a[t_8]$
4	$v = a[t_1]$	19	$a[t_7] = t_9$
5	$i = i + 1$	20	$t_{10} = 4 * j$
6	$t_2 = 4 * i$	21	$a[t_{10}] = x$
7	$t_3 = a[t_2]$	22	goto (5)
8	if $t_3 < v$ goto (5)	23	$t_{11} = 4 * I$
9	$j = j – 1$	24	$x = a[t_{11}]$
10	$t_4 = 4 * j$	25	$t_{12} = 4 * i$
11	$t_5 = a[t_4]$	26	$t_{13} = 4 * n$
12	if $t_5 > v$ goto (9)	27	$t_{14} = a[t_{13}]$
13	if $i >= j$ goto (23)	28	$a[t_{12}] = t_{14}$
14	$t_6 = 4 * i$	29	$t_{15} = 4 * n$
15	$x = a[t_6]$	30	$a[t_{15}] = x$

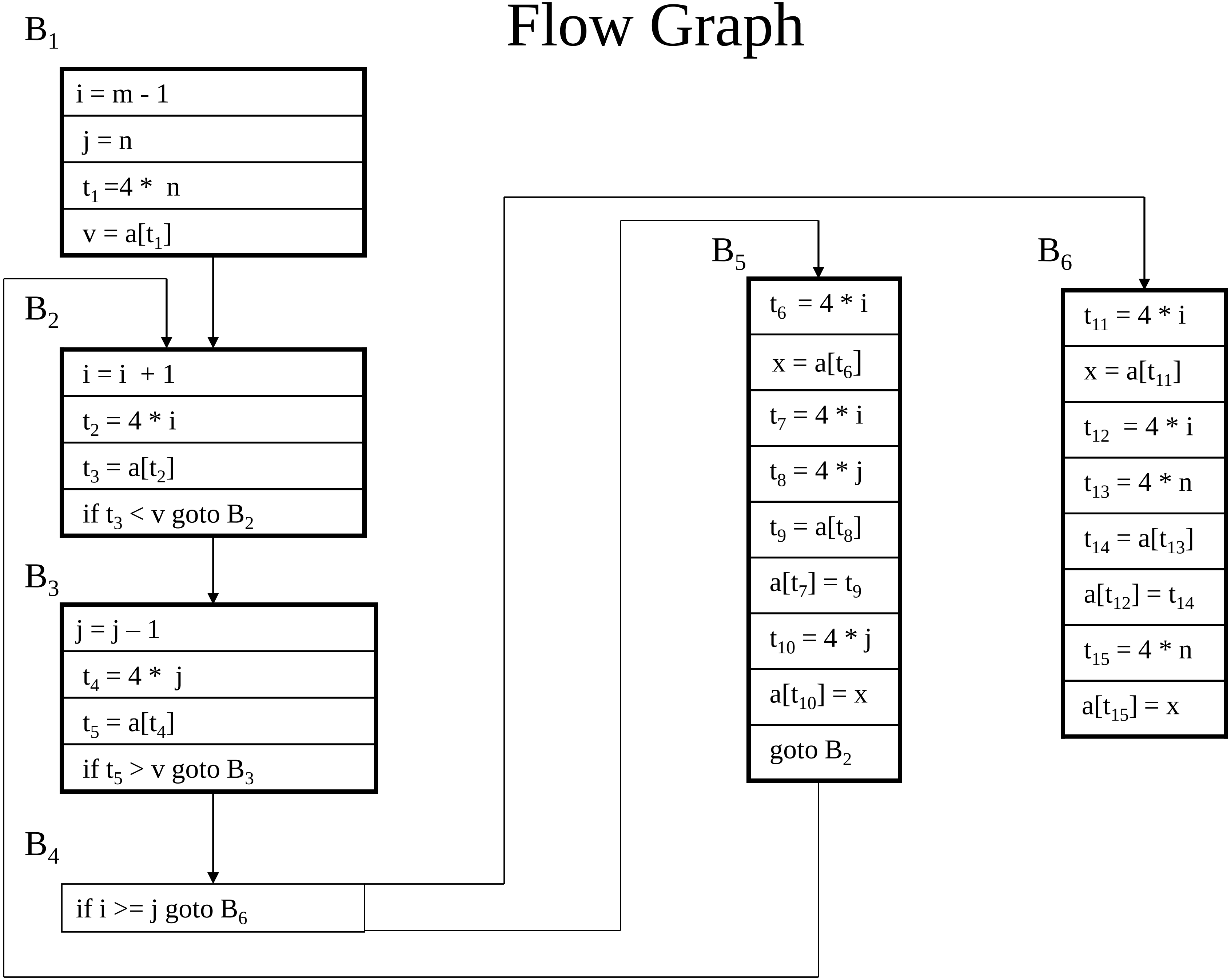
Each array element reserves 4 bytes

Find The Basic Block

1	$i = m - 1$
2	$j = n$
3	$t_1 = 4 * n$
4	$v = a[t_1]$
5	$i = i + 1$
6	$t_2 = 4 * i$
7	$t_3 = a[t_2]$
8	if $t_3 < v$ goto (5)
9	$j = j - 1$
10	$t_4 = 4 * j$
11	$t_5 = a[t_4]$
12	if $t_5 > v$ goto (9)
13	if $i >= j$ goto (23)
14	$t_6 = 4 * i$
15	$x = a[t_6]$

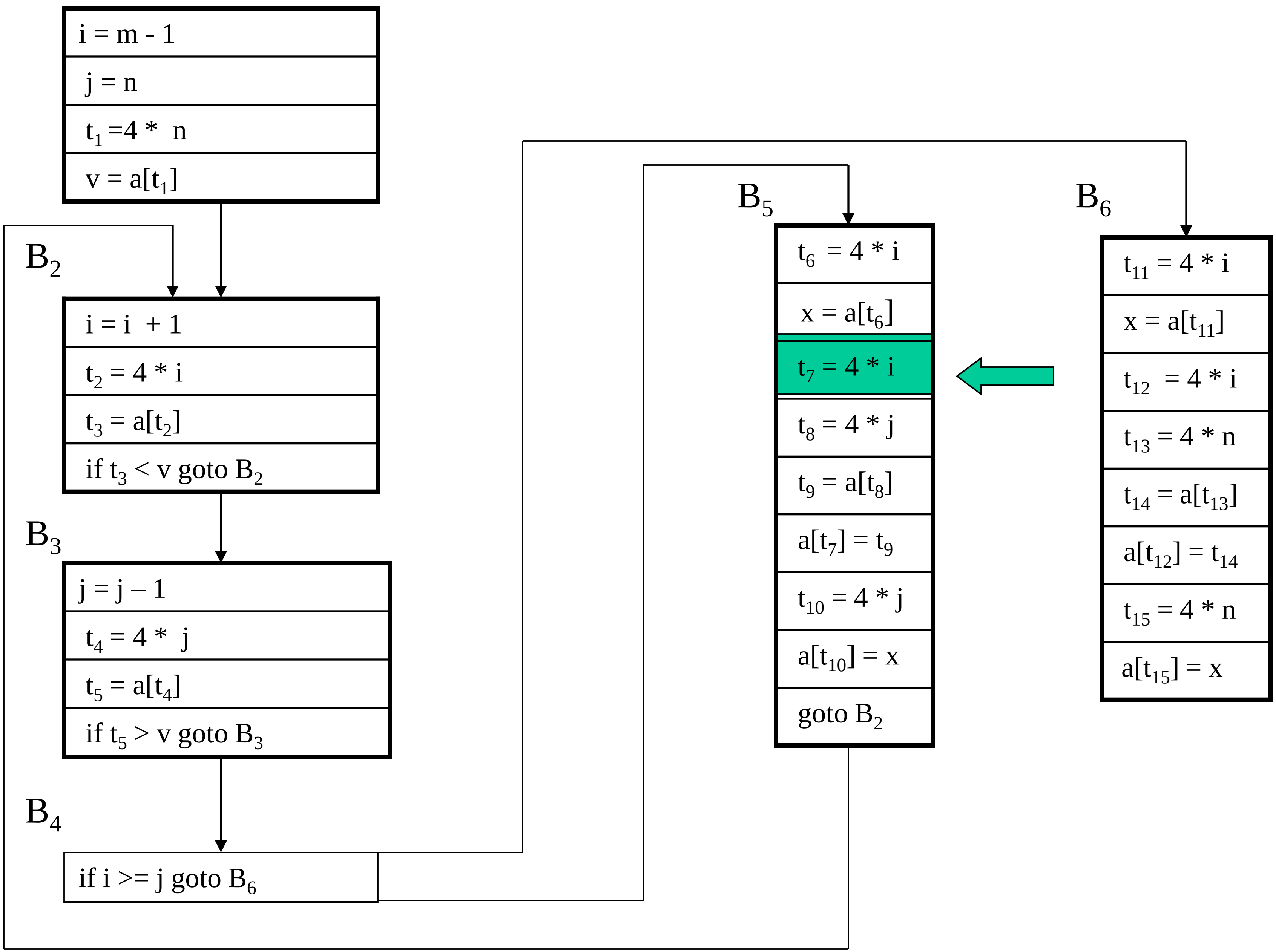
16	$t_7 = 4 * I$
17	$t_8 = 4 * j$
18	$t_9 = a[t_8]$
19	$a[t_7] = t_9$
20	$t_{10} = 4 * j$
21	$a[t_{10}] = x$
22	goto (5)
23	$t_{11} = 4 * i$
24	$x = a[t_{11}]$
25	$t_{12} = 4 * i$
26	$t_{13} = 4 * n$
27	$t_{14} = a[t_{13}]$
28	$a[t_{12}] = t_{14}$
29	$t_{15} = 4 * n$
30	$a[t_{15}] = x$

Flow Graph

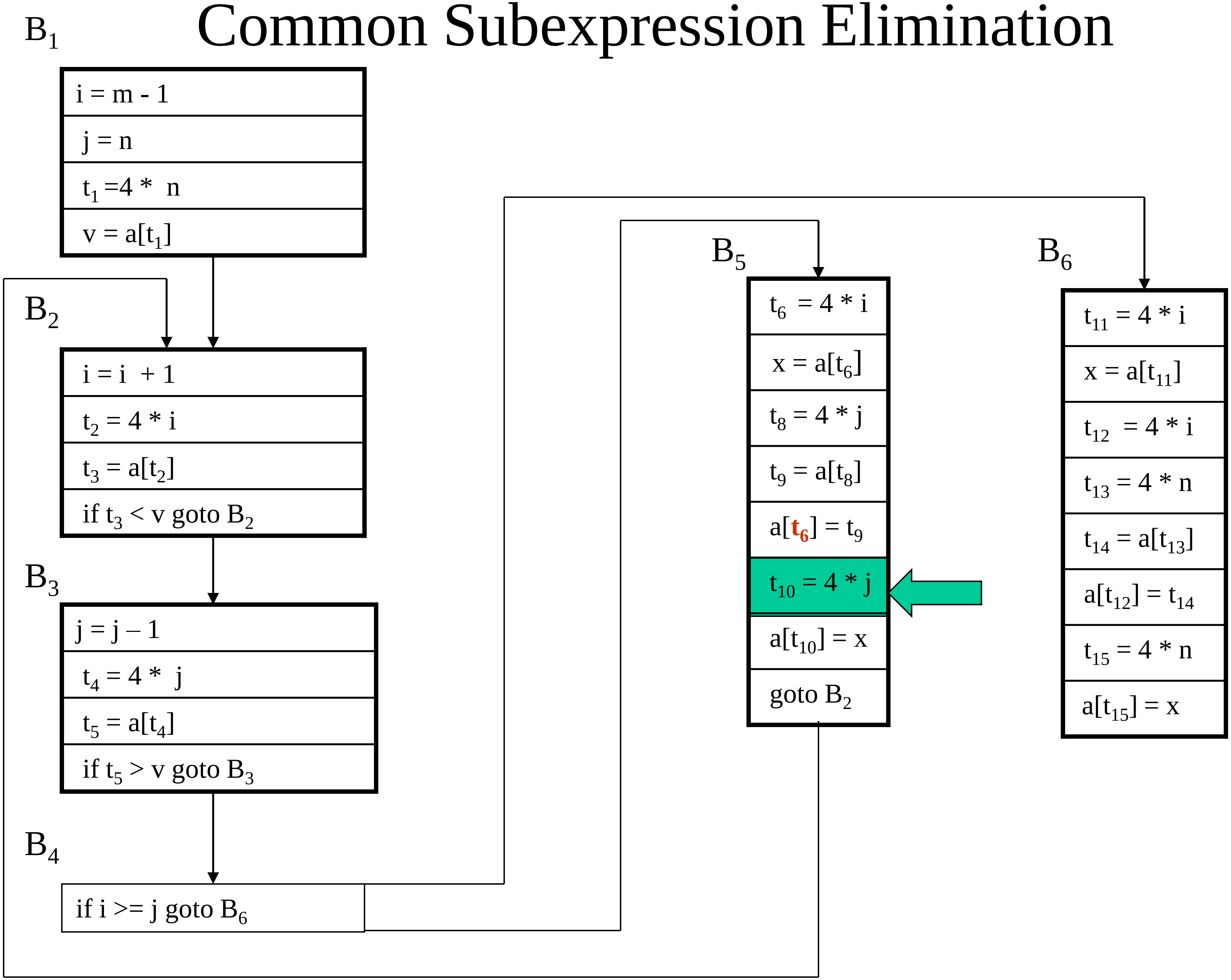


B1
i = m - 1
j = n
t1 =4 * n
v = a[t1]
B2
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B3
j = j – 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B4
if i >= j goto B6
B5
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
B6
t11 = 4 * i
x = a[t11]
t12 = 4 *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

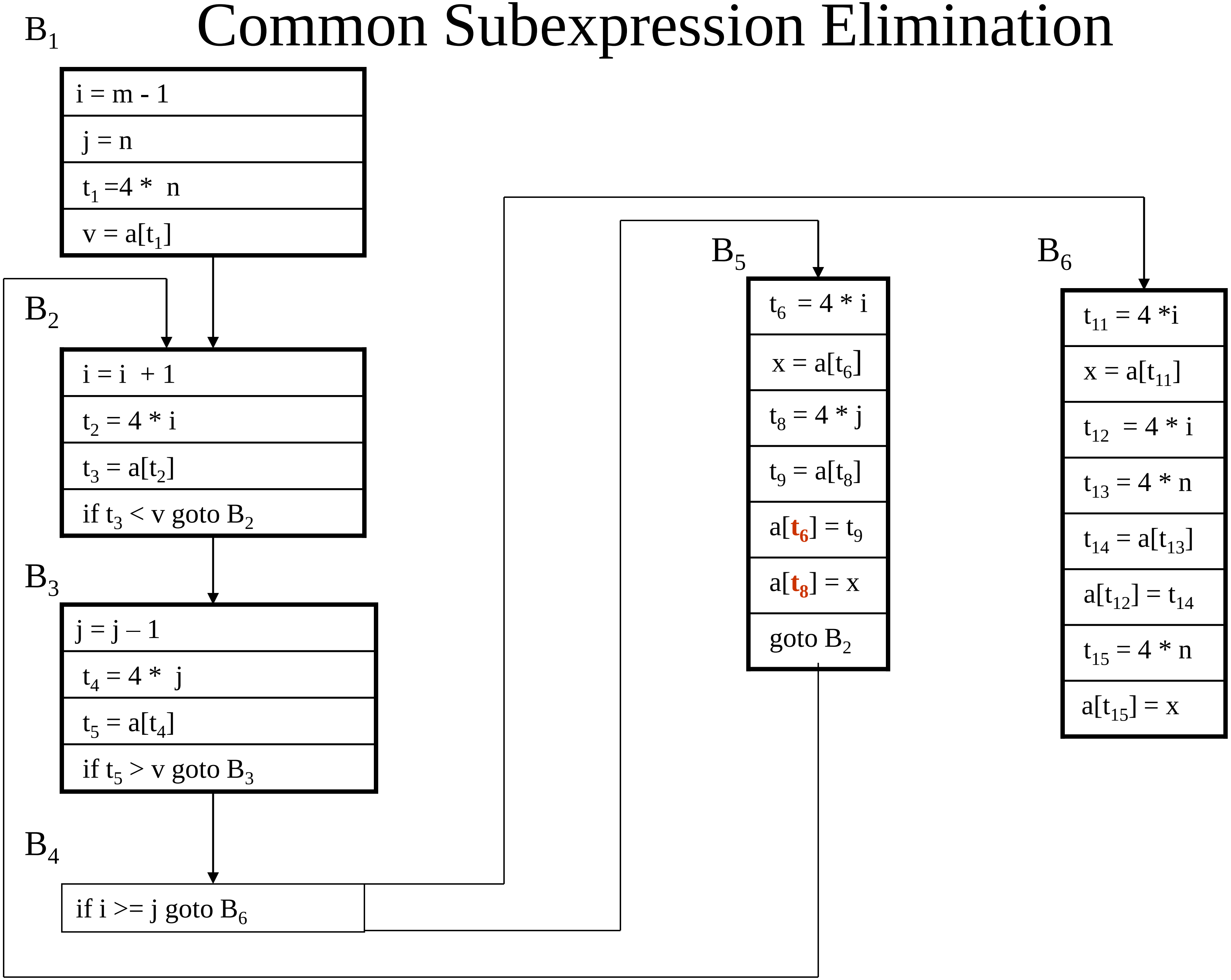
Common Subexpression Elimination



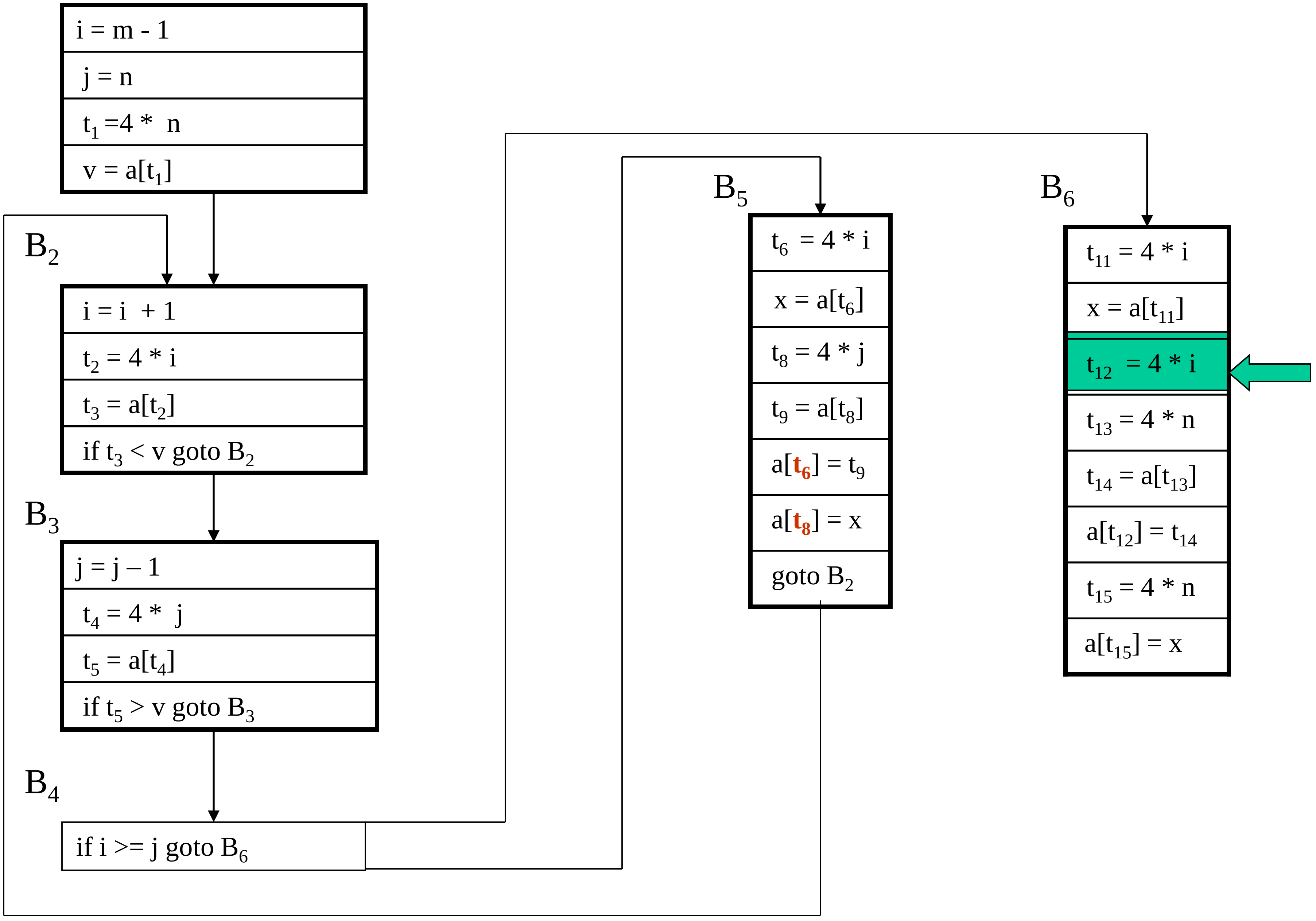
Common Subexpression Elimination



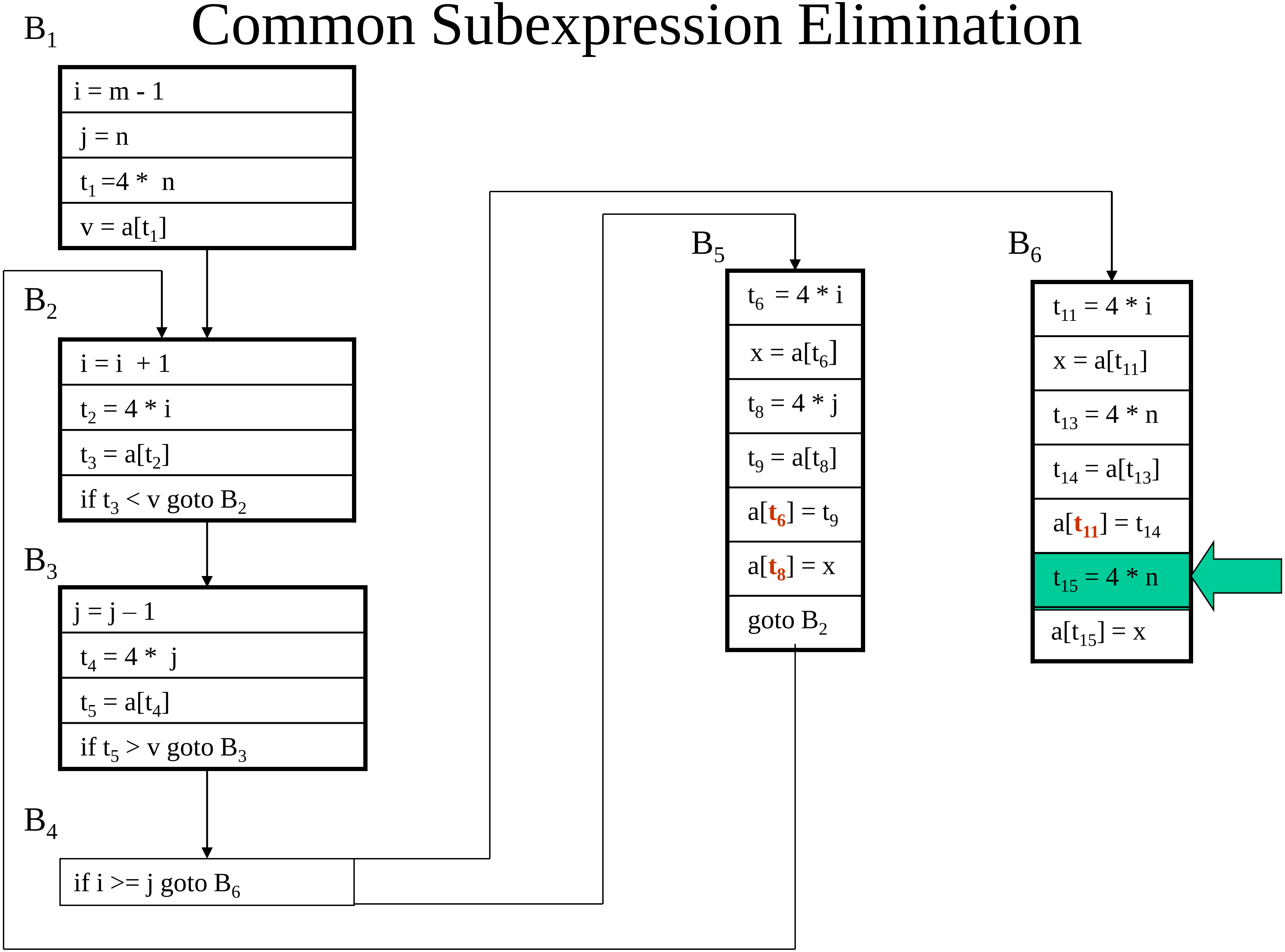
Common Subexpression Elimination



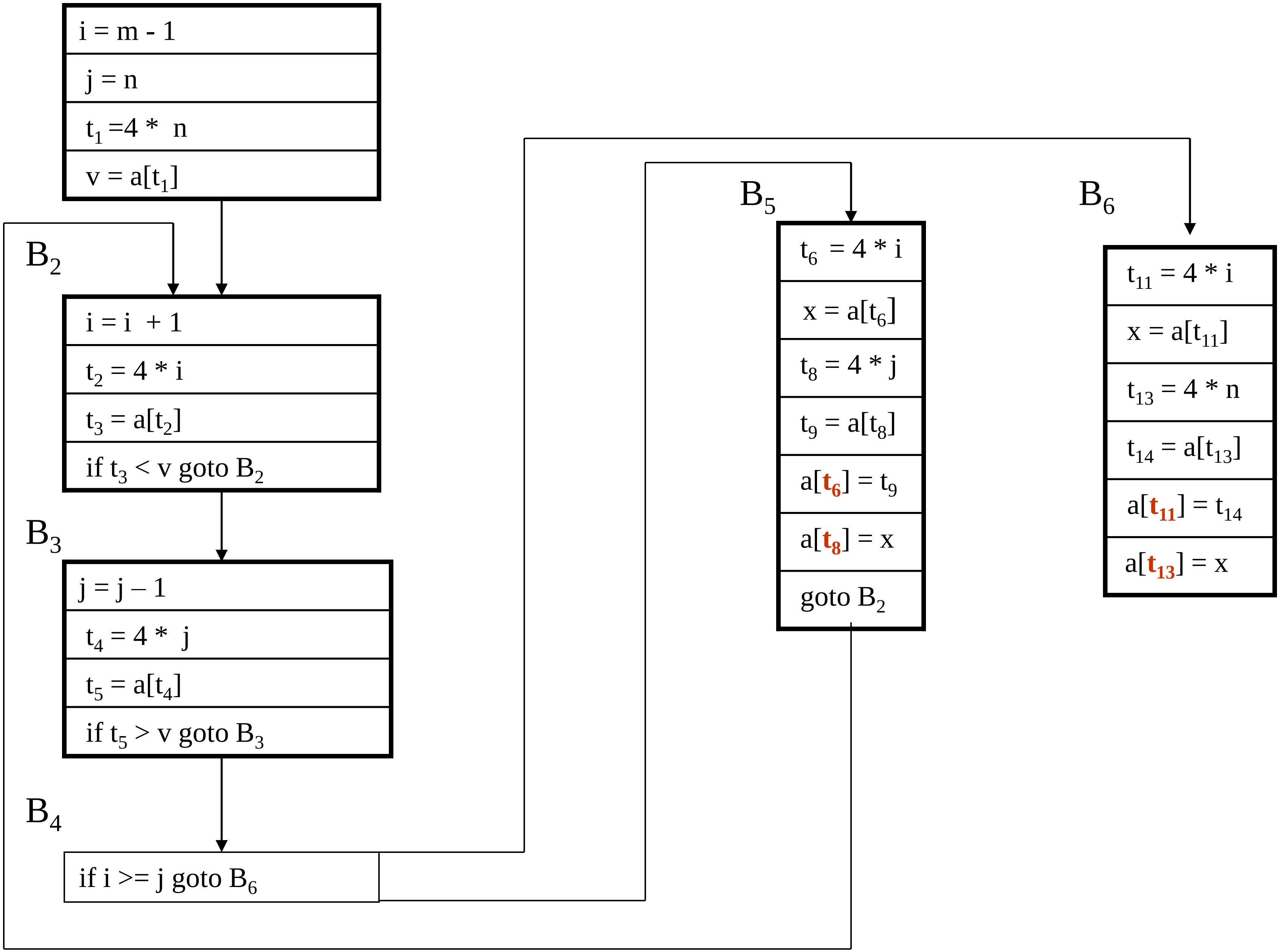
Common Subexpression Elimination



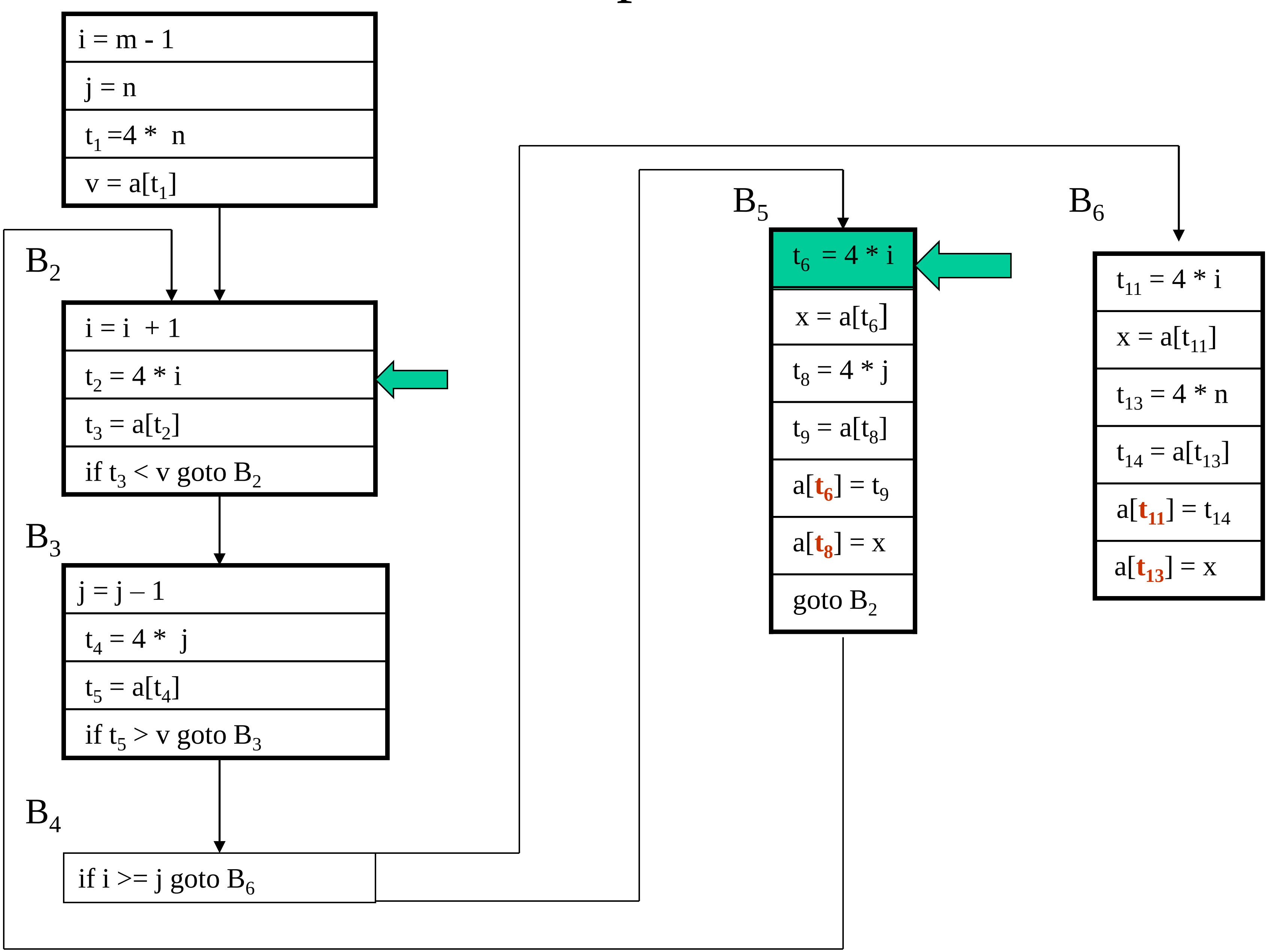
Common Subexpression Elimination



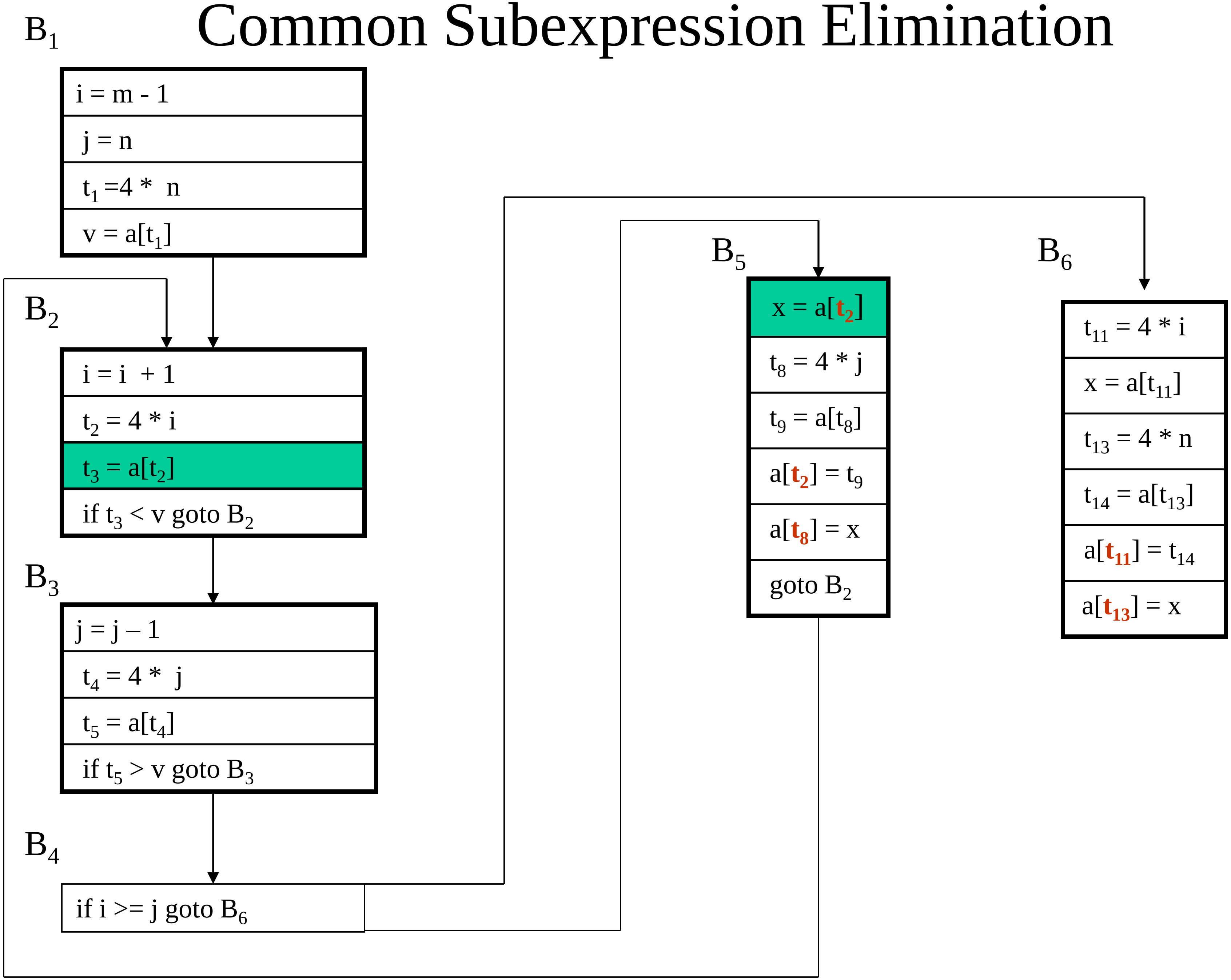
Common Subexpression Elimination



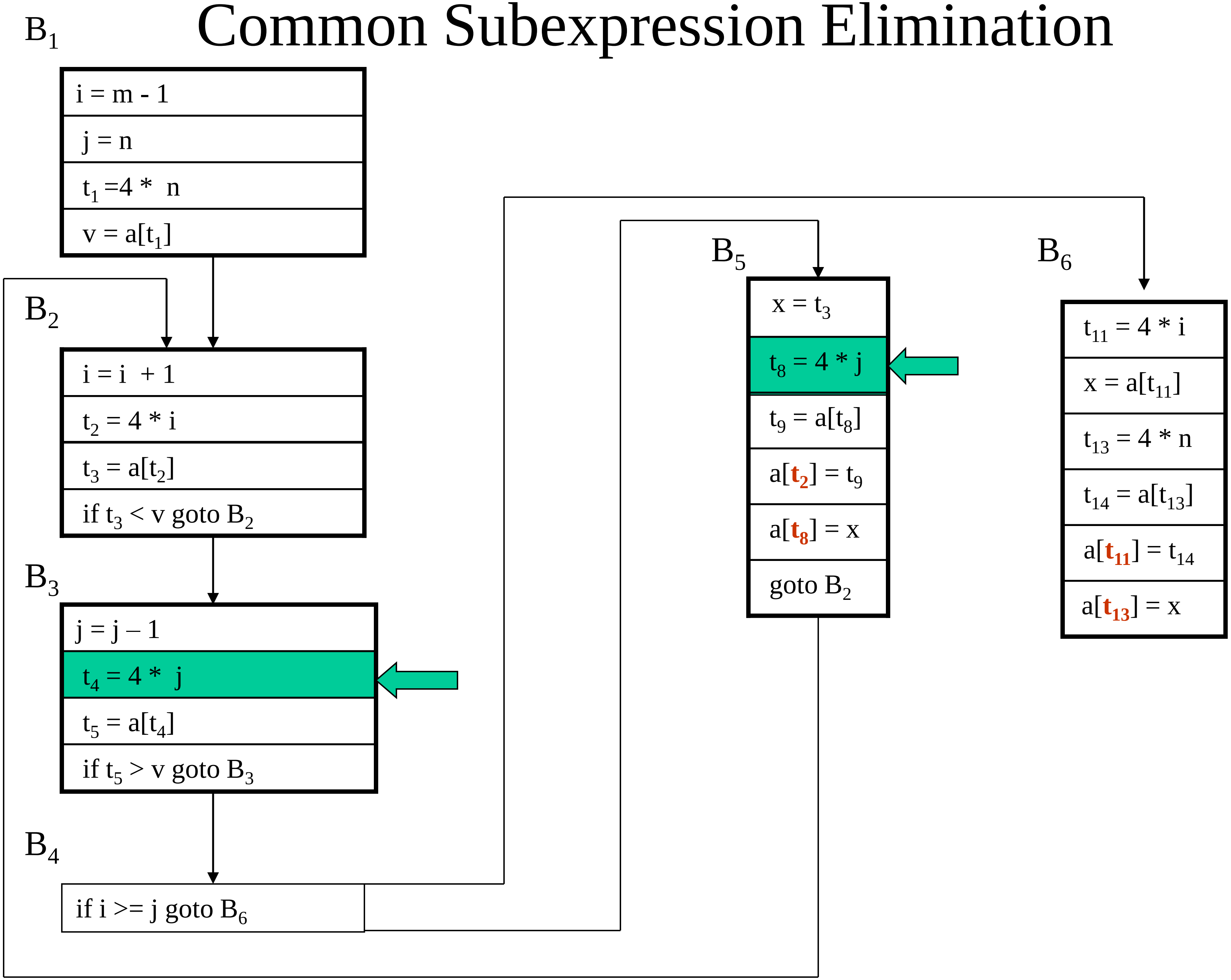
Common Subexpression Elimination



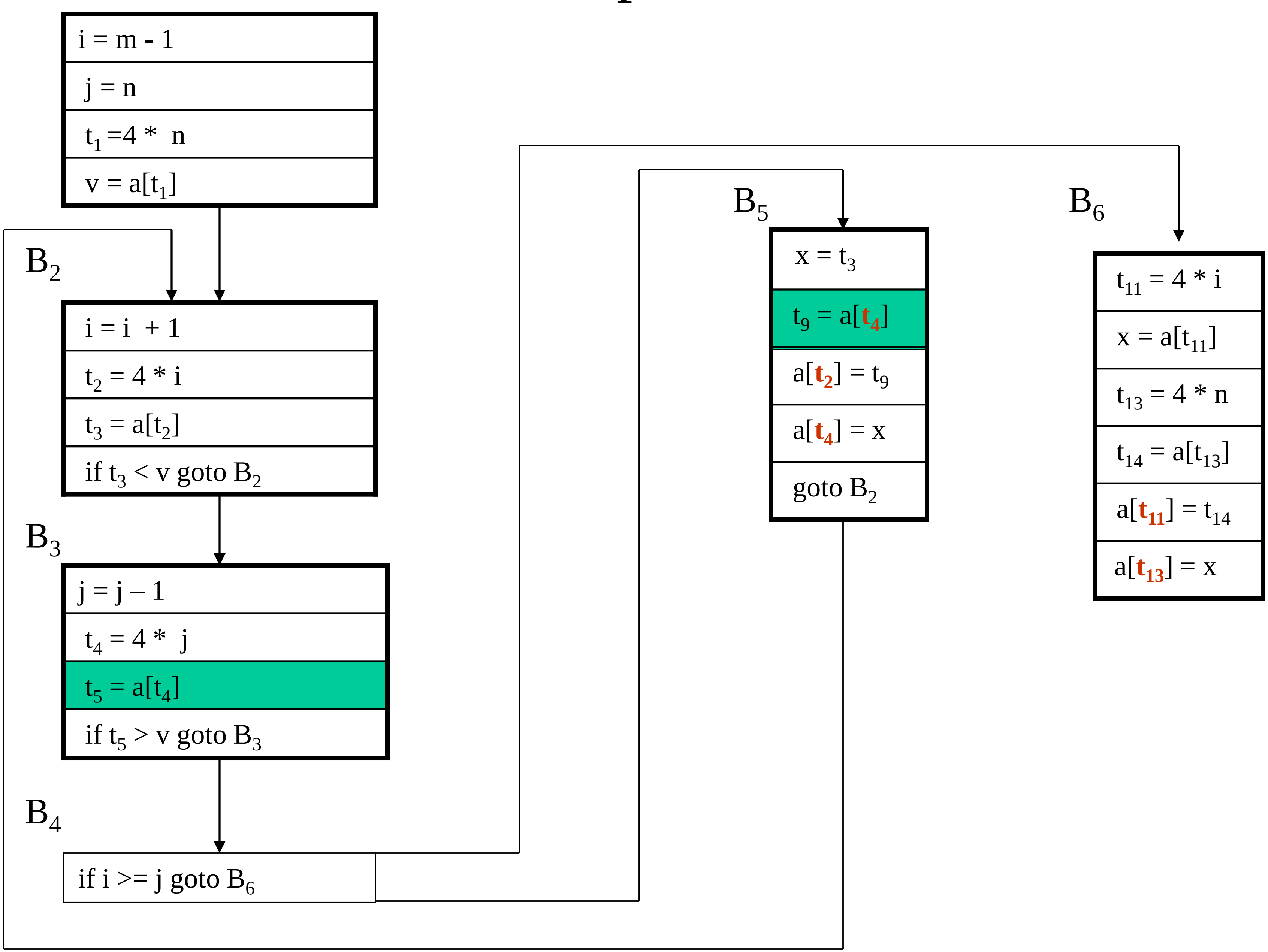
Common Subexpression Elimination



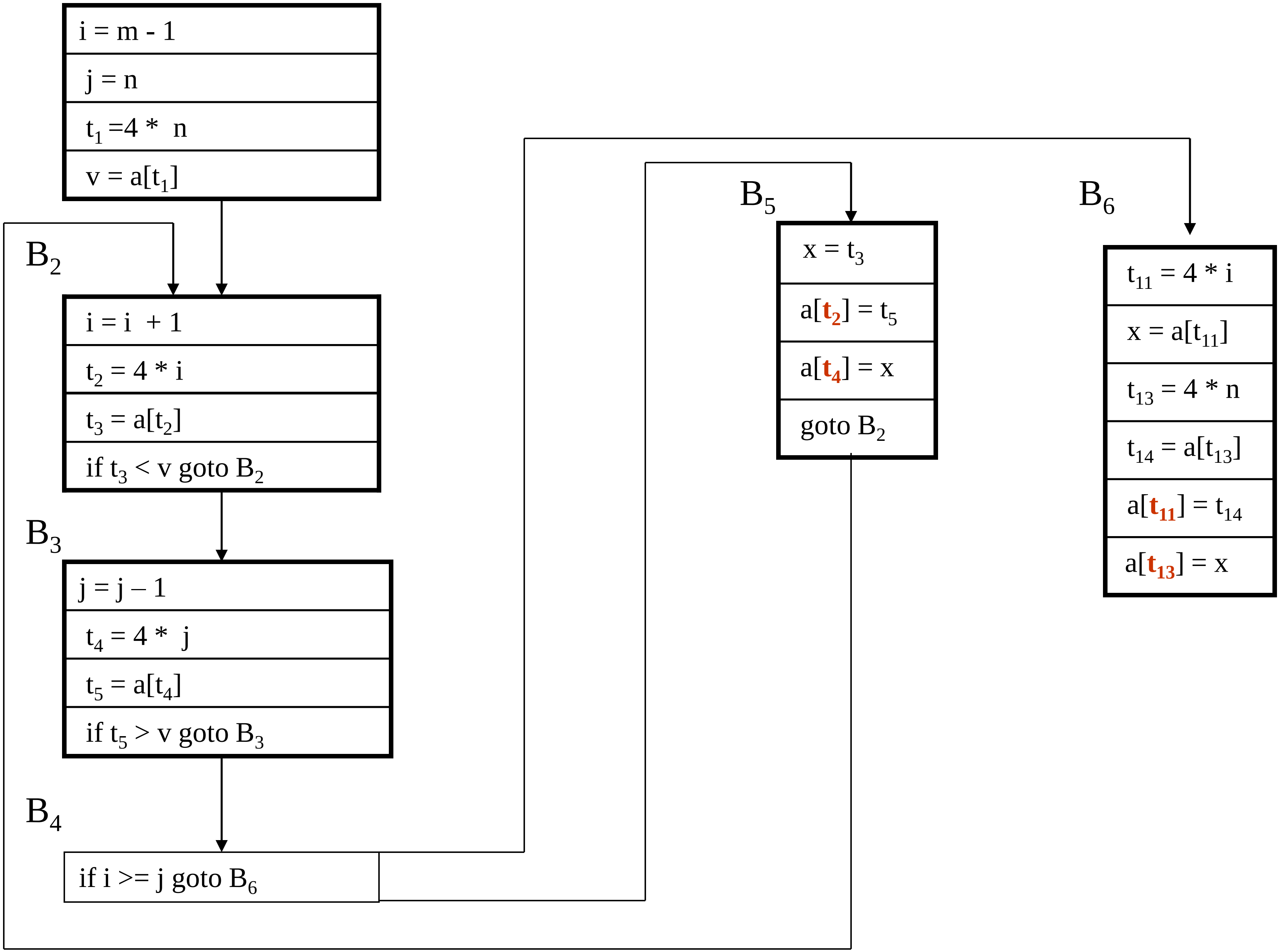
Common Subexpression Elimination



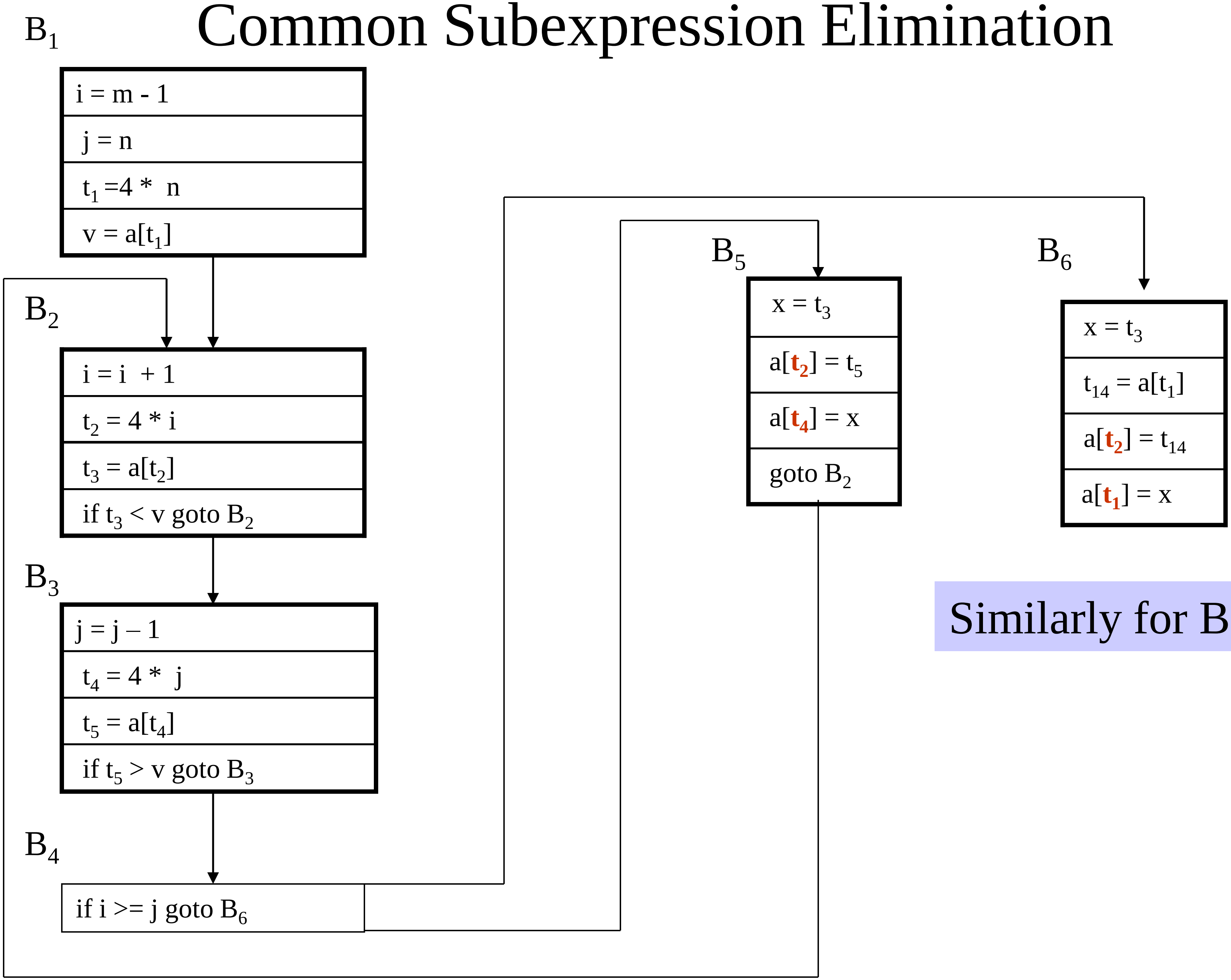
Common Subexpression Elimination



Common Subexpression Elimination

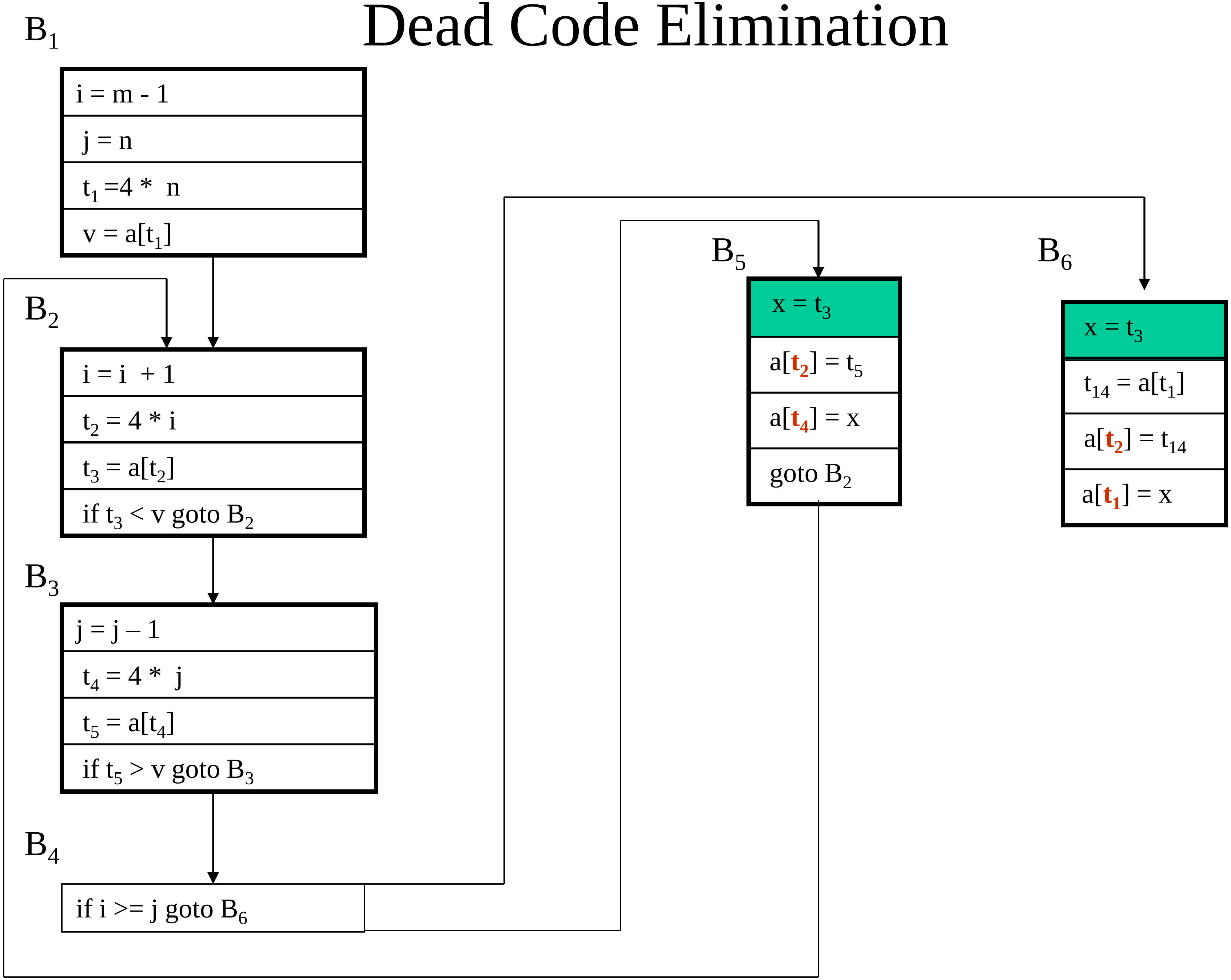


Common Subexpression Elimination

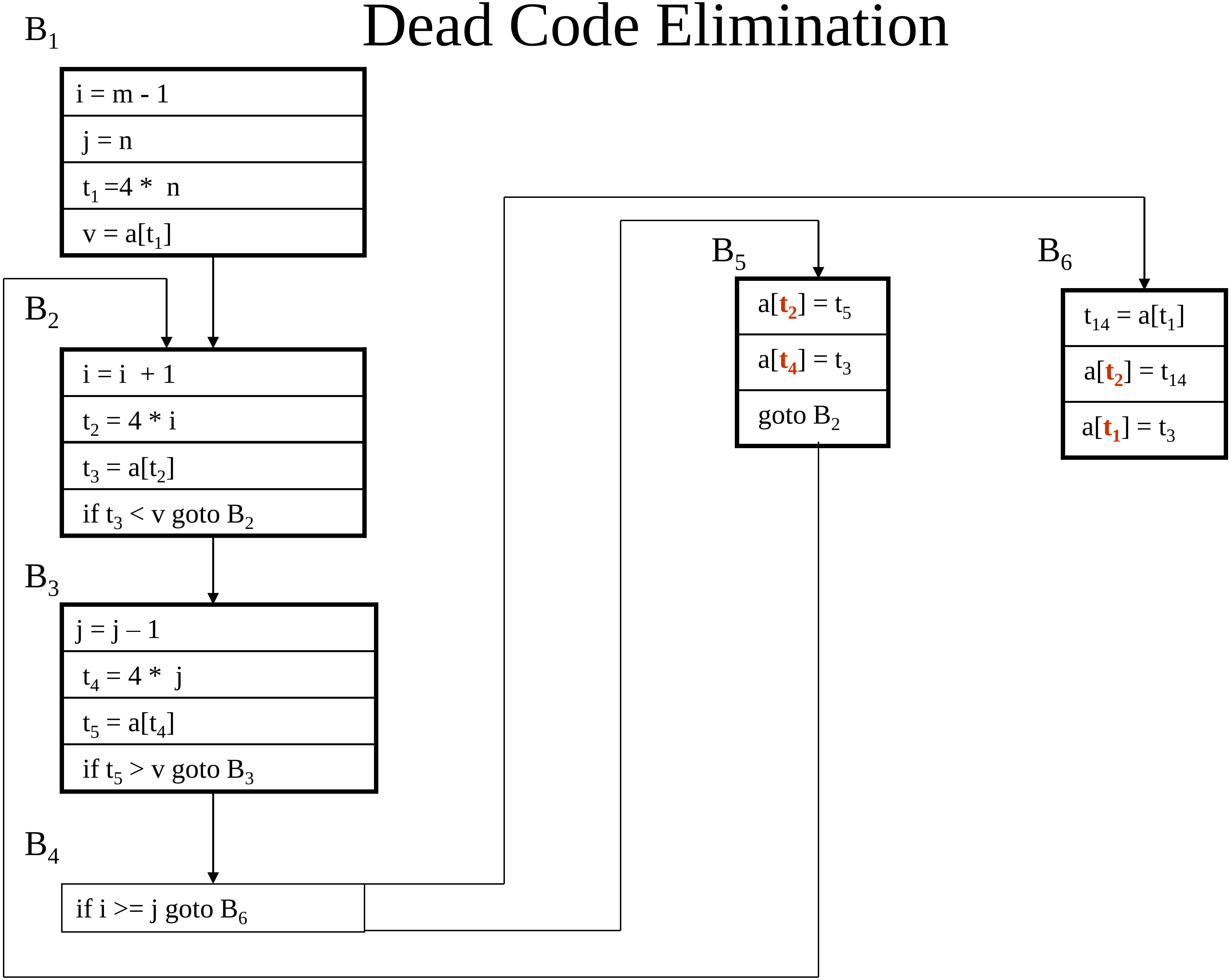


Similarly for B_6

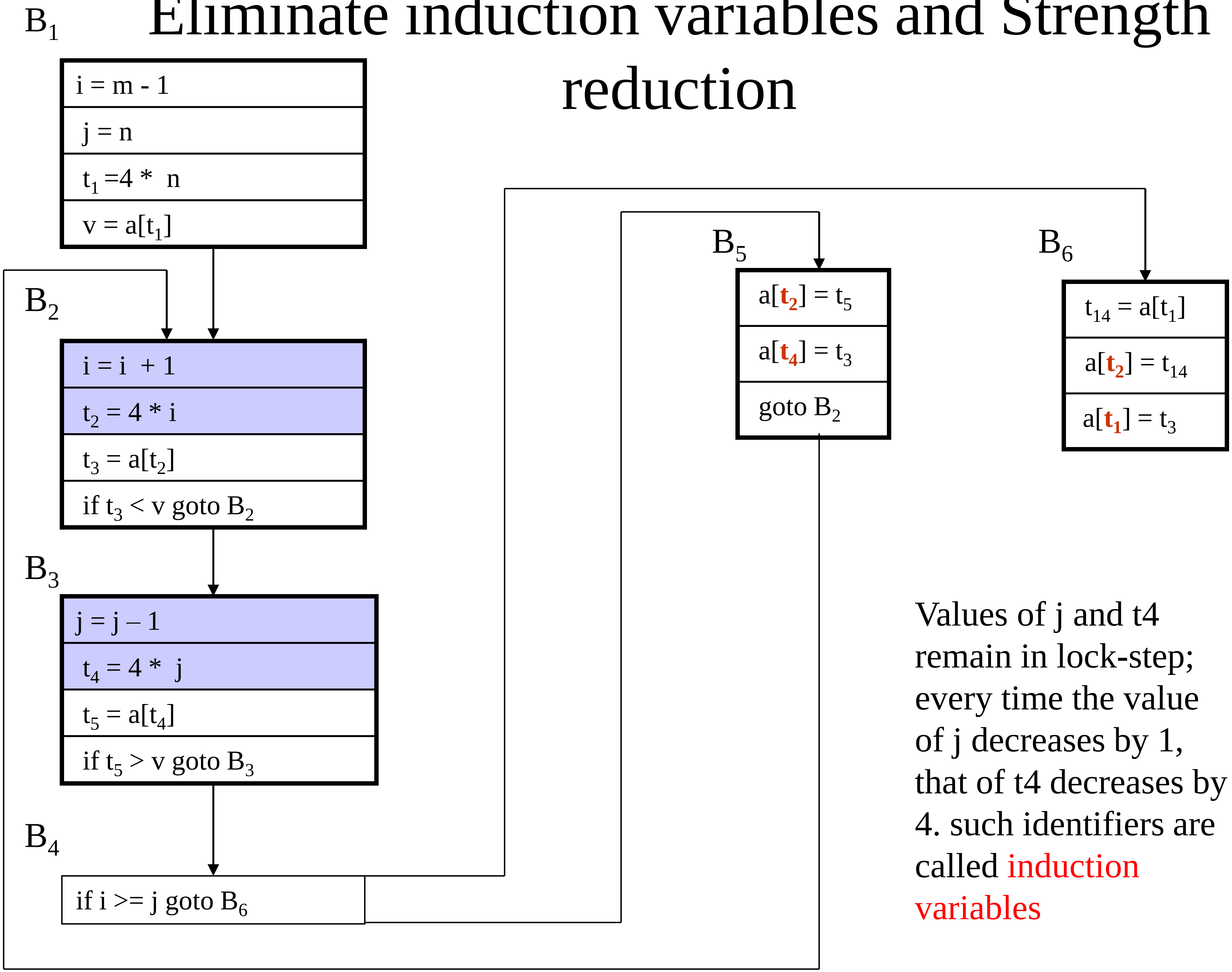
Dead Code Elimination



Dead Code Elimination

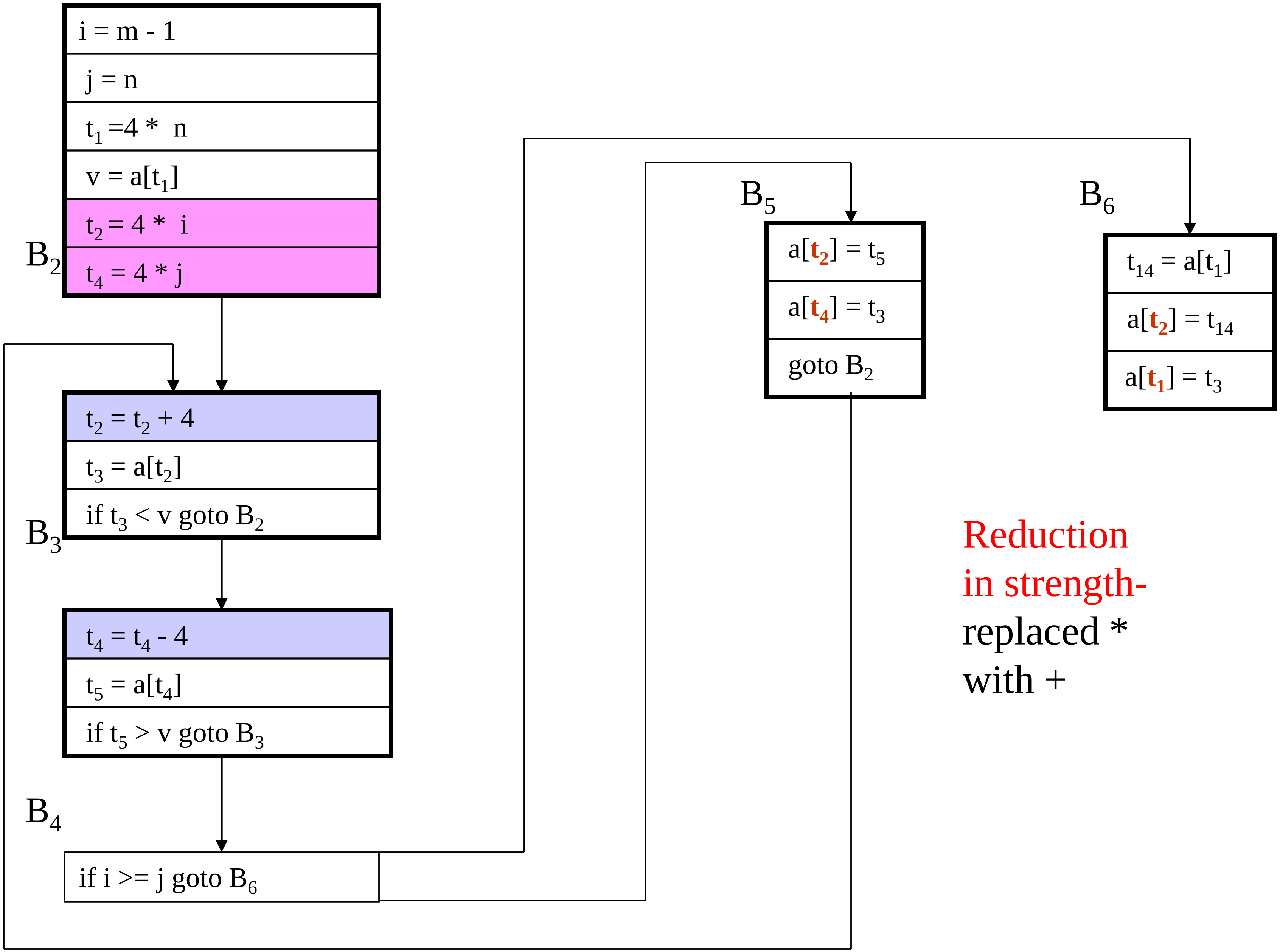


Eliminate induction variables and Strength reduction



Values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4. such identifiers are called induction variables

Reduction in Strength



Reduction in strength-
replaced * with +