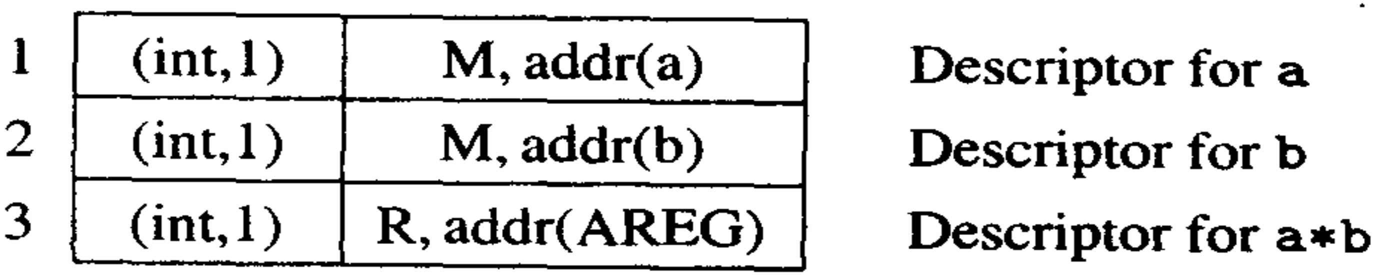
Compilers : Synthesis Phase

Intermediate Code Generation

- *Intermediate codes* are machine independent codes, close to machine instructions.
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
- Operand Descriptors and Register Descriptors.
- Tuples and Quadruples for intermediate code generation.

Operand descriptors

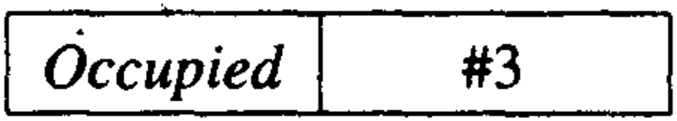
- An operand descriptor has the following fields:
  1. **Attributes:** Contains the subfields type, length and miscellaneous information
  2. **Addressability:** Specifies where the operand is located, and how it can be accessed. It has two subfields
     a. Addressability code: Takes the values 'M' (operand is in memory), and 'R' (operand is in register).
     b. Address: Address of a CPU register or memory word.



1	(int,1)	M, addr(a)	Descriptor for a
2	(int,1)	M, addr(b)	Descriptor for b
3	(int,1)	R, addr(AREG)	Descriptor for a*b

Register descriptors

- A register descriptor has two fields
  1. **Status:** Contains the code free or occupied to indicate register status.
  2. **Operand descriptor #:** If status = occupied, this field contains the descriptor for the operand contained in the register.
- Register descriptors are stored in an array called Register_descriptor.
- One register descriptor exists for each CPU register.



Occupied	#3

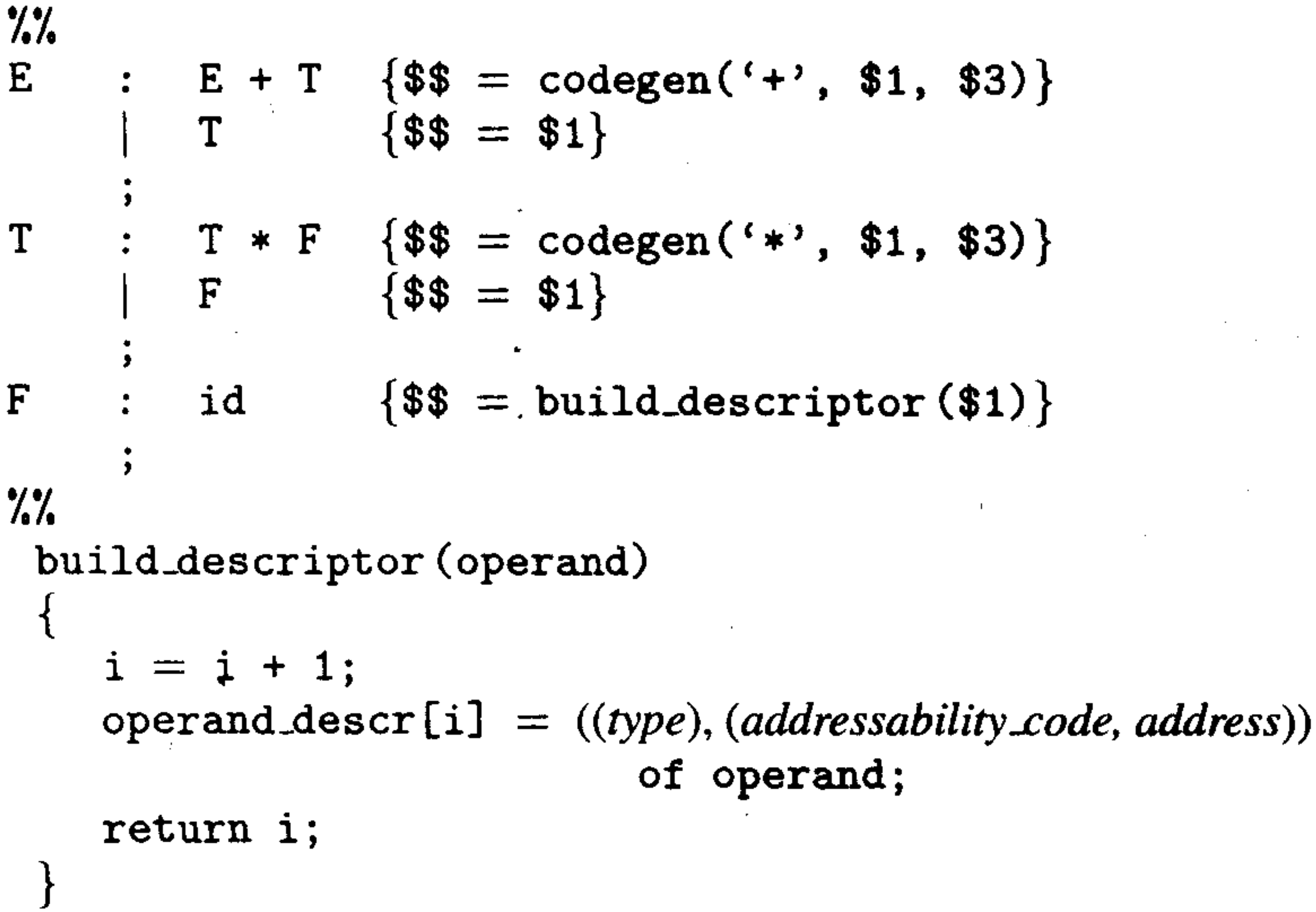
Generating an instruction

- When an operator OP_i is reduced by the parser, the function codegen is called with OP_i and descriptors of its operands as parameters.
- A single instruction can be generated to evaluate OP_i, if the descriptors indicate that one operand is in a register and the other is in memory.
- If both operands are in memory, an instruction is generated to move one of them into a register.
- This is followed by an instruction to evaluate OP_i.

Saving partial results

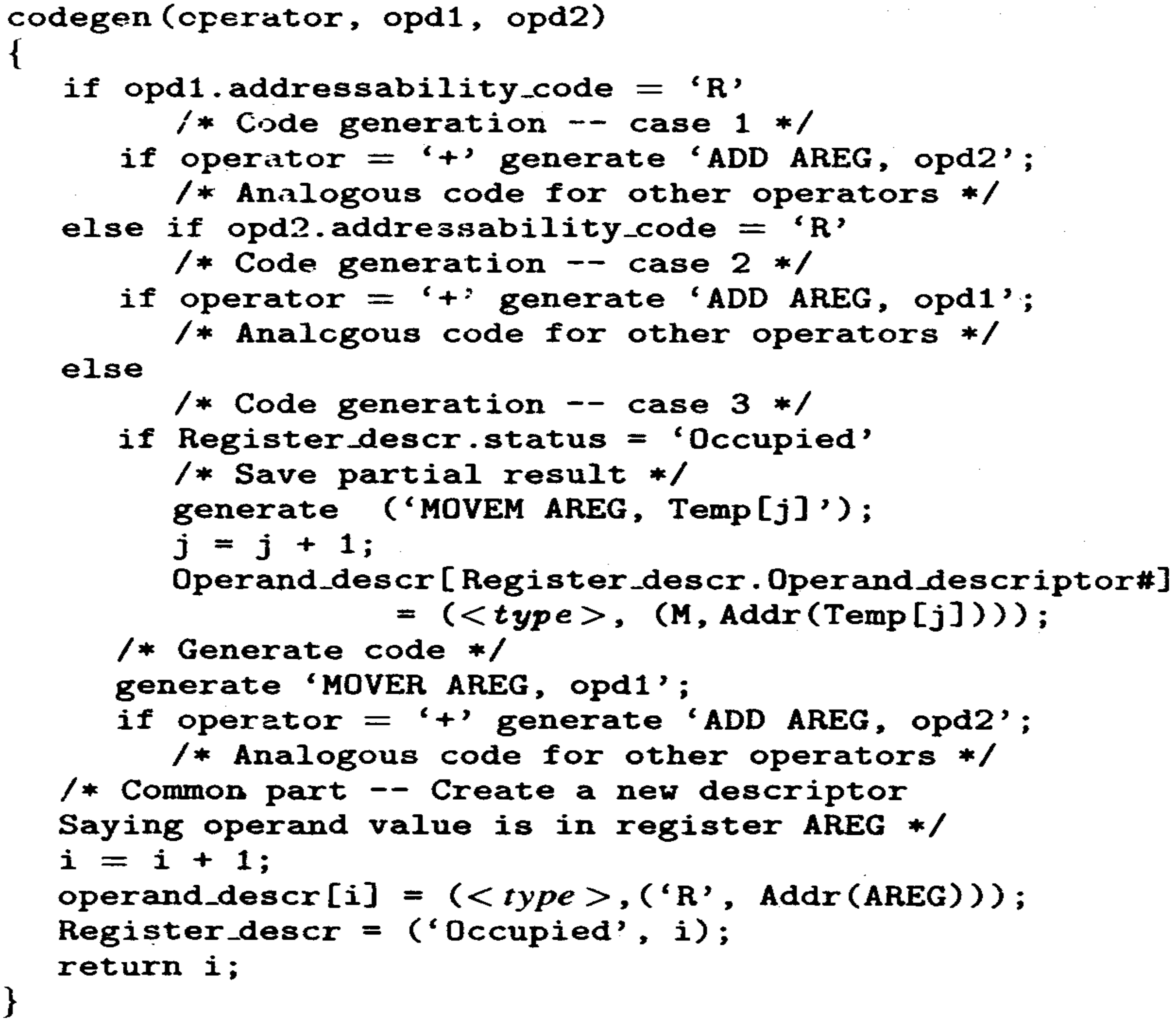
- If all registers are occupied (i.e. they contain partial results) when operator OP_i, is to be evaluated, a register *r* is freed by copying its contents into a temporary location in the memory.
- *r* is then used to evaluate operator OP_i.
- For simplicity we assume that an array temp is declared in the target program (i.e. in the generated code) to hold partial results.
- A partial result is always stored in the next free entry of temp.
- Note that when a partial result is moved to a temporary location, the descriptor of the partial result must change.
- The operand descriptor # field of the register descriptor is used to achieve this.

Skeleton of the code generator



```
%%
E    :   E + T   {$$ = codegen('+', $1, $3)}
     |   T       {$$ = $1}
     ;
T    :   T * F   {$$ = codegen('*', $1, $3)}
     |   F       {$$ = $1}
     ;
F    :   id      {$$ = build_descriptor($1)}
     ;
%%
 build_descriptor(operand)
 {
    i = i + 1;
    operand_descr[i] = ((type), (addressability_code, address))
                                    of operand;
    return i;
 }
```

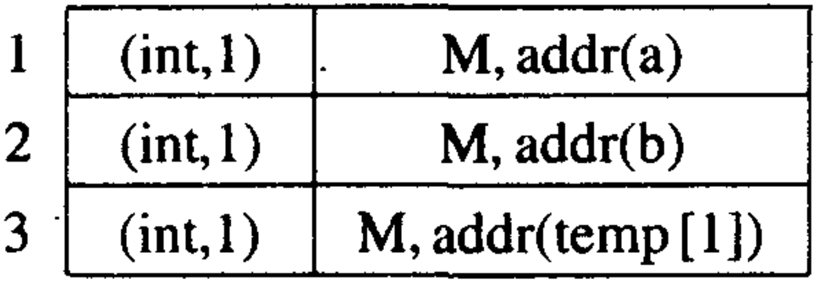
Code generation routine



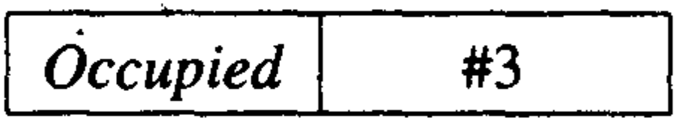
```
codegen(operator, opd1, opd2)
{
    if opd1.addressability_code = 'R'
            /* Code generation -- case 1 */
        if operator = '+' generate 'ADD AREG, opd2';
            /* Analogous code for other operators */
    else if opd2.addressability_code = 'R'
            /* Code generation -- case 2 */
        if operator = '+' generate 'ADD AREG, opd1';
            /* Analogous code for other operators */
    else
            /* Code generation -- case 3 */
        if Register_descr.status = 'Occupied'
            /* Save partial result */
            generate  ('MOVEM AREG, Temp[j]');
            j = j + 1;
            Operand_descr[Register_descr.Operand_descriptor#]
                        = (<type>, (M, Addr(Temp[j])));
        /* Generate code */
        generate 'MOVER AREG, opd1';
        if operator = '+' generate 'ADD AREG, opd2';
            /* Analogous code for other operators */
    /* Common part -- Create a new descriptor
    Saying operand value is in register AREG */
    i = i + 1;
    operand_descr[i] = (<type>,('R', Addr(AREG)));
    Register_descr = ('Occupied', i);
    return i;
}
```

Example

- Consider the expression a*b+c*d.
- After generating code for a*b, the operand and register descriptors would be as shown in figure.
- After the partial result a*b is moved to a temporary location, say temp [1], the operand descriptors must become to indicate that the value of the operand described by operand descriptor # 3 (viz. a*b) has been moved to memory location temp [1].



1	(int, 1)	M, addr(a)
2	(int, 1)	M, addr(b)
3	(int, 1)	M, addr(temp [1])



Occupied	#3

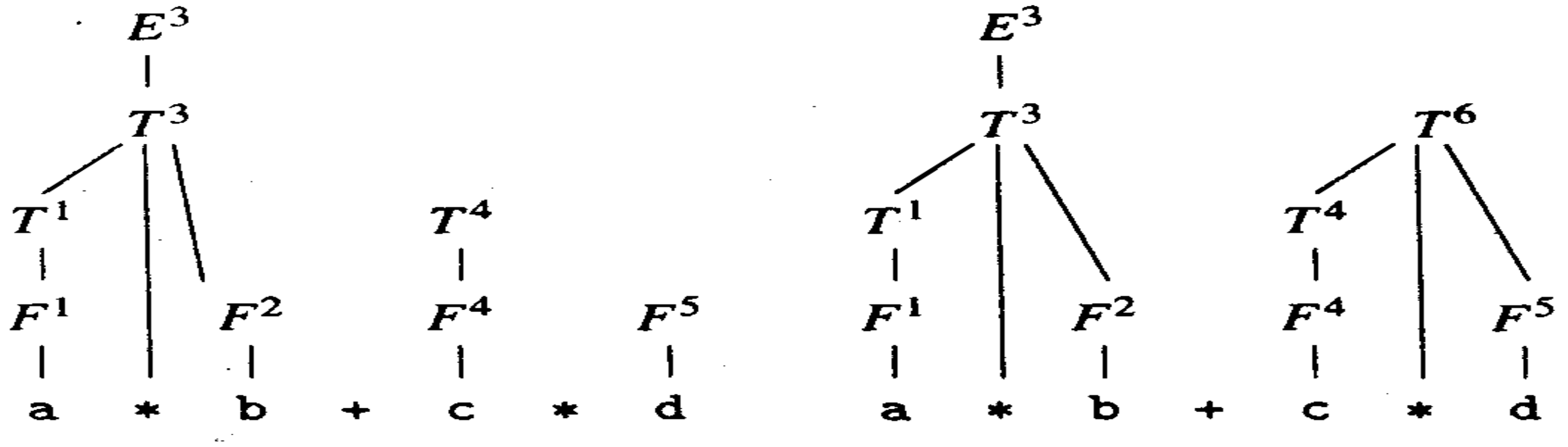
Code generation actions for a*b+c*d

Step no.	*Parsing action*	*Code generation action*
1.	$<id>_a \rightarrow F^1$	Build descriptor # 1
2.	$F^1 \rightarrow T^1$	–
3.	$<id>_b \rightarrow F^2$	Build descriptor # 2
4.	$T^1 * F^2 \rightarrow T^3$	Generate `MOVER AREG, A` `MULT  AREG, B` Build descriptor # 3
5.	$T^3 \rightarrow E^3$	–
6.	$<id>_c \rightarrow F^4$	Build descriptor # 4
7.	$F^4 \rightarrow T^4$	–
8.	$<id>_d \rightarrow F^5$	Build descriptor # 5

9.	$T^4 * F^5 \rightarrow T^6$	Generate `MOVEM AREG, TEMP_1`
		`MOVER AREG, C`
		`MULT  AREG, D`
		Build descriptor # 6

10.	$E^3 + T^6 \rightarrow E^7$	Generate `ADD   AREG, TEMP_1`

Code generation actions



Operand descriptors

(a)

1	(int,1)	M, addr(a)
2	(int,1)	M, addr(b)
3	(int,1)	R, addr(AREG)
4	(int,1)	M, addr(c)
5	(int,1)	M, addr(d)

(b)

1	(int,1)	M, addr(a)
2	(int,1)	M, addr(b)
3	(int,1)	M, addr(temp[1])
4	(int,1)	M, addr(c)
5	(int,1)	M, addr(d)
6	(int,1)	R, addr(AREG)

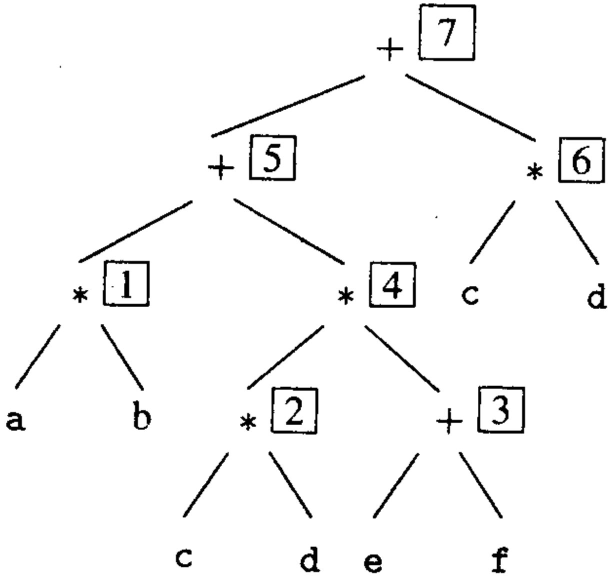
Register descriptor

(a)

Occ.	3

(b)

Occ.	6



Expression tree

```
MOVER   AREG, A
MULT    AREG, B
MOVEM   AREG, TEMP_1
MOVER   AREG, C
MULT    AREG, D
MOVEM   AREG, TEMP_2
MOVER   AREG, E
ADD     AREG, F
MULT    AREG, TEMP_2
ADD     AREG, TEMP_1
MOVEM   AREG, TEMP_3
MOVER   AREG, C
MULT    AREG, D
ADD     AREG, TEMP_3
```

(a)

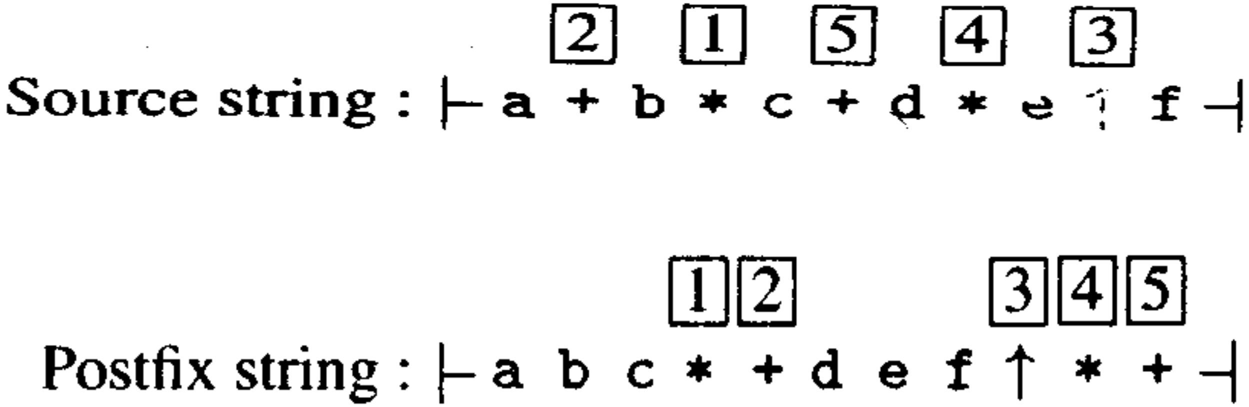
```
MOVER   AREG, A
MULT    AREG, B
MOVEM   AREG, TEMP_1
MOVER   AREG, C
MULT    AREG, D
MOVEM   AREG, TEMP_2
MOVER   AREG, E
ADD     AREG, F
MULT    AREG, TEMP_2
ADD     AREG, TEMP_1
MOVEM   AREG, TEMP_1
MOVER   AREG, C
MULT    AREG, D
ADD     AREG, TEMP_1
```

(b)

Illustration for temporary location usage

Postfix strings

- In the postfix notation, each operator appears immediately after its last operand.
- Thus operators can be evaluated in the order in which they appear in the string.
- Consider the following string, The numbers appearing over the operators indicate their evaluation order.

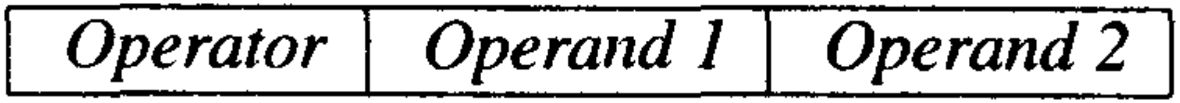


Source string : $\vdash a \overset{\boxed{2}}{+} b \overset{\boxed{1}}{*} c \overset{\boxed{5}}{+} d \overset{\boxed{4}}{*} e \overset{\boxed{3}}{\uparrow} f \dashv$

Postfix string : $\vdash a\ b\ c \overset{\boxed{1}}{*} \overset{\boxed{2}}{+} d\ e\ f \overset{\boxed{3}}{\uparrow} \overset{\boxed{4}}{*} \overset{\boxed{5}}{+} \dashv$

Triples

- A triple is a representation of an elementary operation in the form of a pseudo machine instruction
- Triples are numbered in some convenient manner. Each operand of a triple is either a variable/constant or the result of some evaluation represented by another triple.



Operator	*Operand 1*	*Operand 2*

Triples for string

a b c * + d e f ↑ * +

	operator	*operand 1*	*operand 2*
1	*	b	c
2	+	$\boxed{1}$	a
3	↑	e	f
4	*	d	$\boxed{3}$
5	+	$\boxed{2}$	$\boxed{4}$

indirect triples

- A program representation called indirect triples is useful in optimising compilers.
- In this representation, a table is built to contain all distinct triples in the program.
- A program statement is represented as a list of triple numbers.
- This arrangement is useful to detect the occurrences of identical expressions in a program.
- For efficiency reasons, a hash organisation can be used tor the table of triples.
- The indirect triples representation provides memory economy.
- It also aids in certain forms of optimisation, viz, common sub expression elimination.

indirect triples

```
z := a+b*c+d*e↑f;
y := x+b*c;
```

	operator	*operand 1*	*operand 2*
1	*	b	c
2	+	$\boxed{1}$	a
3	↑	e	f
4	*	d	$\boxed{3}$
5	+	$\boxed{2}$	$\boxed{4}$
6	+	x	$\boxed{1}$

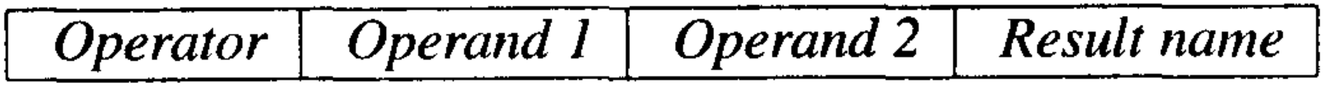
triples' table

stmt no.	*triple nos.*
1	1,2,3,4,5
2	1,6

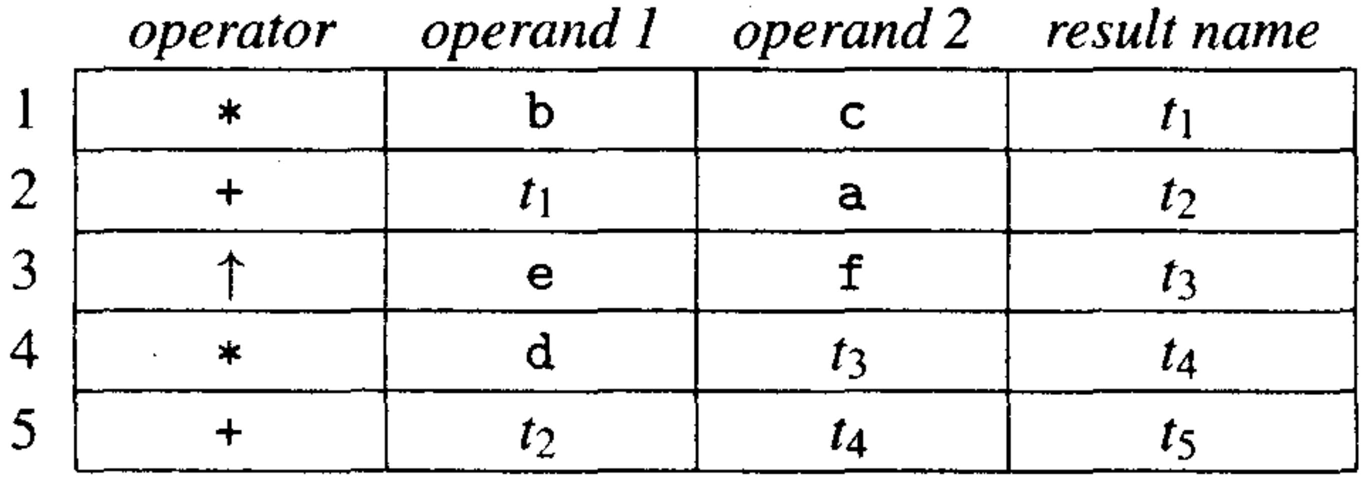
statement table

Quadruples

- A quadruple represents an elementary evaluation in the following format:
- Here, result name designates the result of the evaluation. It can be used as the operand of another quadruple. This is more convenient than using a number (as in the case of triples) to designate a sub expression.



Operator	*Operand 1*	*Operand 2*	*Result name*

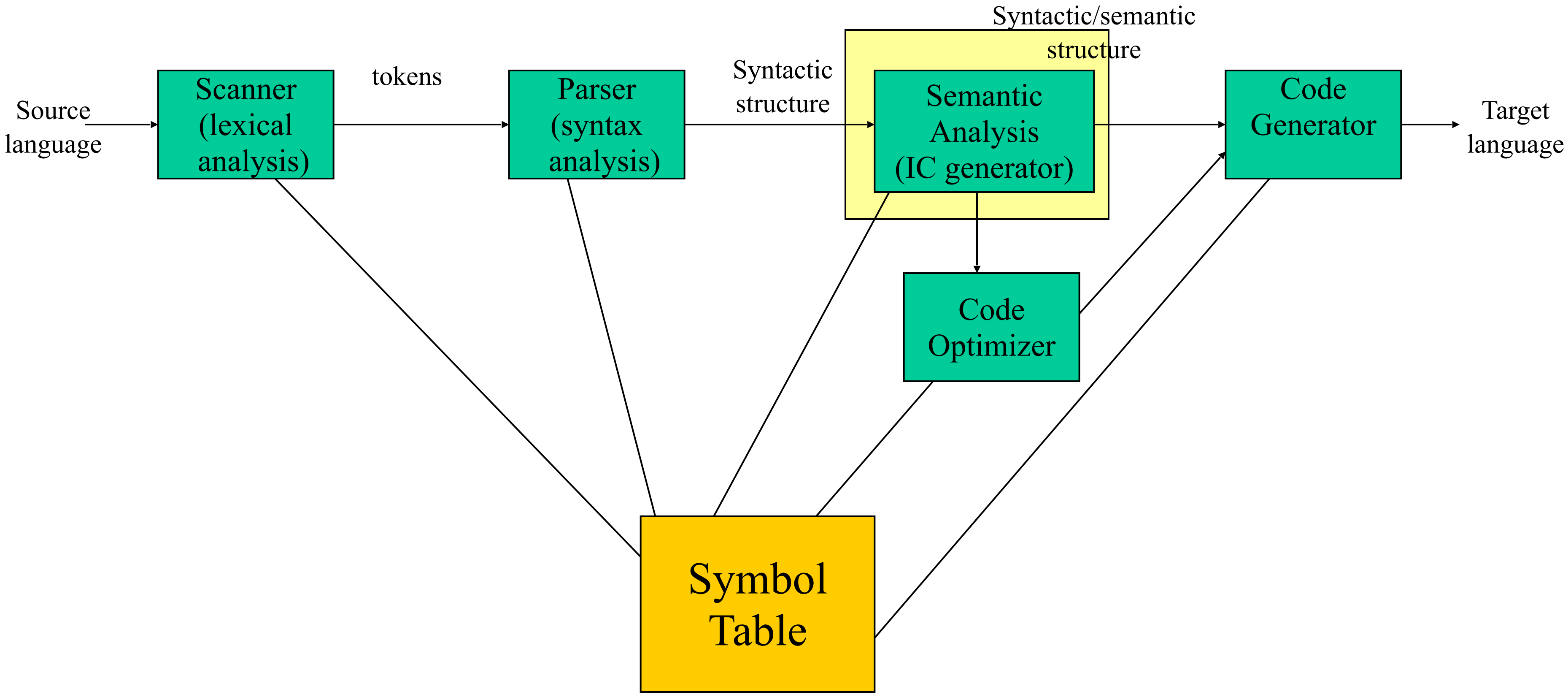


	operator	*operand 1*	*operand 2*	*result name*
1	*	b	c	t_1
2	+	t_1	a	t_2
3	↑	e	f	t_3
4	*	d	t_3	t_4
5	+	t_2	t_4	t_5

Quadruples for the expression string

CODE OPTIMISATION

Code Optimisation



Code Optimisation

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

Register Allocation

- 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, y
ADD  R0, z
MOV  R0, x
```

Peephole optimisation

- Peephole optimisation is an optimisation technique performed on a small set of compiler-generated instructions, the small set is known as the peephole or window.
- Peephole optimisation involves changing the small set of instructions to an equivalent set that has better performance.

Peephole Optimisations

- Constant Folding

```
x := 32                  becomes    x := 64
x := x + 32
```

- Unreachable Code

```
goto L2
x := x + 1               unneeded
```

- Flow of control optimisations

```
goto L1                  becomes    goto L2
 …
L1: goto L2
```

Peephole Optimisations

- 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
```

Loop optimisation

- 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 loop invariant computations.
- Move them outside the loop – meaning should not be changed.
- Perform Control Flow Analysis - To detect loops in the program
- 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);
}
```

Three-address code representations

```
(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

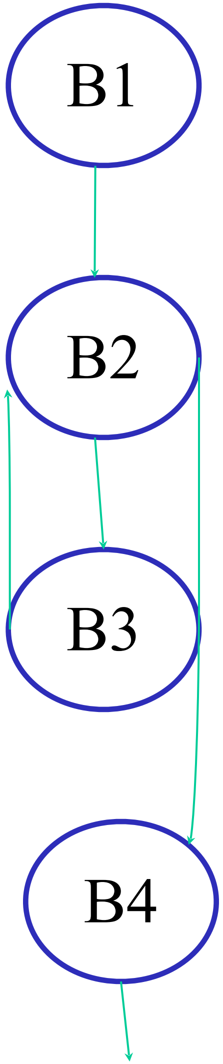
```
If i<=x goto(8)
```

- Block B3

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

- Block B4

```
goto calling program
```



Optimisation

Apply below mentioned optimisation techniques at basic block level to every blocks

- Common Subexpression elimination
- Constant Propagation
- Copy propagation
- Dead code elimination
- Redundancy elimination
- Code Motion
- Strength reduction
- Loop unrolling … etc.

Common Subexpression elimination

Common Subexpression Elimination is an optimisation that searches for instances of identical expressions, and replaces them with a single variable holding the computed value.

- 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

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

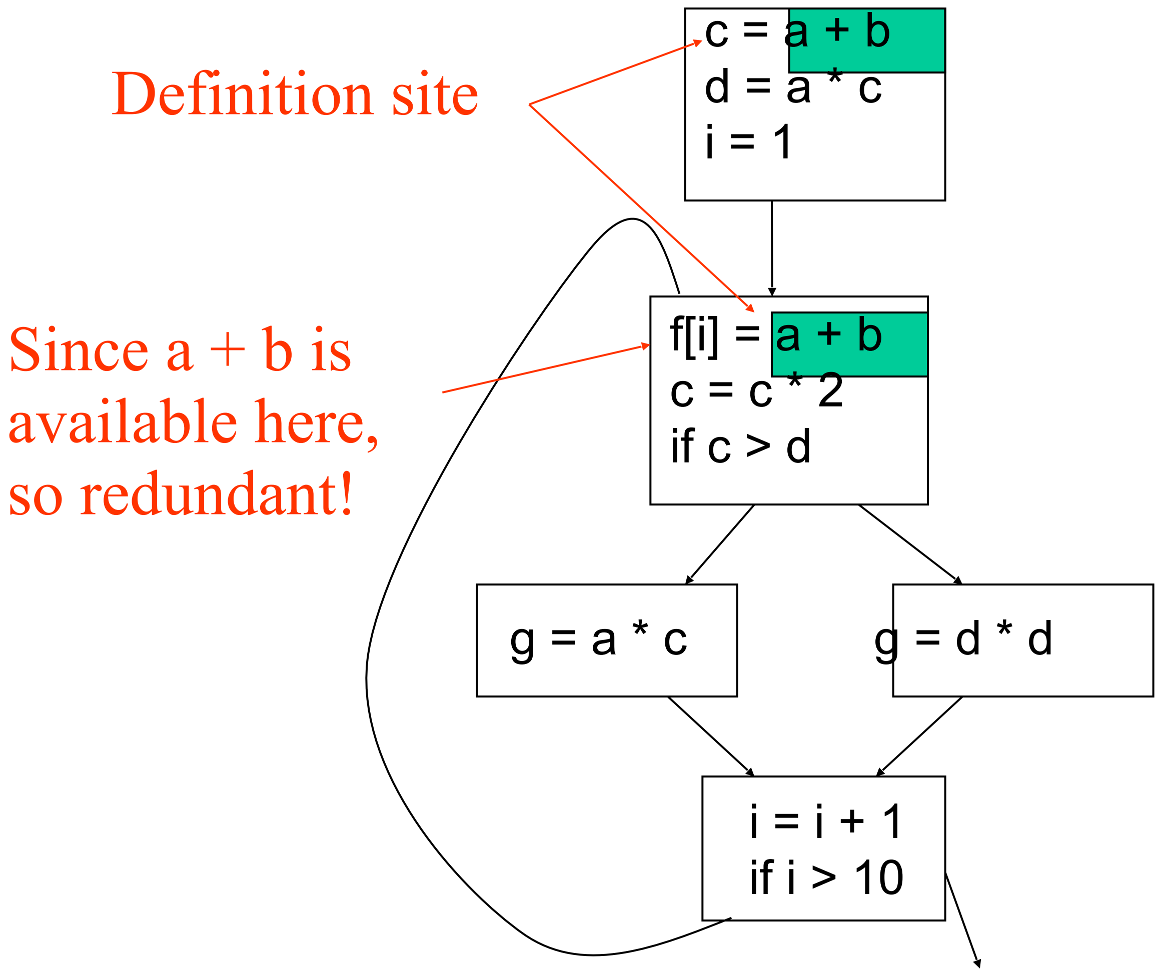
Redundant Expressions

Eliminates expressions that are redundant on some but not necessarily all paths through a program. PRE is a form of common subexpression elimination.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

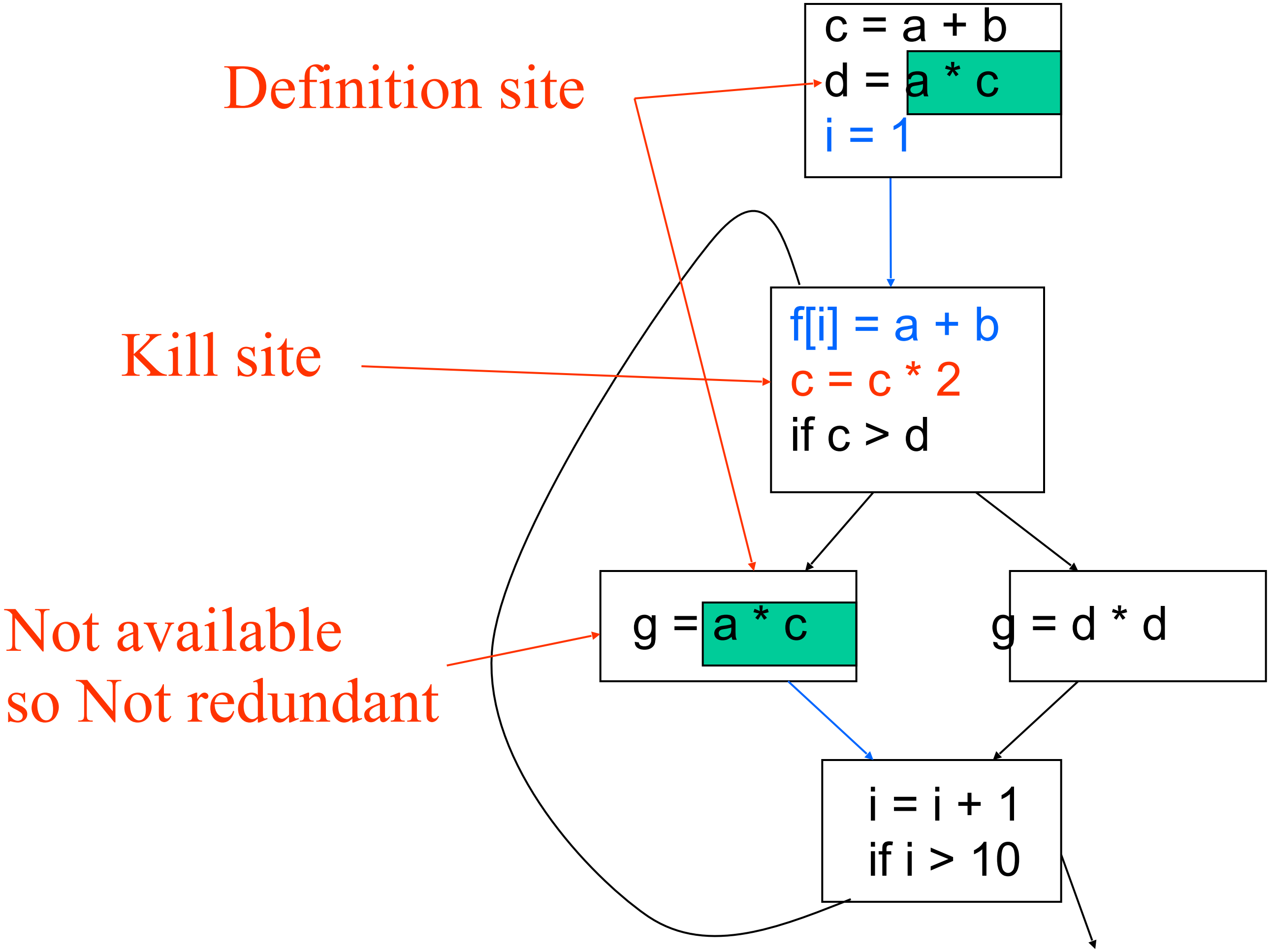
Redundant Expressions



Candidates:

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

Redundant Expressions



Definition site

Kill site

Not available
so Not redundant

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

f[i] = a + b
c = c * 2
if c > d

g = a * c

g = d * d

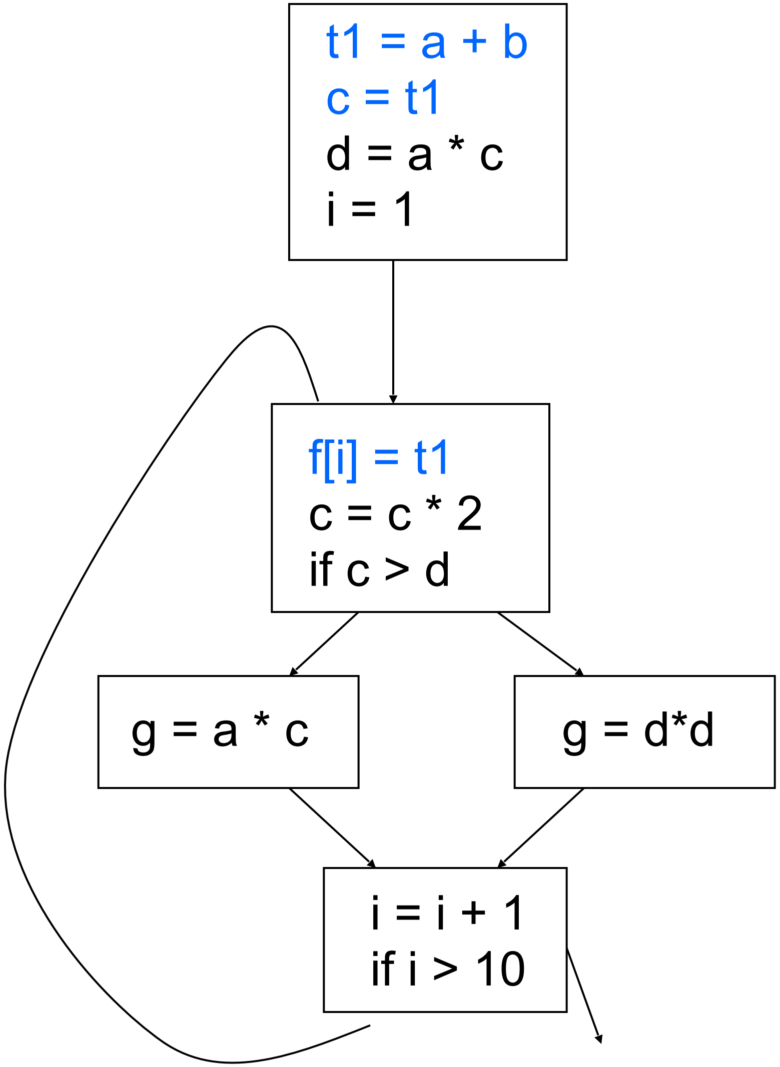
i = i + 1
if i > 10

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



t1 = a + b
c = t1
d = a * c
i = 1

f[i] = t1
c = c * 2
if c > d

g = a * c

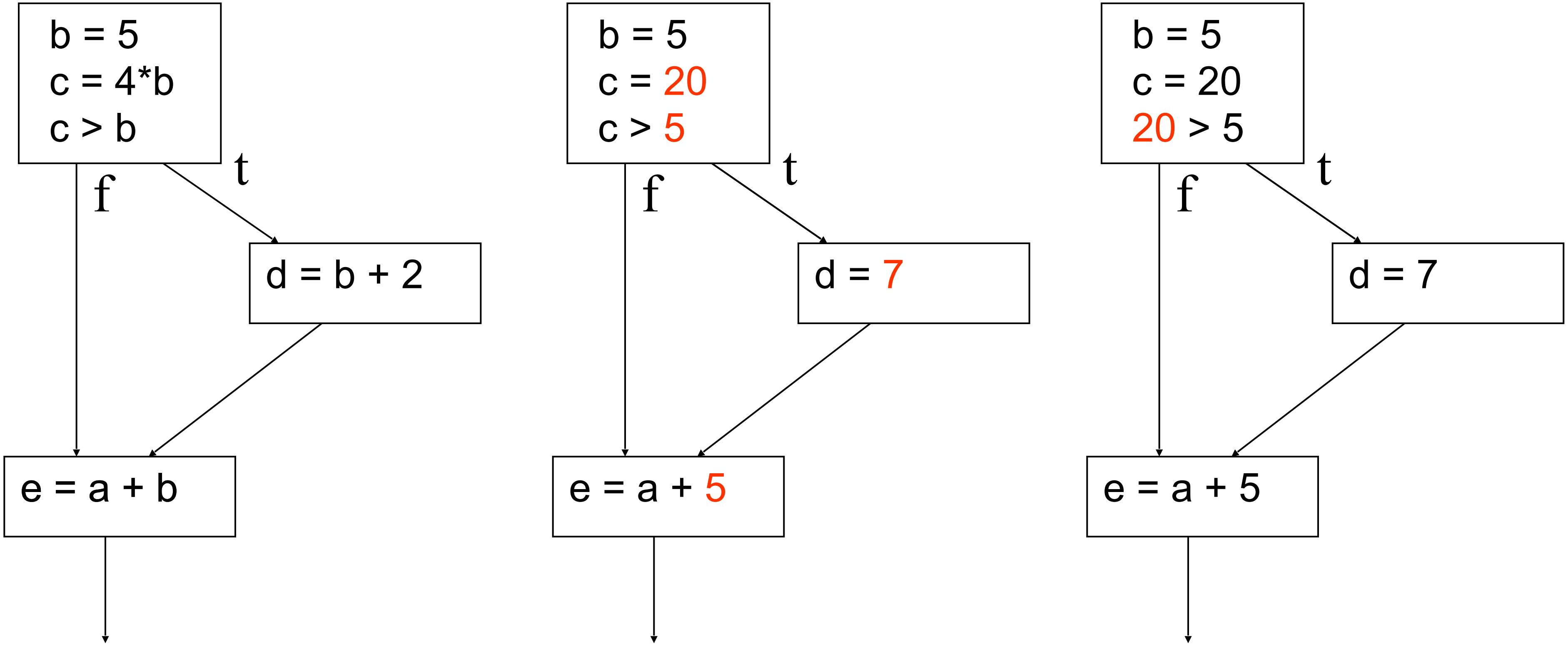
g = d*d

i = i + 1
if i > 10

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

Constant Propagation

Constant propagation is the process of substituting the values of known constants in expressions. Constant propagation eliminates cases in which values are copied from one location or variable to another, in order to simply assign their value to another variable.



b = 5
c = 4*b
c > b

d = b + 2

e = a + b

b = 5
c = 20
c > 5

d = 7

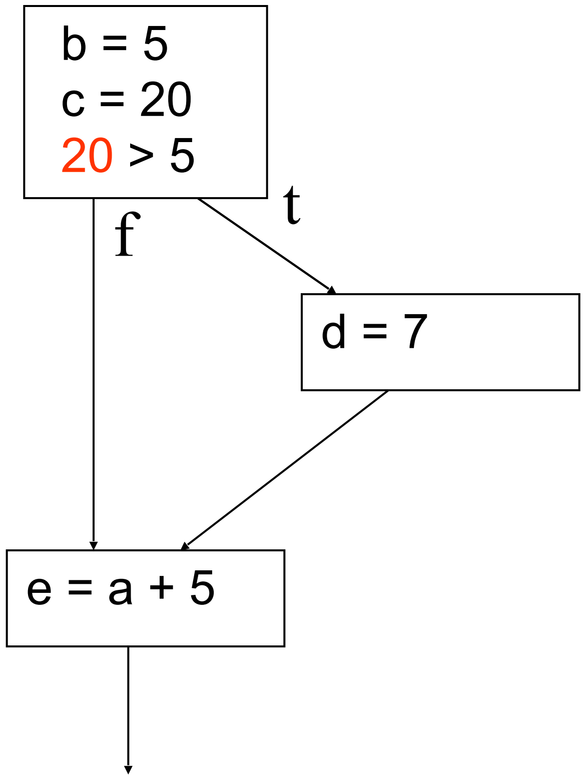
e = a + 5

b = 5
c = 20
20 > 5

d = 7

e = a + 5

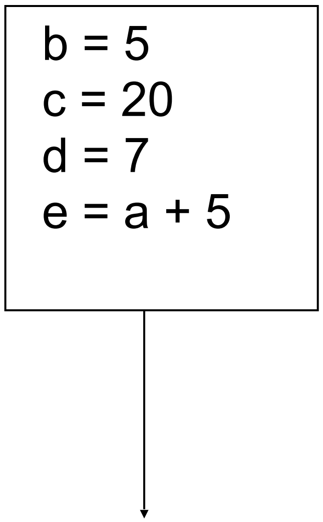
Constant Propagation



b = 5
c = 20
20 > 5

d = 7

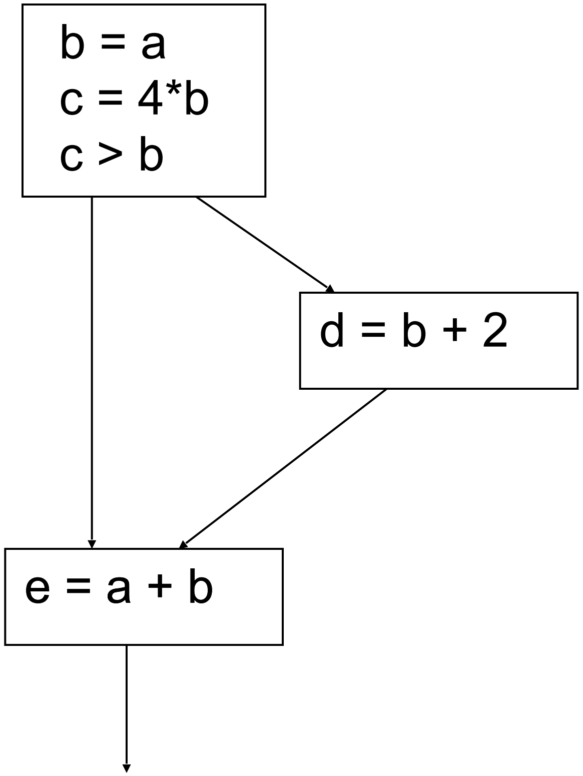
e = a + 5



b = 5
c = 20
d = 7
e = a + 5

Copy Propagation

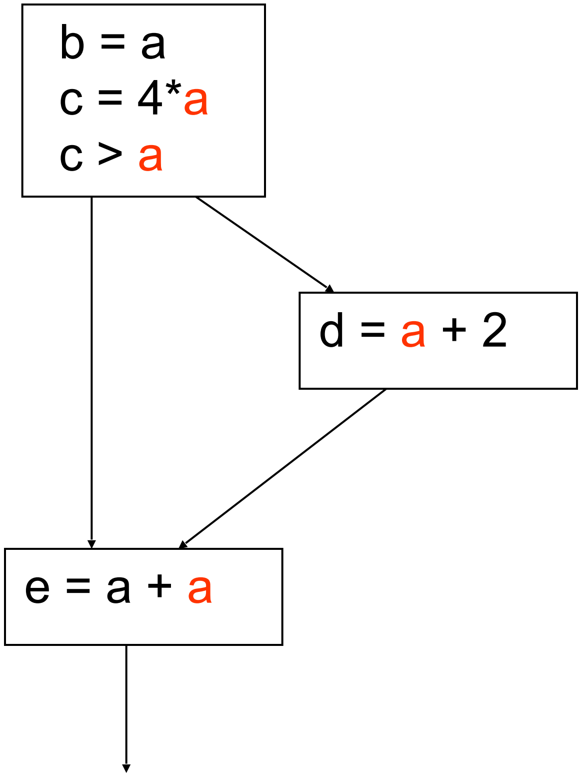
copy propagation is the process of replacing the occurrences of targets of direct assignments with their values. A direct assignment is an instruction of the form x = y , which simply assigns the value of y to x



b = a
c = 4*b
c > b

d = b + 2

e = a + b



b = a
c = 4*a
c > a

d = a + 2

e = a + a

Code Motion

Code is moved out of the loop as it won't have any difference if it is performed inside the loop repeatedly or outside the loop once. The compiler is taking the code that doesn't need to be in the loop and moving it outside of it for optimisation purposes.

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

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

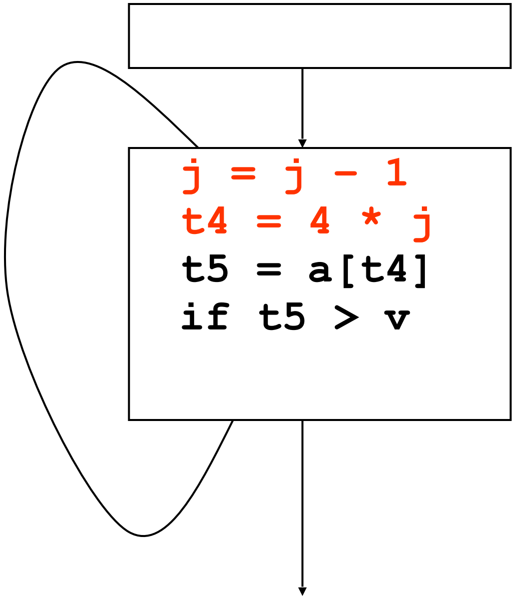
```
t := limit - 2
    while (i <= t)
```

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

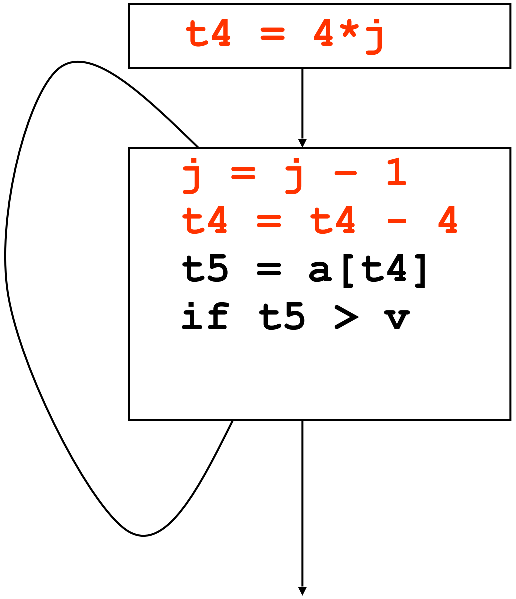
Strength Reduction

strength reduction is a compiler optimisation where expensive operations are replaced with equivalent but less expensive operations.

- Induction Variables control loop iterations



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



```
t4 = 4*j
```

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

induction variable is a variable that gets increased or decreased by a fixed amount on every iteration of a loop or is a linear function of another induction variable.

Loop Unrolling

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

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

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

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

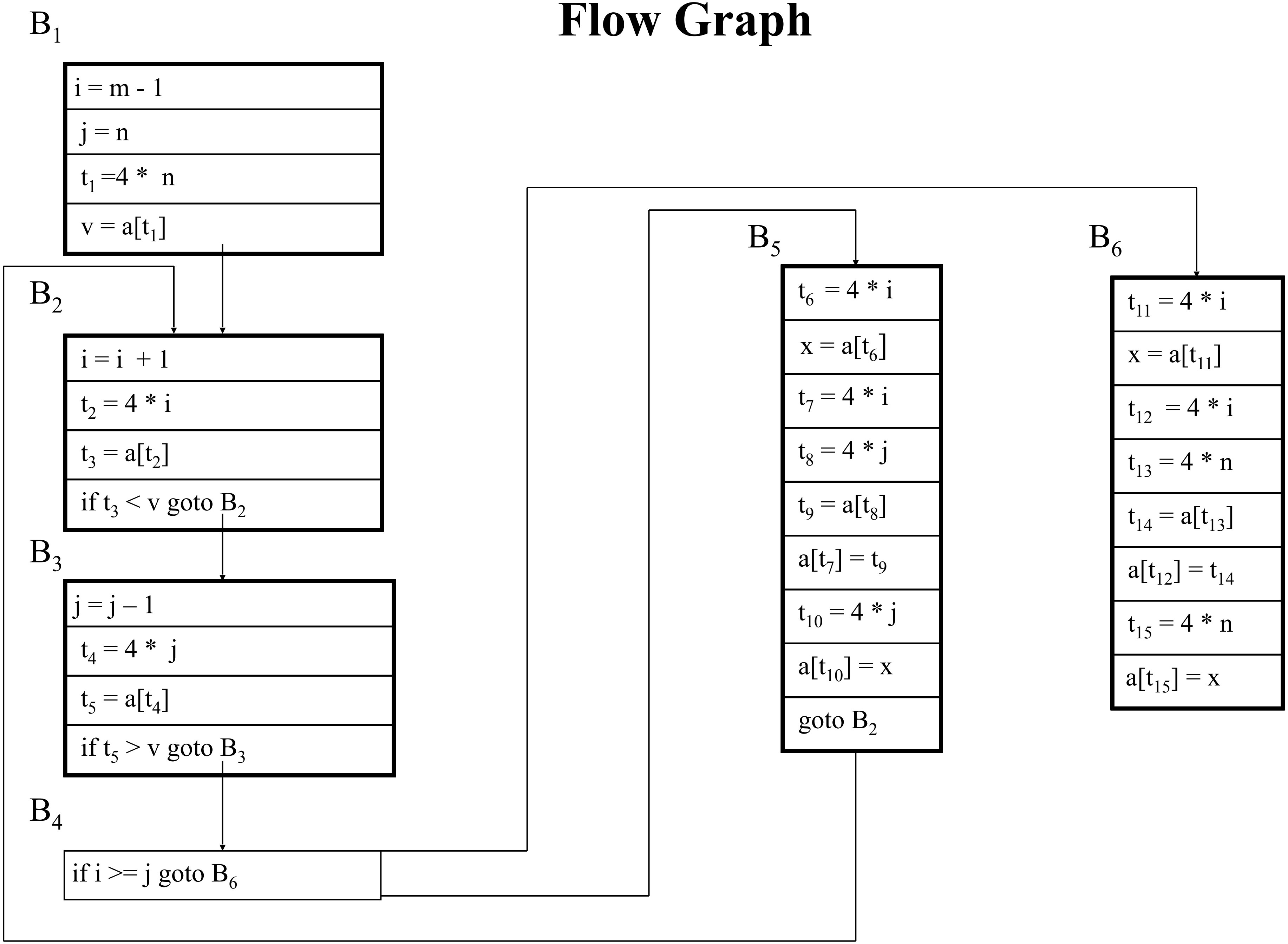
1	i = m - 1	16	t_7 = 4 * I
2	j = n	17	t_8 = 4 * j
3	t_1 =4 * n	18	t_9 = a[t_8]
4	v = a[t_1]	19	a[t_7] = t_9
5	i = i + 1	20	t_{10} = 4 * j
6	t_2 = 4 * i	21	a[t_{10}] = x
7	t_3 = a[t_2]	22	goto (5)
8	if t_3 < v goto (5)	23	t_{11} = 4 * I
9	j = j – 1	24	x = a[t_{11}]
10	t_4 = 4 * j	25	t_{12} = 4 * i
11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

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



B_1

- $i = m - 1$
- $j = n$
- $t_1 = 4 * n$
- $v = a[t_1]$

B_2

- $i = i + 1$
- $t_2 = 4 * i$
- $t_3 = a[t_2]$
- if $t_3 < v$ goto B_2

B_3

- $j = j - 1$
- $t_4 = 4 * j$
- $t_5 = a[t_4]$
- if $t_5 > v$ goto B_3

B_4

- if $i >= j$ goto B_6

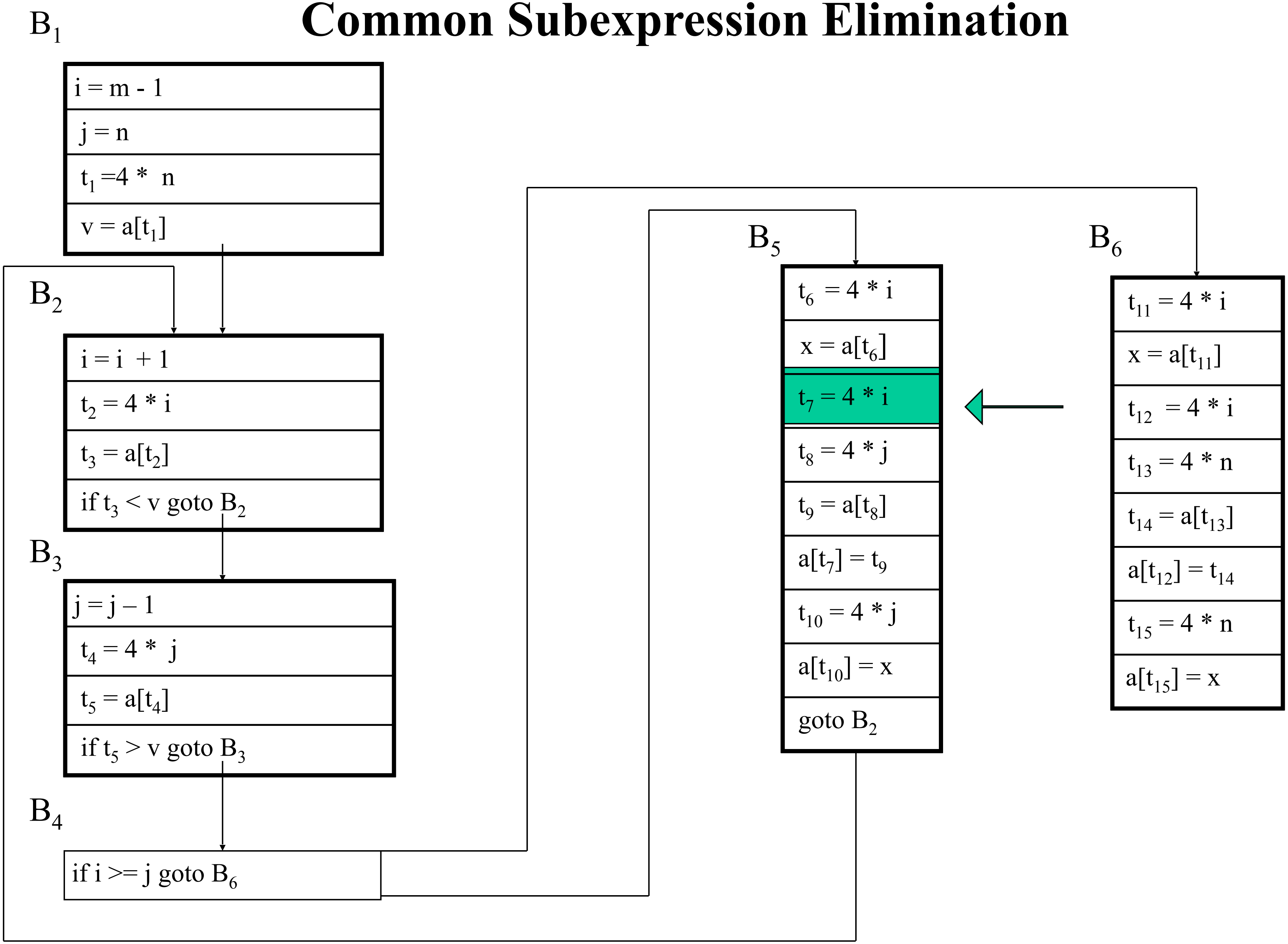
B_5

- $t_6 = 4 * i$
- $x = a[t_6]$
- $t_7 = 4 * i$
- $t_8 = 4 * j$
- $t_9 = a[t_8]$
- $a[t_7] = t_9$
- $t_{10} = 4 * j$
- $a[t_{10}] = x$
- goto B_2

B_6

- $t_{11} = 4 * i$
- $x = a[t_{11}]$
- $t_{12} = 4 * i$
- $t_{13} = 4 * n$
- $t_{14} = a[t_{13}]$
- $a[t_{12}] = t_{14}$
- $t_{15} = 4 * n$
- $a[t_{15}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$

B_2

$i = i + 1$
$t_2 = 4 * i$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2

B_3

$j = j – 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

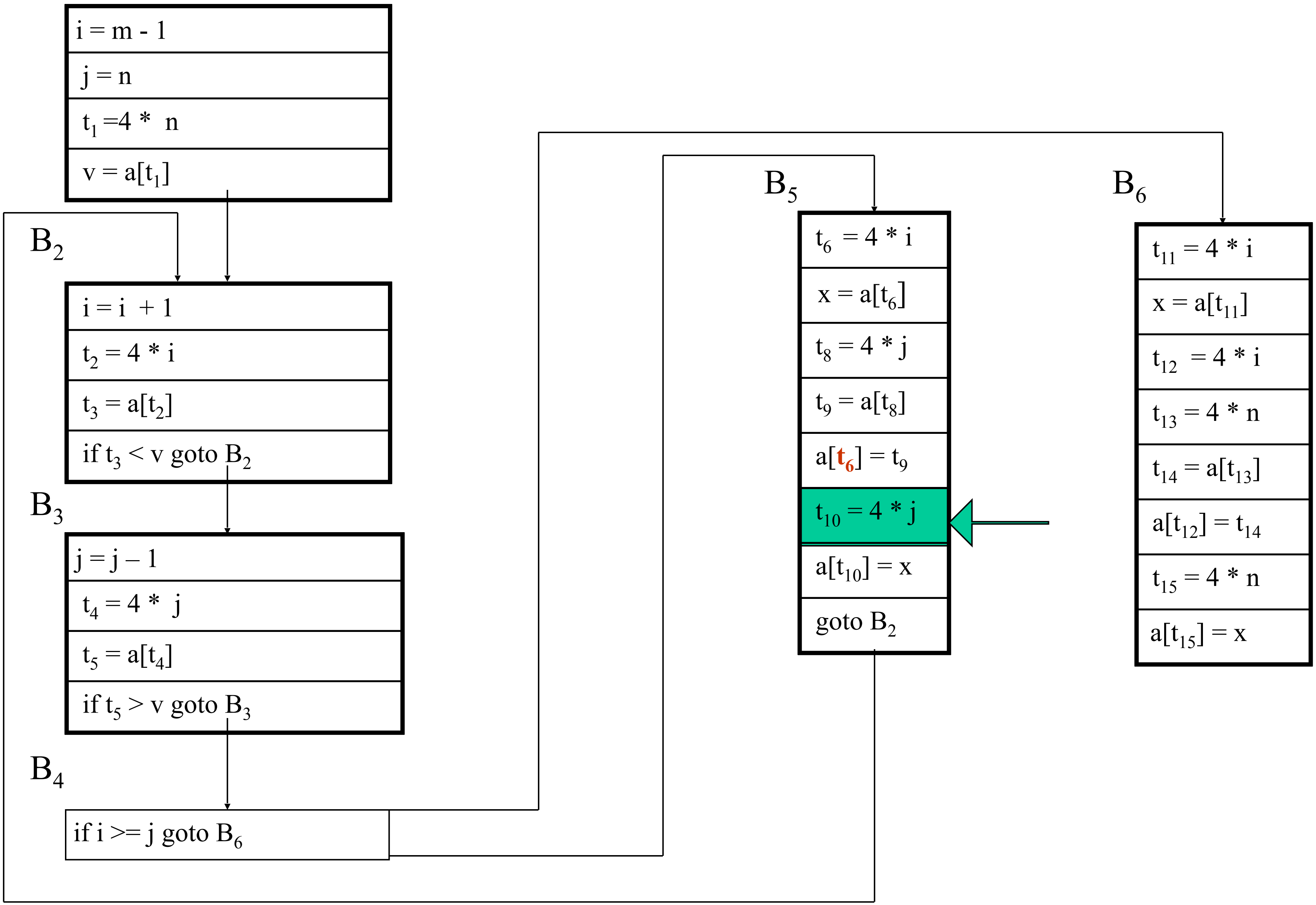
B_5

$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * i$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

B_6

$t_{11} = 4 * i$
$x = a[t_{11}]$
$t_{12} = 4 * i$
$t_{13} = 4 * n$
$t_{14} = a[t_{13}]$
$a[t_{12}] = t_{14}$
$t_{15} = 4 * n$
$a[t_{15}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$

B_2

$i = i + 1$
$t_2 = 4 * i$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2

B_3

$j = j – 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

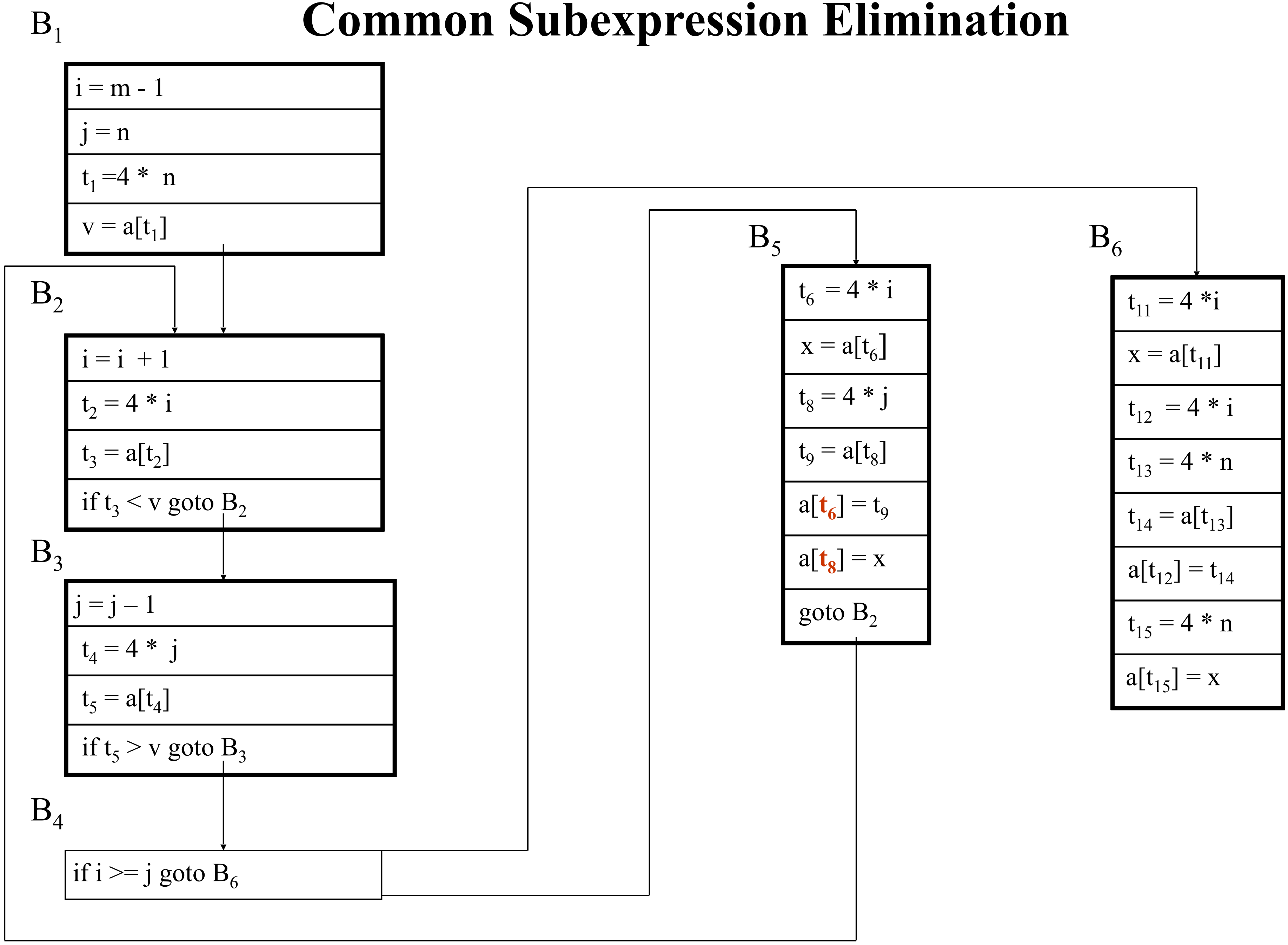
B_5

$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

B_6

$t_{11} = 4 * i$
$x = a[t_{11}]$
$t_{12} = 4 * i$
$t_{13} = 4 * n$
$t_{14} = a[t_{13}]$
$a[t_{12}] = t_{14}$
$t_{15} = 4 * n$
$a[t_{15}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j – 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$t_6 = 4 * i$

$x = a[t_6]$

$t_8 = 4 * j$

$t_9 = a[t_8]$

$a[t_6] = t_9$

$a[t_8] = x$

goto B_2

B_6

$t_{11} = 4 * i$

$x = a[t_{11}]$

$t_{12} = 4 * i$

$t_{13} = 4 * n$

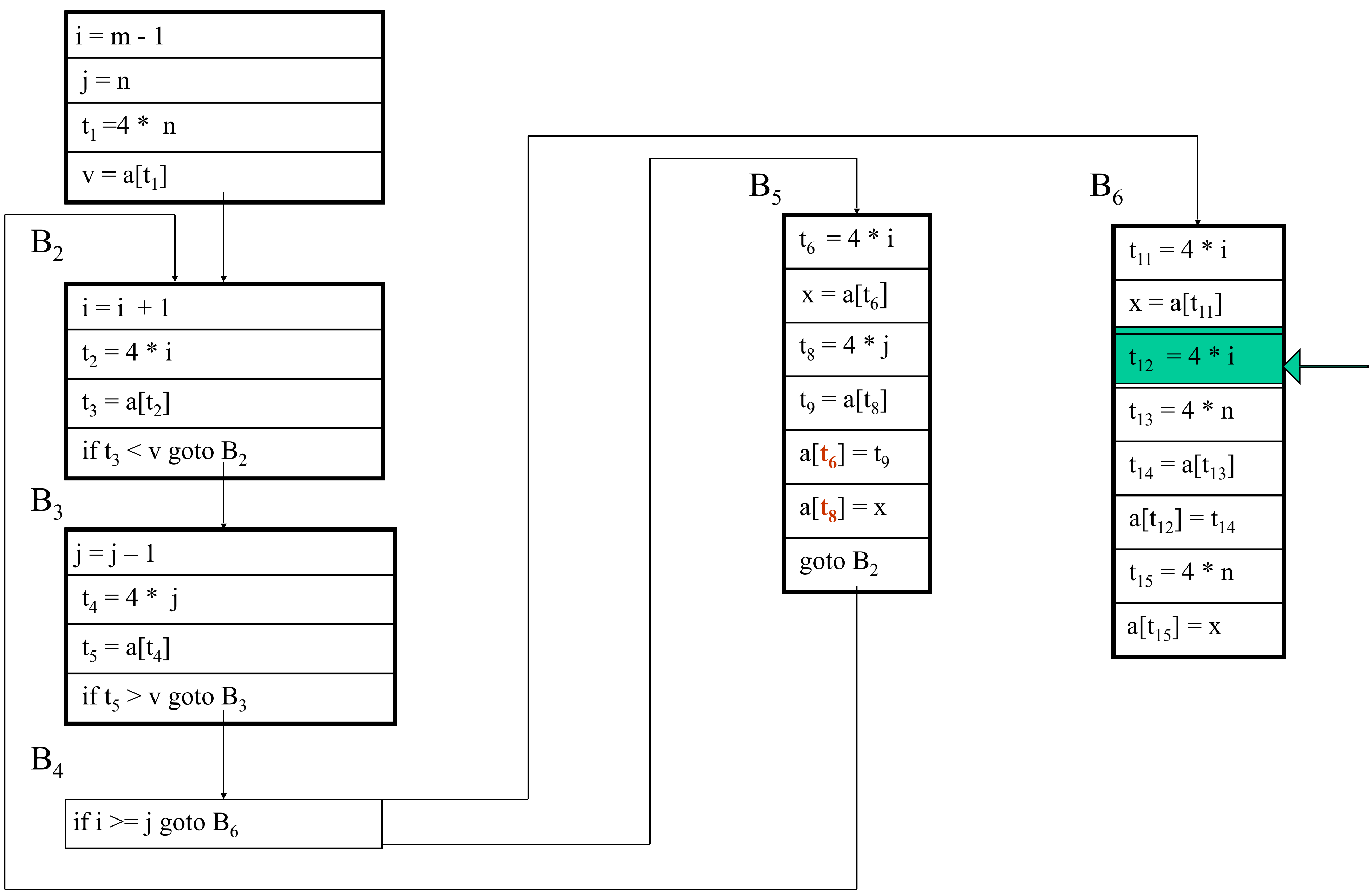
$t_{14} = a[t_{13}]$

$a[t_{12}] = t_{14}$

$t_{15} = 4 * n$

$a[t_{15}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j – 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$t_6 = 4 * i$

$x = a[t_6]$

$t_8 = 4 * j$

$t_9 = a[t_8]$

$a[t_6] = t_9$

$a[t_8] = x$

goto B_2

B_6

$t_{11} = 4 * i$

$x = a[t_{11}]$

$t_{12} = 4 * i$

$t_{13} = 4 * n$

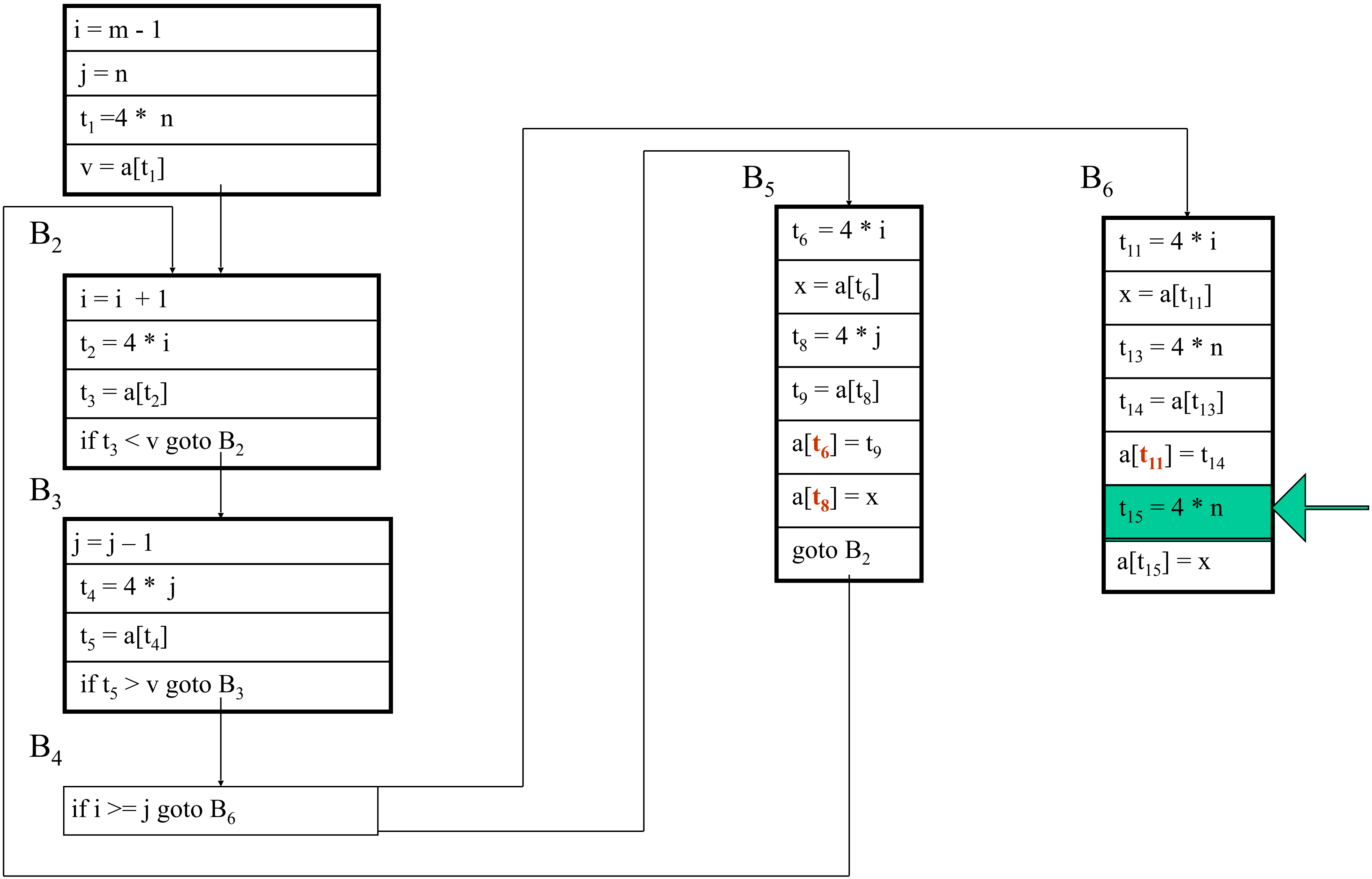
$t_{14} = a[t_{13}]$

$a[t_{12}] = t_{14}$

$t_{15} = 4 * n$

$a[t_{15}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j - 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$t_6 = 4 * i$

$x = a[t_6]$

$t_8 = 4 * j$

$t_9 = a[t_8]$

$a[t_6] = t_9$

$a[t_8] = x$

goto B_2

B_6

$t_{11} = 4 * i$

$x = a[t_{11}]$

$t_{13} = 4 * n$

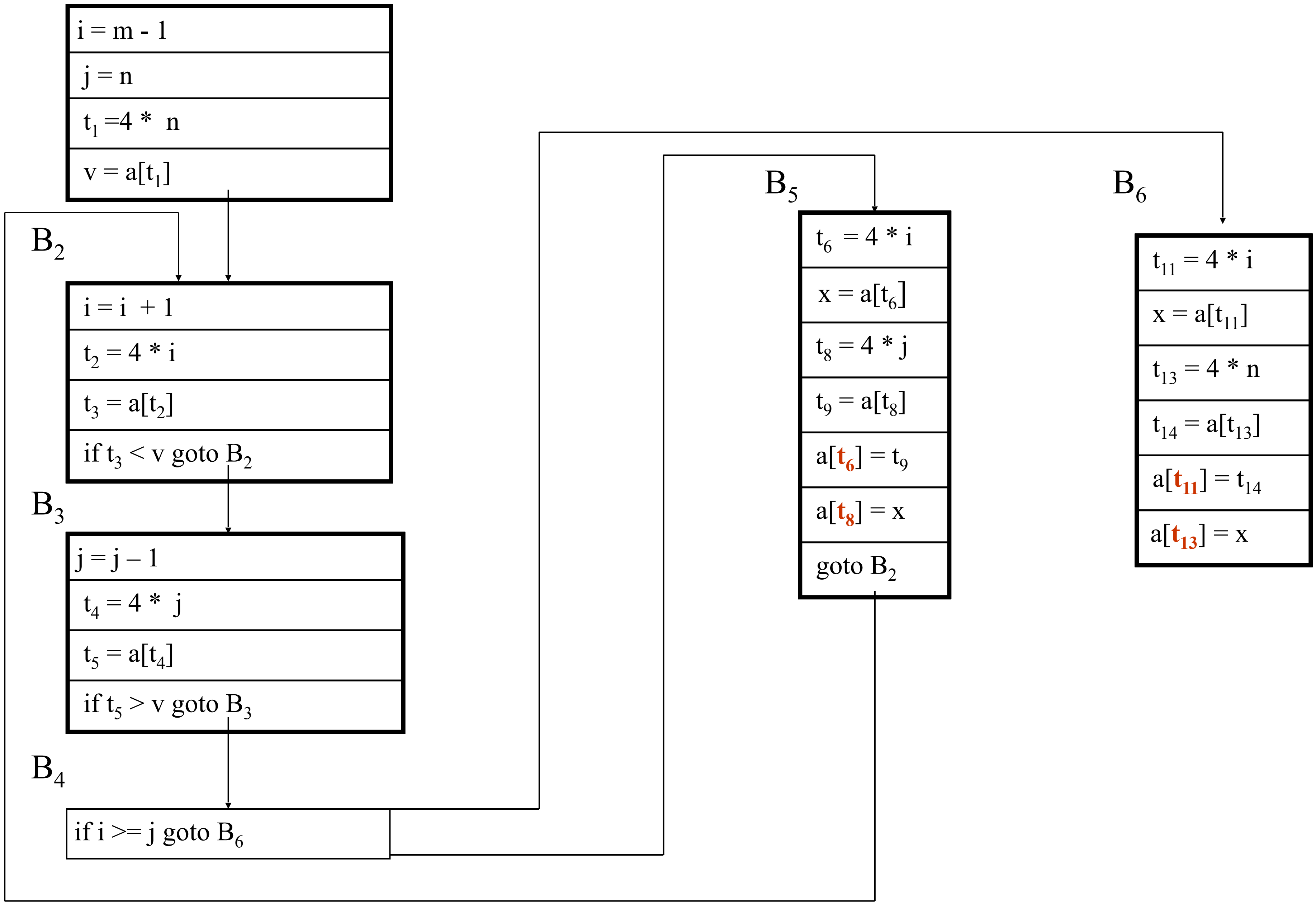
$t_{14} = a[t_{13}]$

$a[t_{11}] = t_{14}$

$t_{15} = 4 * n$

$a[t_{15}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j - 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$t_6 = 4 * i$

$x = a[t_6]$

$t_8 = 4 * j$

$t_9 = a[t_8]$

$a[t_6] = t_9$

$a[t_8] = x$

goto B_2

B_6

$t_{11} = 4 * i$

$x = a[t_{11}]$

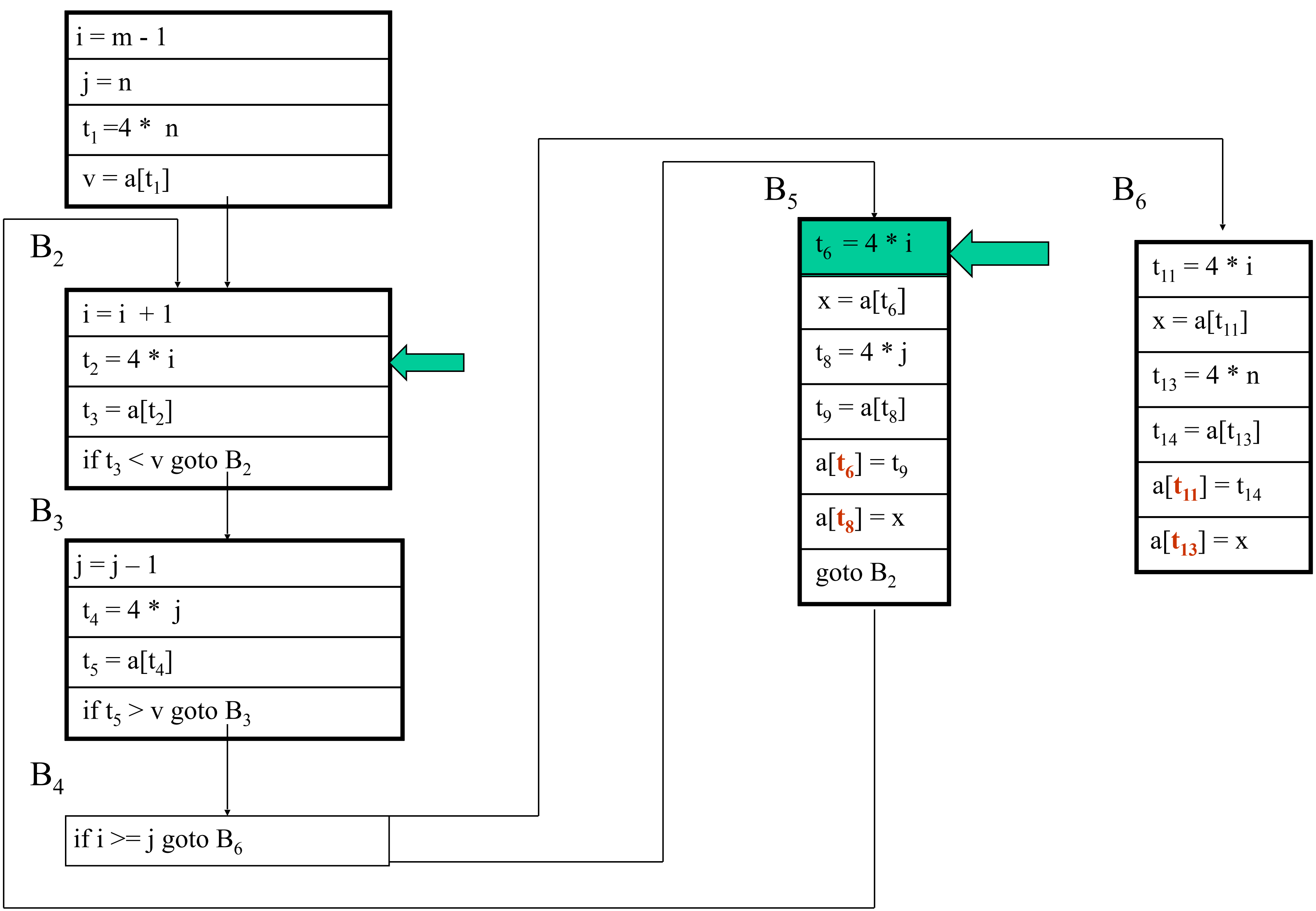
$t_{13} = 4 * n$

$t_{14} = a[t_{13}]$

$a[t_{11}] = t_{14}$

$a[t_{13}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j - 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$t_6 = 4 * i$

$x = a[t_6]$

$t_8 = 4 * j$

$t_9 = a[t_8]$

$a[t_6] = t_9$

$a[t_8] = x$

goto B_2

B_6

$t_{11} = 4 * i$

$x = a[t_{11}]$

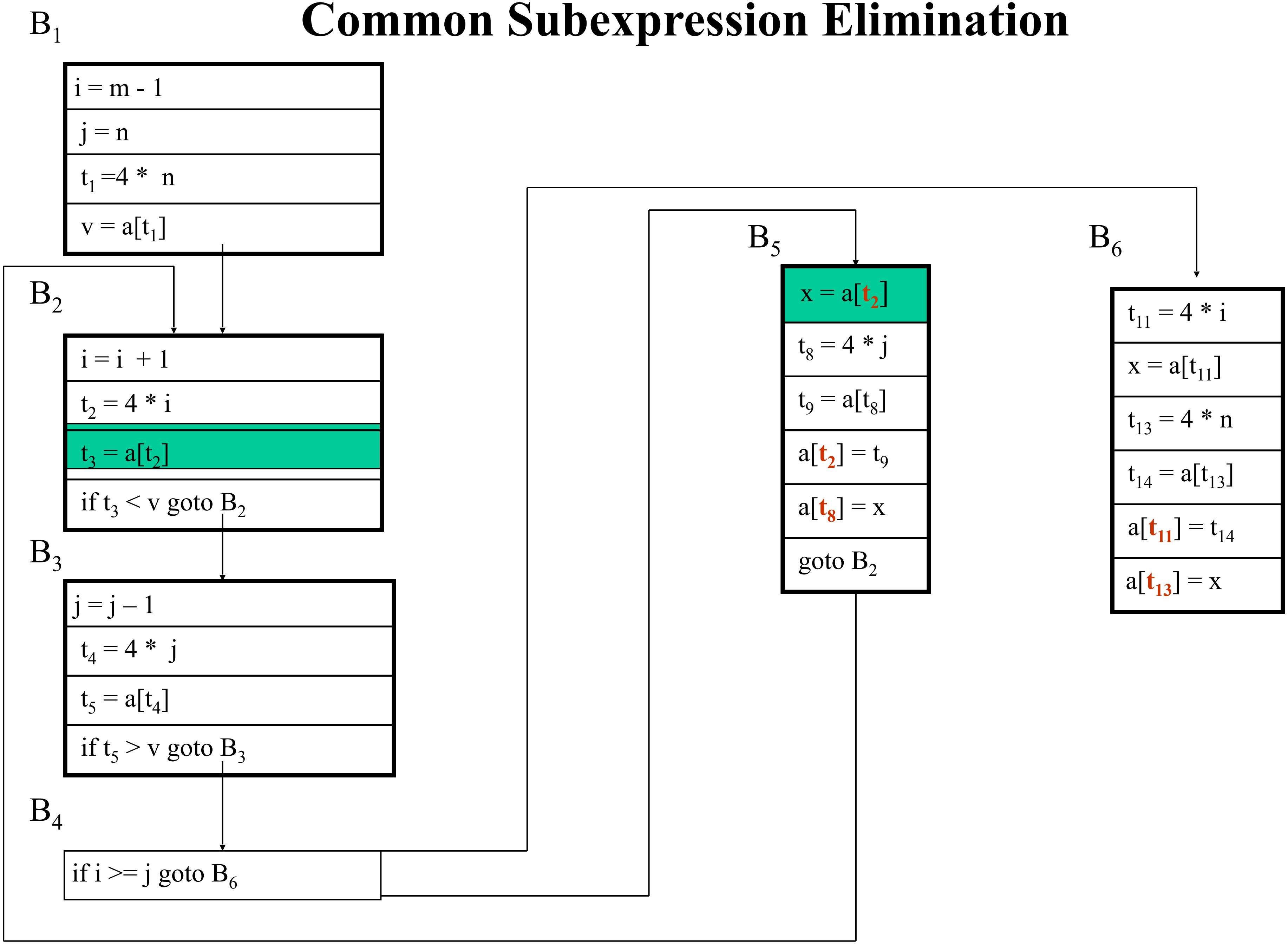
$t_{13} = 4 * n$

$t_{14} = a[t_{13}]$

$a[t_{11}] = t_{14}$

$a[t_{13}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j - 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$x = a[t_2]$

$t_8 = 4 * j$

$t_9 = a[t_8]$

$a[t_2] = t_9$

$a[t_8] = x$

goto B_2

B_6

$t_{11} = 4 * i$

$x = a[t_{11}]$

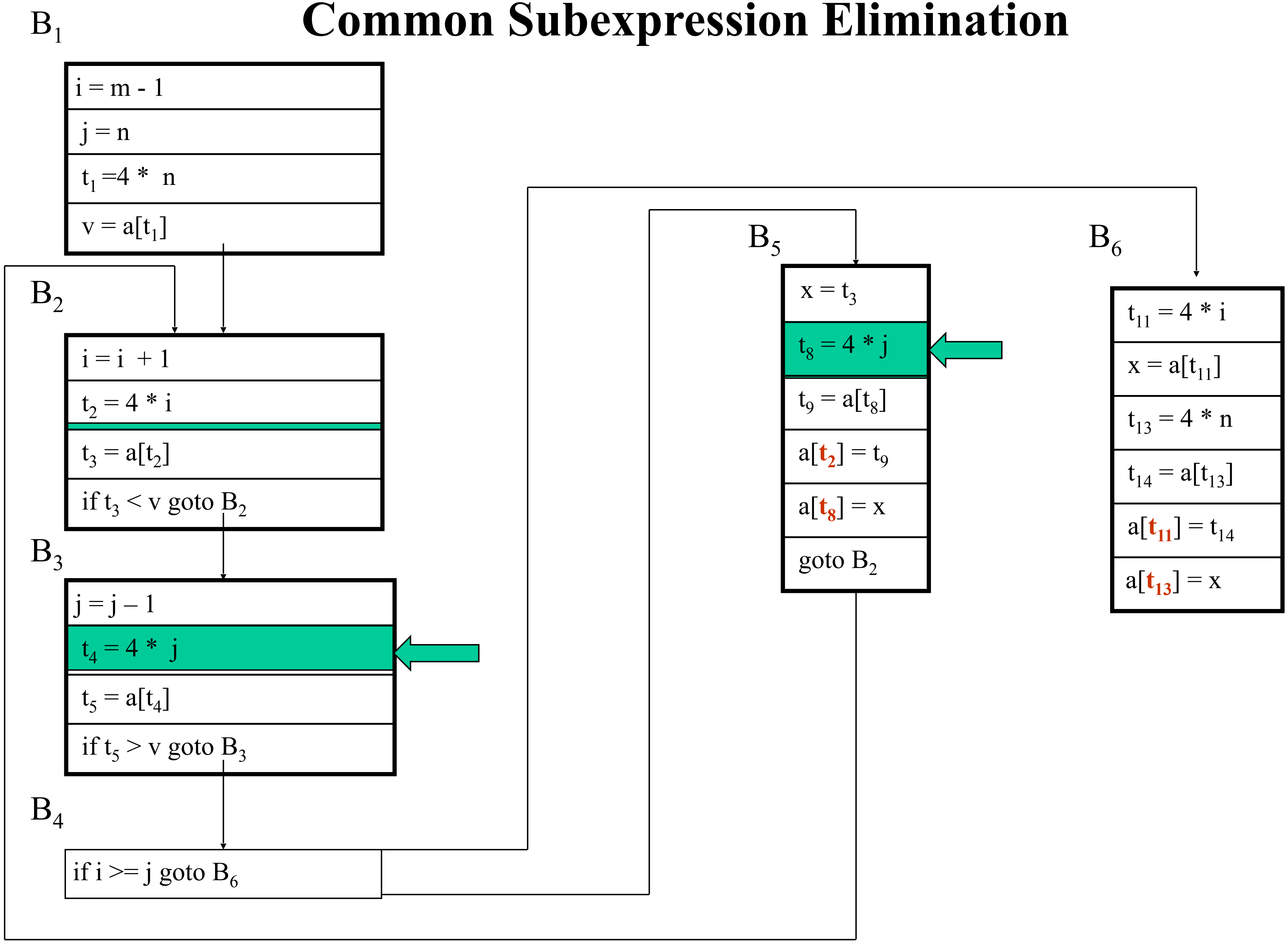
$t_{13} = 4 * n$

$t_{14} = a[t_{13}]$

$a[t_{11}] = t_{14}$

$a[t_{13}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j - 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$x = t_3$

$t_8 = 4 * j$

$t_9 = a[t_8]$

$a[t_2] = t_9$

$a[t_8] = x$

goto B_2

B_6

$t_{11} = 4 * i$

$x = a[t_{11}]$

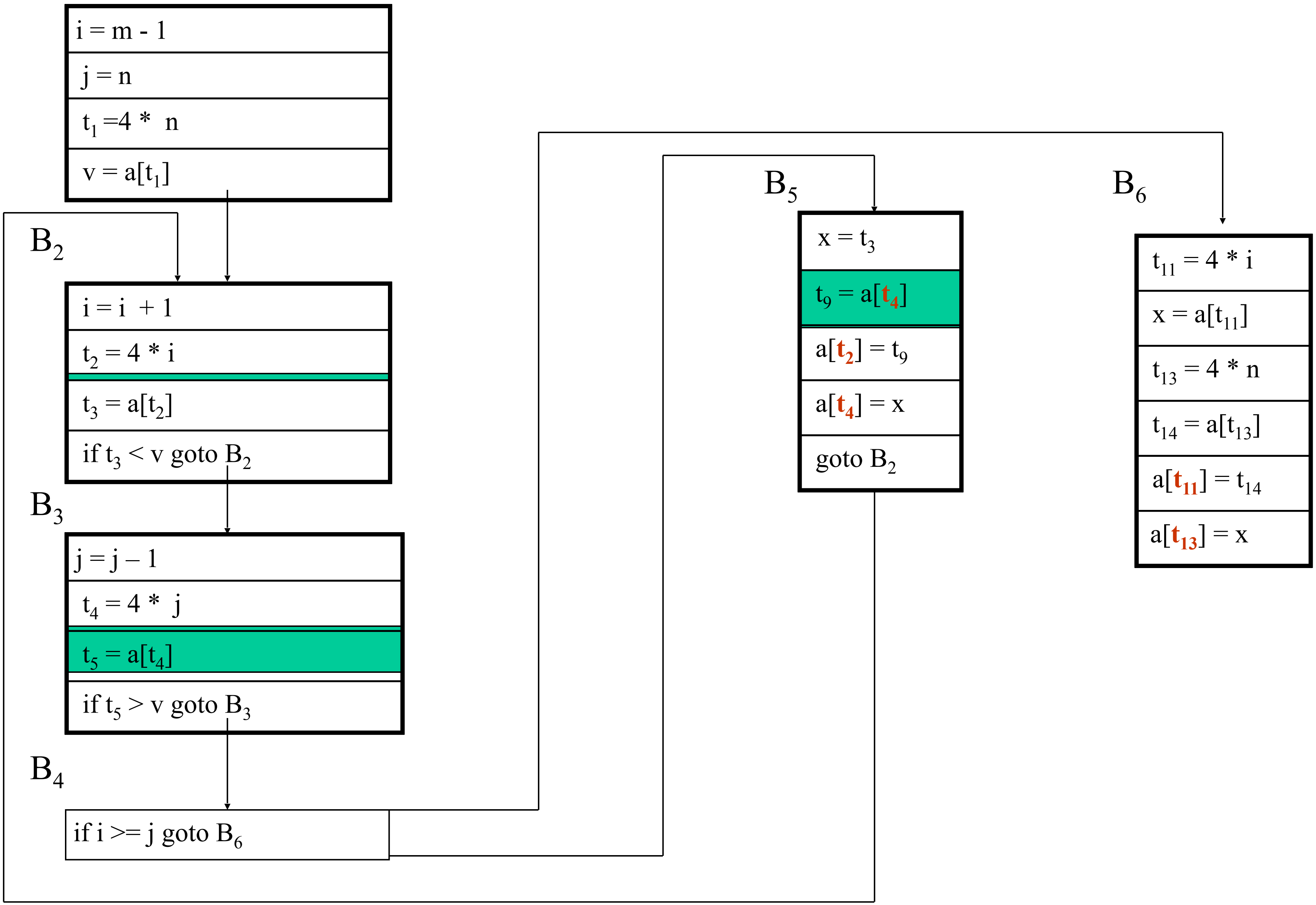
$t_{13} = 4 * n$

$t_{14} = a[t_{13}]$

$a[t_{11}] = t_{14}$

$a[t_{13}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j - 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$x = t_3$

$t_9 = a[t_4]$

$a[t_2] = t_9$

$a[t_4] = x$

goto B_2

B_6

$t_{11} = 4 * i$

$x = a[t_{11}]$

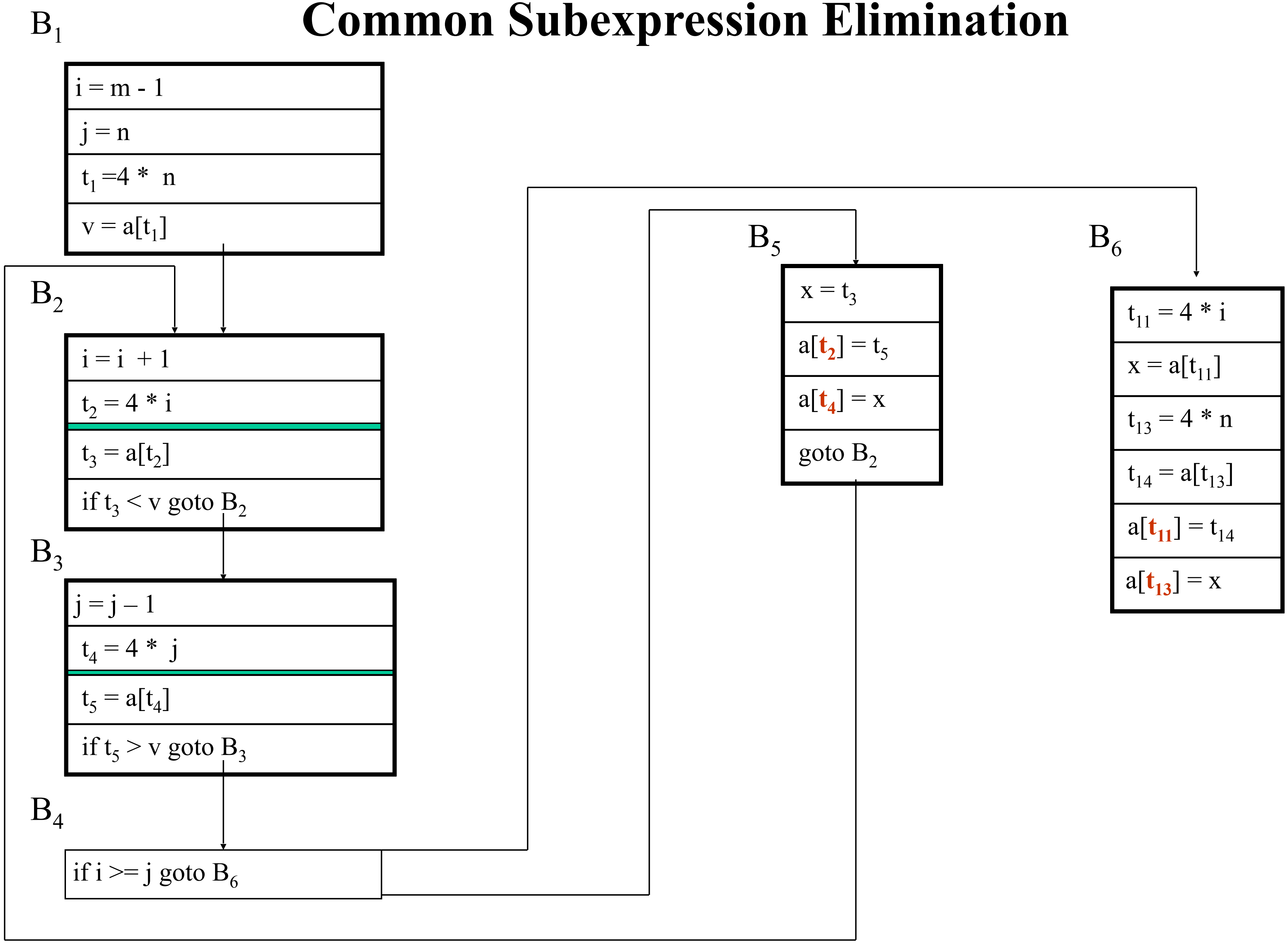
$t_{13} = 4 * n$

$t_{14} = a[t_{13}]$

$a[t_{11}] = t_{14}$

$a[t_{13}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j - 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$x = t_3$

$a[\mathbf{t_2}] = t_5$

$a[\mathbf{t_4}] = x$

goto B_2

B_6

$t_{11} = 4 * i$

$x = a[t_{11}]$

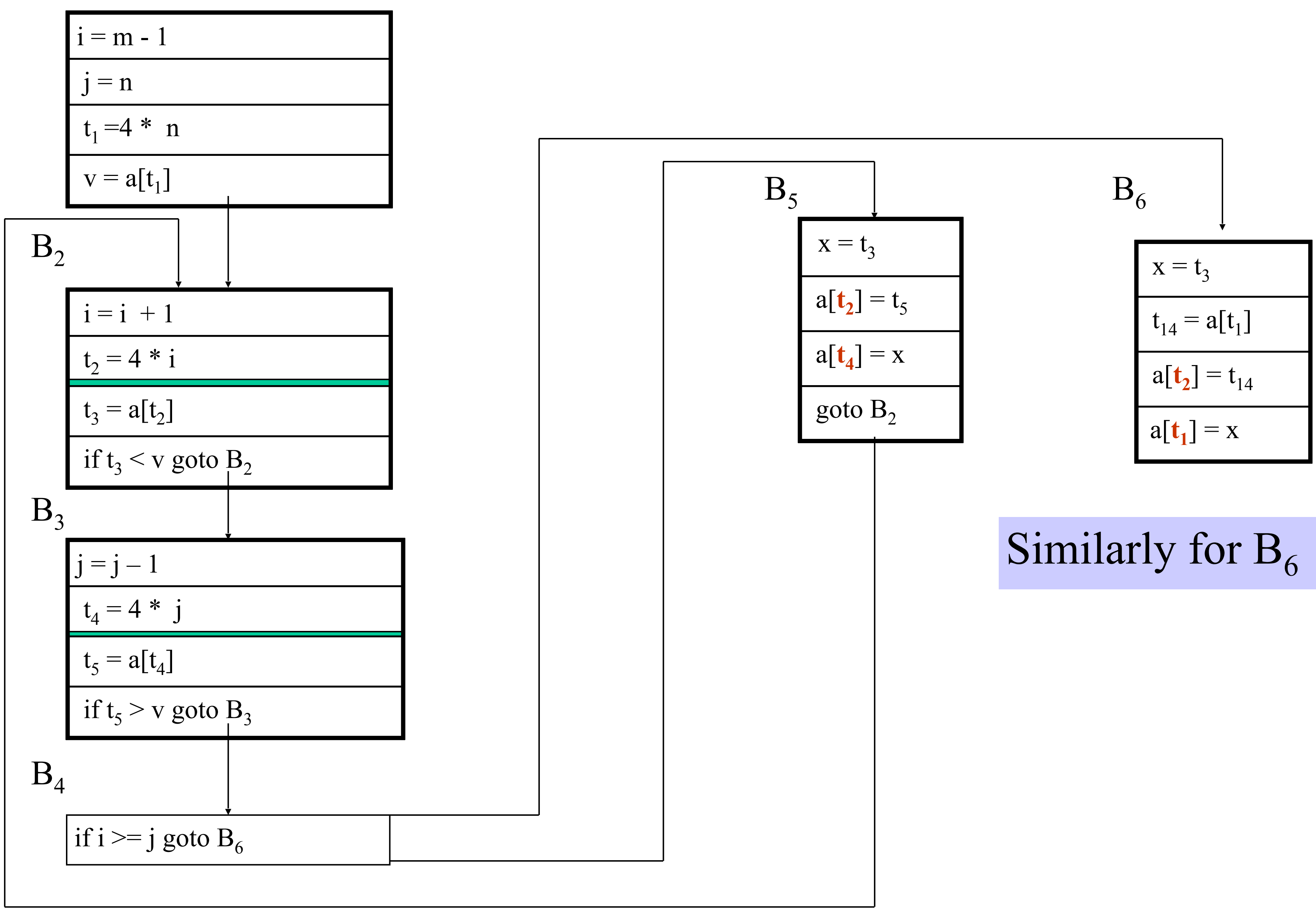
$t_{13} = 4 * n$

$t_{14} = a[t_{13}]$

$a[\mathbf{t_{11}}] = t_{14}$

$a[\mathbf{t_{13}}] = x$

Common Subexpression Elimination



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j - 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$x = t_3$

$a[\mathbf{t_2}] = t_5$

$a[\mathbf{t_4}] = x$

goto B_2

B_6

$x = t_3$

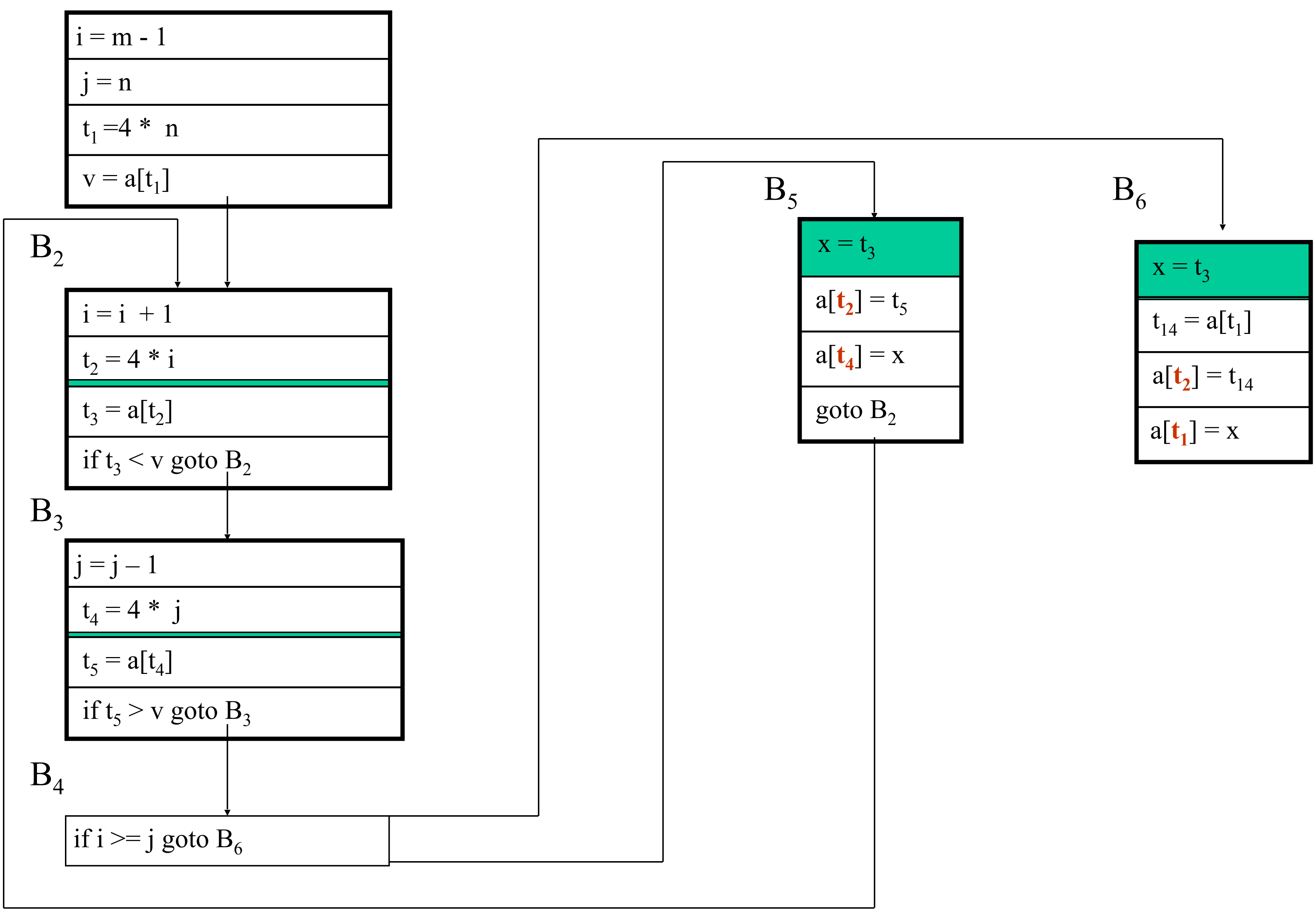
$t_{14} = a[t_1]$

$a[\mathbf{t_2}] = t_{14}$

$a[\mathbf{t_1}] = x$

Similarly for B_6

Dead Code Elimination



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j – 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$x = t_3$

$a[t_2] = t_5$

$a[t_4] = x$

goto B_2

B_6

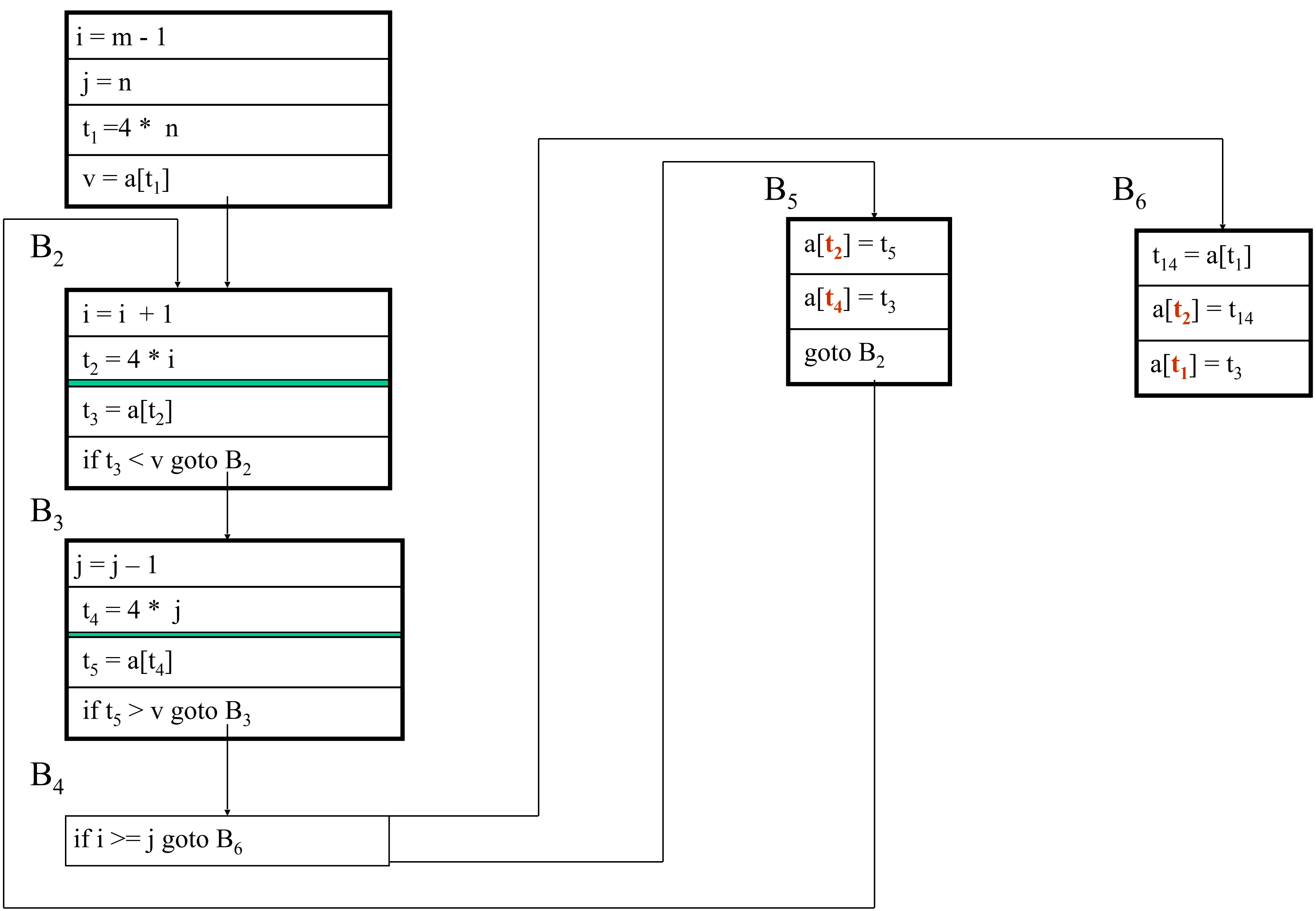
$x = t_3$

$t_{14} = a[t_1]$

$a[t_2] = t_{14}$

$a[t_1] = x$

Dead Code Elimination



B_1

$i = m - 1$

$j = n$

$t_1 =4 * \ n$

$v = a[t_1]$

B_2

$i = i \ + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j – 1$

$t_4 = 4 * \ j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$a[\mathbf{t_2}] = t_5$

$a[\mathbf{t_4}] = t_3$

goto B_2

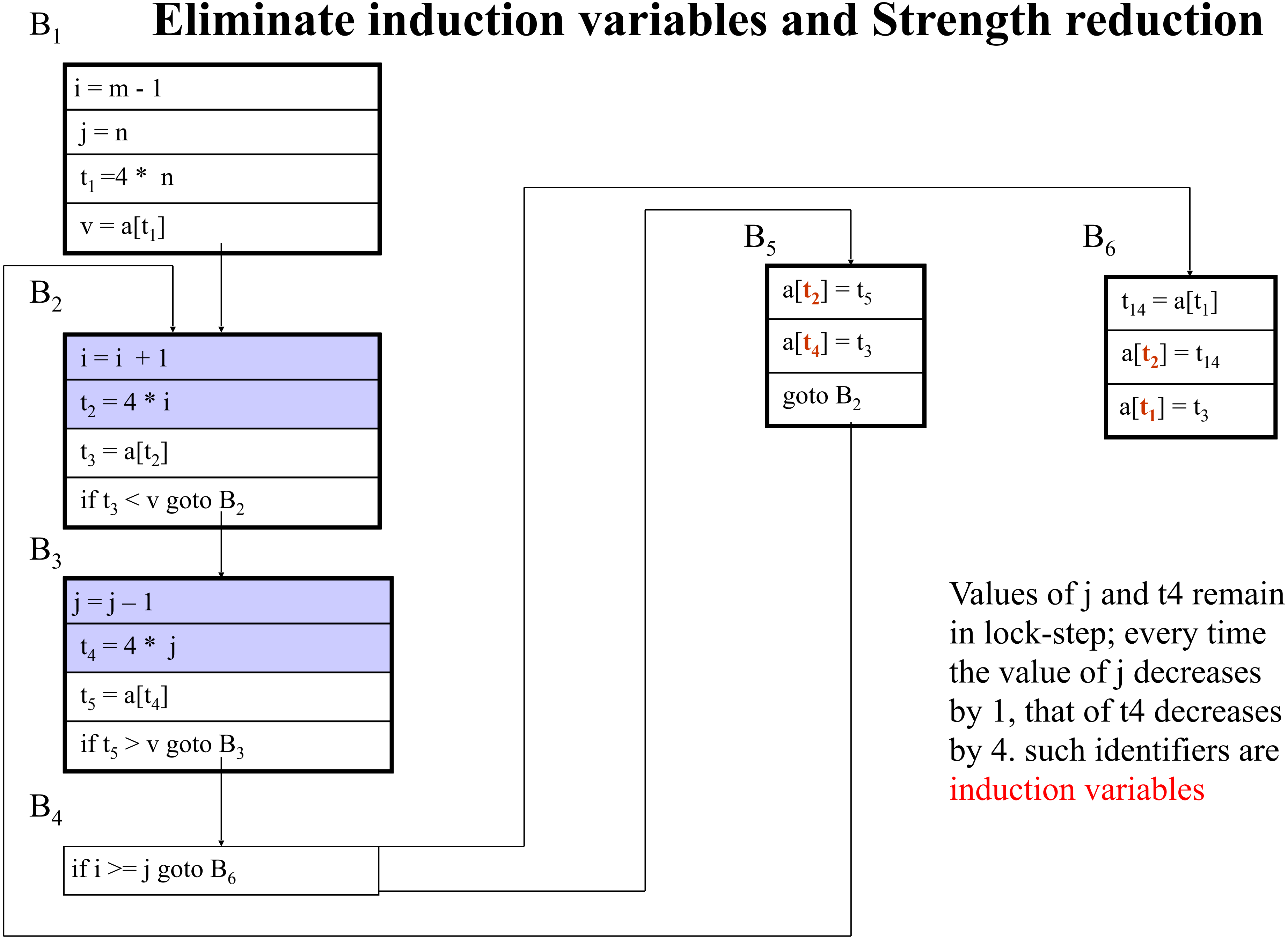
B_6

$t_{14} = a[t_1]$

$a[\mathbf{t_2}] = t_{14}$

$a[\mathbf{t_1}] = t_3$

Eliminate induction variables and Strength reduction



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j - 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$a[t_2] = t_5$

$a[t_4] = t_3$

goto B_2

B_6

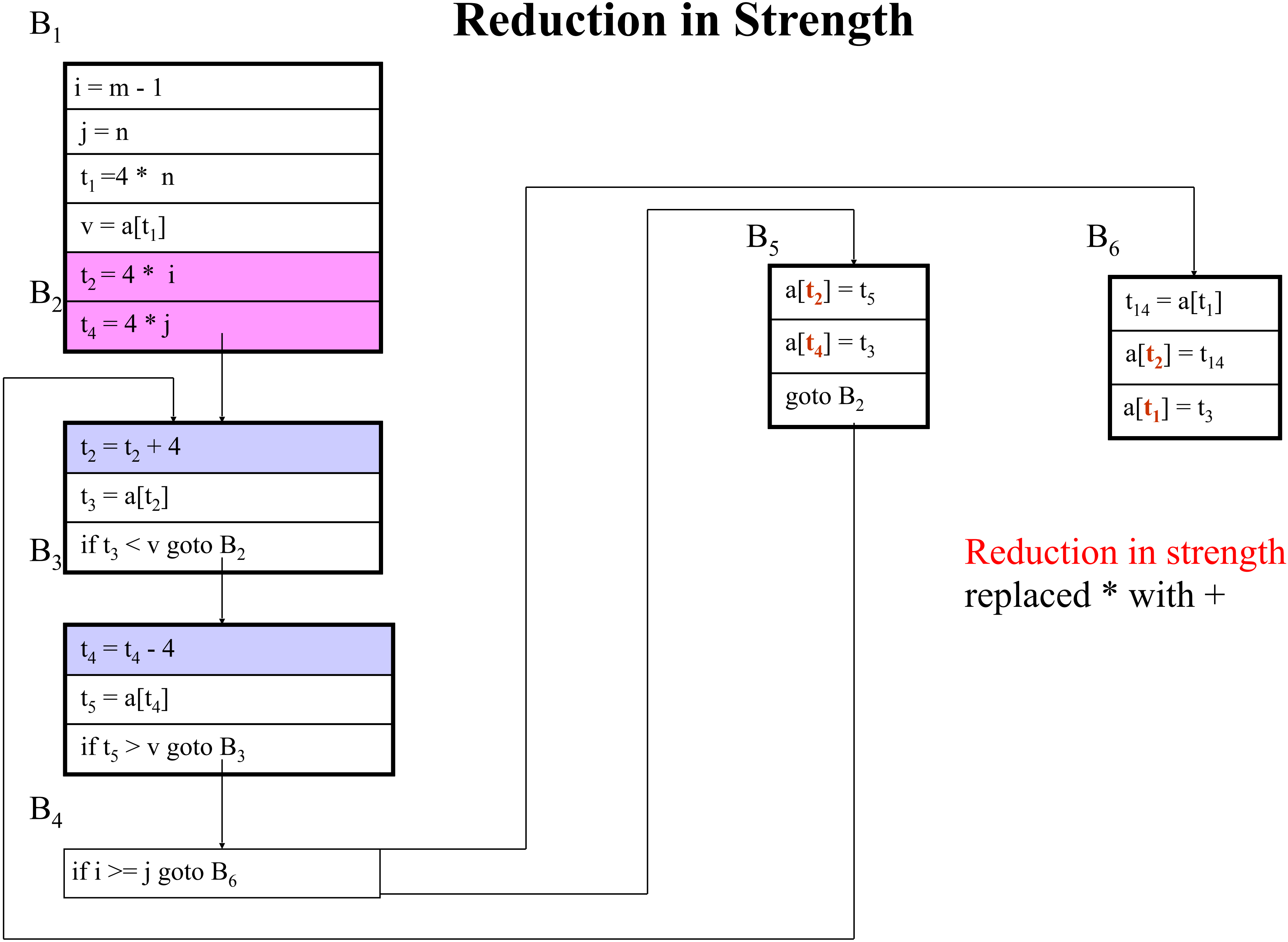
$t_{14} = a[t_1]$

$a[t_2] = t_{14}$

$a[t_1] = t_3$

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 induction variables

Reduction in Strength



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

$t_2 = 4 * i$

$t_4 = 4 * j$

B_2

$t_2 = t_2 + 4$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$t_4 = t_4 - 4$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

$a[t_2] = t_5$

$a[t_4] = t_3$

goto B_2

B_6

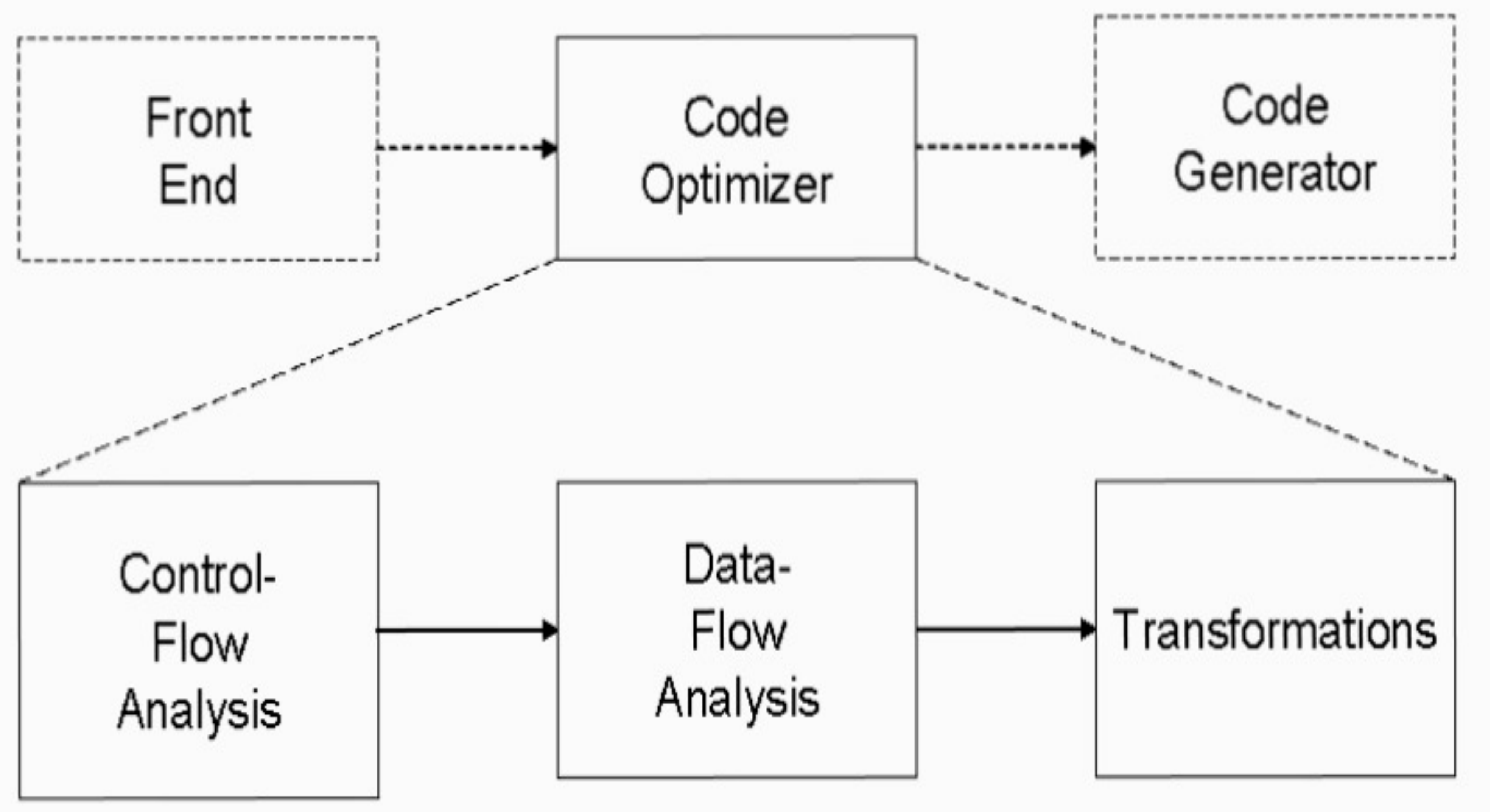
$t_{14} = a[t_1]$

$a[t_2] = t_{14}$

$a[t_1] = t_3$

Reduction in strength-
replaced * with +

Code Optimization and Phases



Global data flow analysis

provides optimisation:

1.Remove variables which are never referenced.

2.Do not make calculations whose results are not used.

3.Remove code which is not called or reachable (*dead code elimination*).

4.Code motion

5.Find uninitialised variables

Data Flow Analysis

- Pre-execution process of ascertaining and collecting information about possible run-time modification, preservation and usage of certain quantities in a program.
- Levels on which we perform data flow analysis
  - Statements
  - Basic blocks
  - Procedures
  - Program
- The first two are local analysis while the last two are global analysis

Data Flow Analysis

- We look at the following problems:
  - Reaching definitions
  - Live variables
  - Definition-use chaining
- The latter two problems can be solved from solutions to the first two problems

Data Flow Analysis

- We assume the following:
  - All relevant local data flow information is available for a particular procedure
  - Any variable aliasing is known
  - The CFG for the procedure is given
- Information can be associated with the top or the bottom of a node
- The sets of information can be conveniently represented by bit vectors

Reaching Definitions

- The problem is to determine the sets RDTOP(x) and RDBOT(x) of variable definitions that can "reach" the top and bottom of each node x in the CFB.
- A *definition-clear path* with respect to a variable v means there is no definition of v on that path.

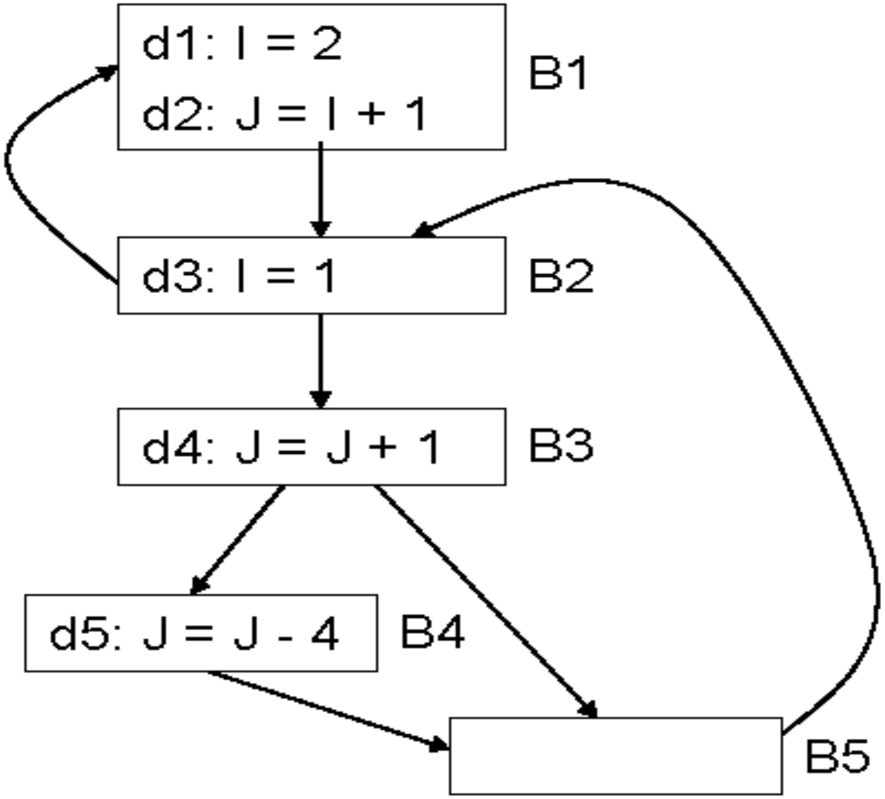
reaching definition

- a **reaching definition** for a given instruction is another instruction, the target variable of which may reach the given instruction without an intervening assignment. For example, in the following code:
  - d1 : y := 3
  - d2 : x := y
- d1 is a reaching definition at d2. In the following, example, however:
  - d1 : y := 3
  - d2 : y := 4
  - d3 : x := y
- d1 is no longer a reaching definition at d3, because d2 kills its reach.

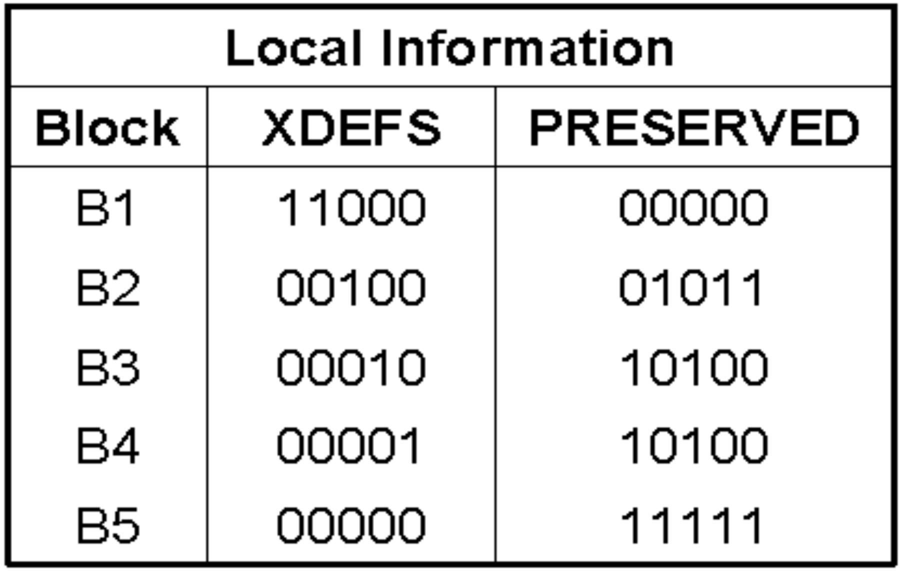
Reaching Definitions

- A definition *d* of a variable *v* in a node *x* is said to reach the top (bottom) of node *y* iff *d* occurs in node *x* and there is a definition-clear path for *v* from *d* to the top (bottom) of node *y*.
- The following information has to be available at the bottom of each node *x*:
  - XDEFS(*x*) – set of locally exposed definitions of node *x*
    - A locally exposed definition is the last definition of a variable in the node.
  - PRESERVED(*x*) – set of definitions *preserved* by node *x*
    - A definition *d* of a variable *v* is said to *kill* all definitions of the same variable that reach *d*.
    - Any definition of a variable *v* that reaches the top of a node *x*, and there is no definition of variable *v* in *x* is said to be preserved by *x*.

Example of Reaching Definitions



Example of Reaching Definitions



Local Information		
Block	**XDEFS**	**PRESERVED**
B1	11000	00000
B2	00100	01011
B3	00010	10100
B4	00001	10100
B5	00000	11111

Live Variables

- Determine the sets LVTOP(*x*) and LVBOT(*x*) of variables that are live.
  - A variable is live if it is referenced without modification
  - A variable *v* in node *y* is said to be live at the bottom (top) of node *x* iff a use of *v* occurs in *y* and there is a definition-clear path for *v* from the use backward to the bottom (top) of node *x*.

Definition-Use Chaining

- Information available for node *x*
  - RDTOP(*x*)
  - LVBOT(*x*)
  - XDEFS(*x*)
  - XUSES(*x*)
- Using both RDTOP(*x*) and XUSES(*x*) together, we can establish a pointer from each use in XUSES(*x*) to the location of zero or more definitions in RDTOP(*x*)
- A similar association can be established between LVBOT(*x*) and XDEFS(*x*)
- This double chaining is called *definition-use chaining*
- Combined with other local information, we know, for a given definition, what uses might be affect by it and, for each use, what definitions can affect it.

Definition-Use Chaining

- These chains are useful for:
  - Dead code elimination
  - Constant propagation
  - Error detection

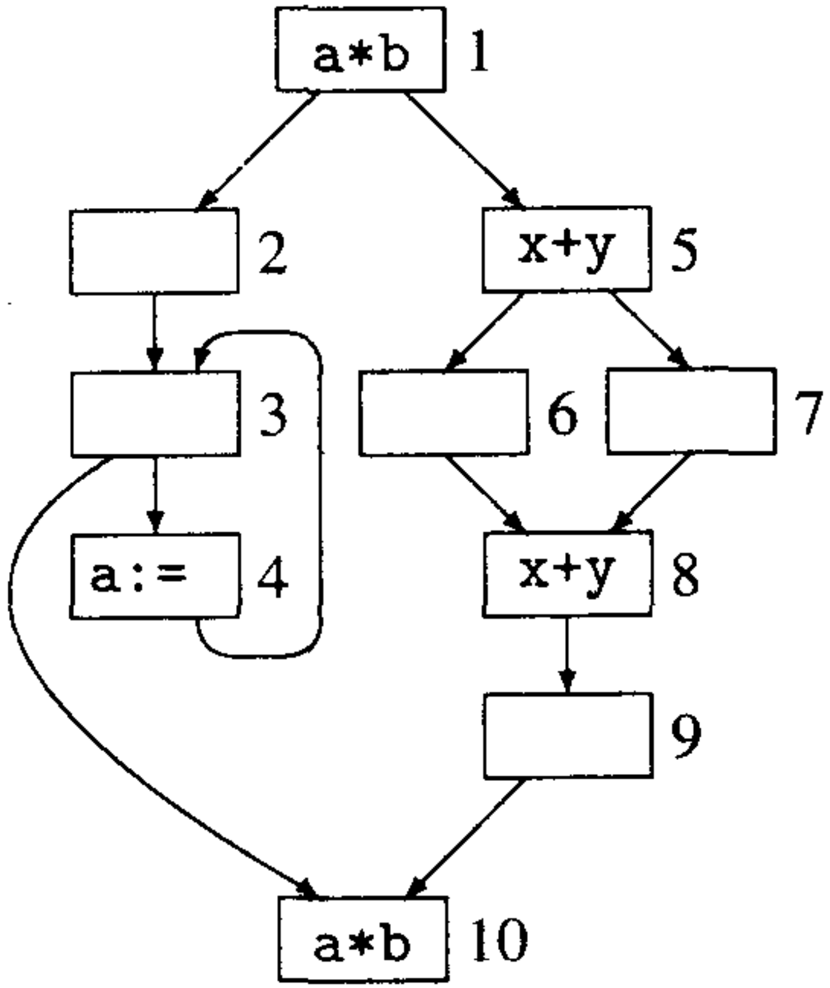
Available Expressions

- An *expression*, such as *y+z*, is *available* at a point *p* in a CFG iff every sequence of branches that the program may take to *p* causes *y+z* to have been computed after that last computation of *y* or *z*
- Thus we may eliminate the redundant computation of some expressions within each node.

Available Expressions

- Assume the following local information
  - NOTKILL(*x*)
    - Set of expressions that are not *killed* in node *x*
    - Expression *y*+*z* is killed iff the value of either *y* or *z* may be modified within node *x*
  - GEN(*x*)
    - Set of expression generated in node *x*
    - Expression *y*+*z* is generated if it is evaluated within node *x* and neither *y* nor *z* is subsequently modified within node *x*.

- a*b:
  - Avail_in = true for blocks 2,5,6,7,8,9
  - Avail_out = true for blocks 1,2,5,6,7,8,9,10
- x+y:
  - Avail_in = true for blocks 6,7,8,9
  - Avail_out = true for blocks 5,6,7,8,9


a*b
1
2
x+y
5
3
6
7
a:=
4
x+y
8
9
a*b
10