11

CODE OPTIMIZATION

11.1 INTRODUCTION

Code Optimization is the optional phase of a compiler which can be applied after Intermediate code generation.

Definition :– It refers to the techniques used by compiler to improve the execution efficiency of the object code.

- → It involves finding out patterns in a program & replacing these patterns by equivalent but more efficient constructs.
- → Increase in efficiency is concerned with size and running time of object program.

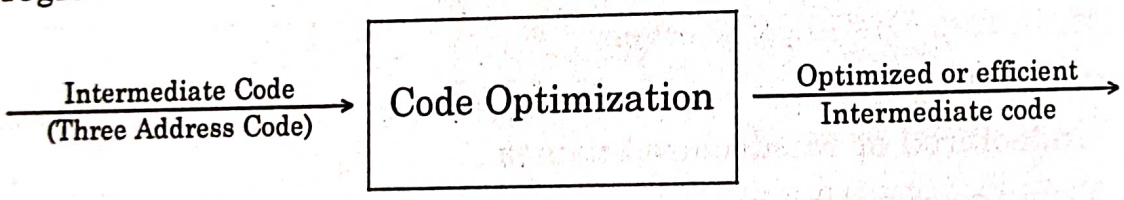


Fig 11.1 : Code Optimization

- → Code optimization is nothing, but actually a code improvement.

Following are the creterias which should be taken care of while optimizing the code :–

1. Code optimization should reduce execution time or space taken by object program.
2. Increase in program efficiency should not change the semantic analysis (meaning) of the program.
3. Algorithm should not be modified in any sense.

11.1.1 CLASSIFICATION OF CODE OPTIMIZATION

Broadly, code optimization can be classified as :–

(*a*) Machine Dependent

(*b*) Machine Independent

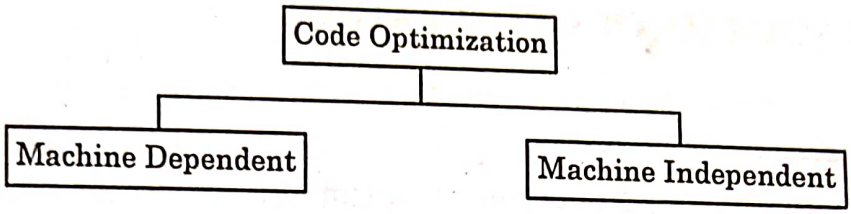


Fig. 11.2 : Classification of Code Optimization

(*a*) **Machine Dependent Optimization :–** Machine dependent optimization requires knowledge of Target Machine. It depends on target machine for instruction set used & addressing modes used for instructions to produce efficient target code. For example :– **Peephole optimization** is type of machine dependent optimization.

Machine dependent optimization can be achieved using following creterias :–

1. Machine's Registers should be utilized efficiently.
2. Efficient utilization of instruction set of a Machine.
3. Allocation of sufficient number of resources to improve the execution efficiency of the program.
4. Immediate instructions should be used wherever required.

(*b*) **Machine Independent Optimization :–**

Machine Independent optimization can be performed independently of target machine for which compiler is generating code *i.e.* optimization is not dependent on Machine's platform.

Machine Independent optimization can be achieved using following creteria :–

1. By Elimination of common sub-expressions.
2. Eliminating unreachable code from the program.
3. Elimination of loop invariant computation *i.e.* eliminating code inside the loop which does not affect the execution of loop and keeping it outside the loop.
4. By Elimination of induction variable.
5. By improving the Algorithm structure *i.e.* algorithmic optimization.

11.1.2 ADVANTAGES OF CODE OPTIMIZATION

1. Program after optimization will occupy less memory space.
2. Program will run faster on performing optimization.
3. Produce better object language program.

11.2 PRINCIPAL SOURCES OF OPTIMIZATION

We can apply various type of transformations on the code to increase the performance or efficiency of the code.

There are various types of sources of optimization which when implemented give rise to increase in efficiency of code.

Following are the principal sources of optimization :–

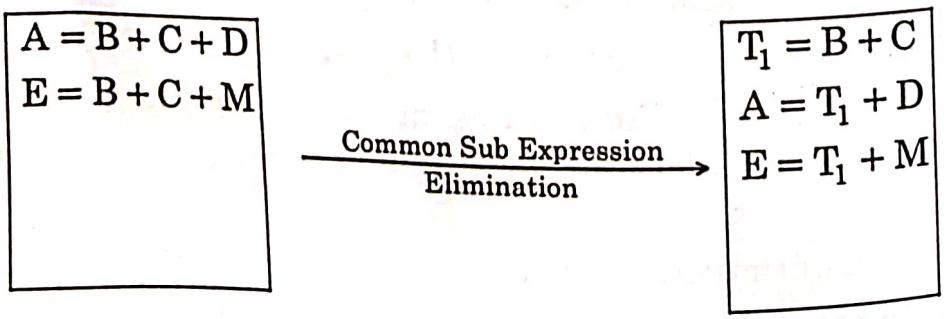
1. **Local Optimization**
   - (*a*) Common Subexpression Elimination
   - (*b*) Constant Folding
   - (*c*) Dead Code Elimination
2. **Loop Optimization**
   - (*a*) Code Motion
   - (*b*) Induction Variable Elimination
   - (*c*) Reduction in Strength
3. **Data Flow Analysis**
4. **Algorithmic Optimization**

1. Local Optimization :– These are optimizations or transformations which are applied with in a straight line.

Following are various transformations which come under local optimization.

(*a*) Common Sub expression Elimination :– An expression E is called common sub expression if it was previously computed. Here, we can avoid recomputing of the expression, if its value is already been computed.

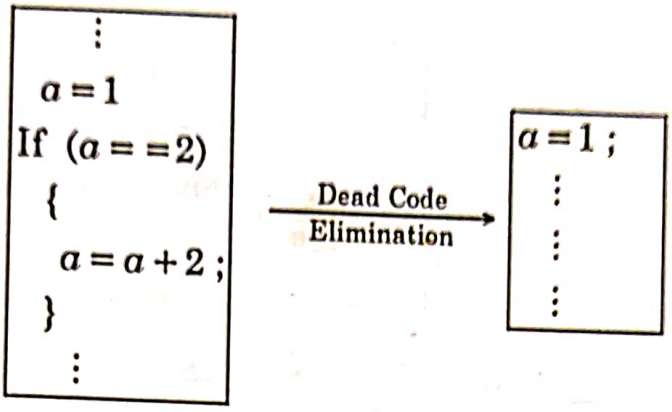
For example :–



Here, B + C is common subexpression.

(b) Dead Code Elimination :- The Dead code is useless code which can be removed from the Program.

For example :-

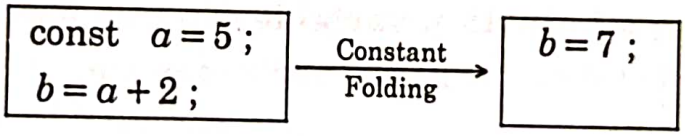


Since, condition if $(a == 2)$ will never be true as $a = 1$.

∴ If statement, is a dead code which never gets satisfied. Hence, it can be eliminated.

(c) Constant Folding :-

In this method of optimization, constant expressions are calculated during compilation. For example :-



2. Loop Optimization :- Inner loops are the greatest source of optimization.

90 – 10 Rule → Inner loops in a program is a place where program spend large amount of time.

90 – 10 Rule states that 90% of the time is spent in 10% of the code.

Thus, inner loops are most heavily travelled part of program.

Therefore loops are the greatest source of optimization.

So, if number of instructions are less in Inner Loops. The running time will be less.

There are three methods to implement loop optimization :-

1. Code Motion
2. Elimination of Induction Variables
3. Reduction in Strength

(a) Code Motion → It is a technique which moves the code outside the loop. If expression lying in loop remains unchanged even after executing the loop for several times. Then, that expression can be placed outside the loop.

For example :–

```
a = 1, b = 2 ;
for (i = 1 ; i <= 10 ; i++)
{
  x = a * b ;
  sum = sum + i ;
}
```

Code Motion →

```
a = 1, b = 2 ;
x = a * b ;
for (i = 1 ; i <= 10 ; i++)
{
  sum = sum + i ;
}
```

As, keeping $x = a * b$ outside the loop will not make any change in result of execution. As x, a, b are independent from loop parameter i.

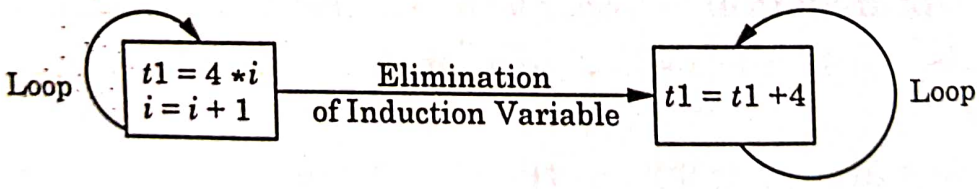
(b) Elimination of Induction Variables → This optimization will decrease the total number of instructions and will decrease the execution time of loop.

∴ x, a, b are loop invariant variables which do not very during execution of loop

∴ Code Motion is also Called Eliminating Loop Invariant Computation.

A variable i is called **induction variable** of a loop if every time it changes its value, it is incremented or decremented by some constant.

For example :–



Here, each increment in value of i lead to increment of $t1$ by 4.

∴ i and $t1$ are in locked state.

Hence i, $t1$ are inductions variable. Above, we have eliminated inductio variable i.

(c) Reduction in Strength → Some operators have different strength. For e Strength of * is higher than +. Higher strength operators can be replaced by low strength operators.

For example :–

```
for (i=1 ; i<=5 ; i++)
 {
   a=i*2 ;
 }
```

Reduction in Strength →

```
int a=0 ;
for (i=1 ; i<=5 ; i++)
 {
   a=a+2 ;
 }
```

'+' operation is much cheaper than '*' operation which is costly.

3. Algorithm Optimization → Algorithm optimization is the important source of optimization in the running time of a program. We can't achieve optimization until and unless we don't have a good algorithm. Good Algorithm means having less time Complexity

Comparison of Bubble Sort & QuickSort :–

Running Time of Algo's

Non optimized Bubble Sort = 100 n^2 microseconds

Optimized Bubble Sort = 50 n^2 microseconds

Quick Sort = 500 n log n microseconds

where n = no. of items to be sorted.

∵ Time complexity of QuickSort is less than Bubble Sort & Optimized Bubble Sort because rate of growth of n log n is less than n^2.

Running Times of 3 Object Programs :–

Algorithm	n	
	100	1000
Bubble Sort	1 sec	100 sec
Optimized Bubble Sort	.5 sec	50 sec
Quick Sort	.1 sec	1.5 sec

As, it can be seen from the Table, that unoptimized Quicksort is 5 times faster than optimized bubble sort for n = 100 & 33 times faster for n = 1000.

So, optimization does not make any difference if we don't have a good algorithm. So, the good Algorithm *i.e.* algorithm with less time complexity, running on unoptimized compiler can be better than bad algorithm running on optimized compiler.

4. **Date Flow Analysis** → In Data Flow Analysis, we analyze the flow of data from one block to another block of a program. It determines definition and use of data in a program.

The whole program is divided into blocks. Blocks are joined to make a flow graph. Finding all such informations can help to apply optimization called Data Flow Analysis.

11.3 BASIC BLOCKS

Definition :– Basic Block is a sequence of consecutive statements in which flow of control enters at the beginning and when entered are executed in a sequence without halt or possibility of branch except at end.

11.3.1 Algorithm for Partitioning into Blocks

First of all the program should be in three address code, if it has to be divided into Basic Blocks.

Input : Sequence of three address statements.

Output : Basic Blocks

Method :– 1. First determine the **Leaders** by using following rules.

(*a*) The first statement is a **Leader.**

(*b*) Any target statement of conditional or unconditional goto is a Leader.

(*c*) Any statement that immediately follows a goto or unconditional goto is a **Leader.**

2. Basic Block is formed by starting at the Leader Statement and ending just before the next Leader statement.

Example 1. Consider following program code for computing dot product of two vectors *a* and *b* of length 20. Partition it into Basic Blocks.

```
prod = 0 ;
i = 1 ;
do
{
    prod = prod + a [ i ] * b [ i ] ;
      i = i + 1 ;
}
    while ( i < = 20 ) ;
```

Ans. First of all, we will convert this program into three address code. If suppose, each word takes 4 bytes.

Three address code will be :–

1. prod = 0
2. i = 1
3. $t1 = 4 * i$
4. $t2$ = addr (a) – 4
5. $t3 = t2$ [$t1$]
6. $t4$ = addr (b) – 4
7. $t5 = t4$ [$t1$]
8. $t6 = t3 * t5$
9. prod = prod + $t6$
10. $i = i + 1$
11. If $i \leq 20$ goto (3)

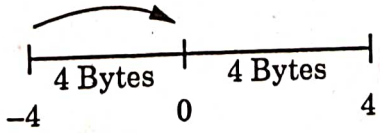
As, starting address of array a *i.e.* addr (a) = 0.

$\therefore$ $t2 = 0 - 4 = -4$

As $i = 1$, $t1 = 4 * i = 4 * 1 = 4$

$\therefore$ $t3 = t2$ [$t1$] = – 4 [4]

$\therefore$ – 4 [4] means moving 4 bytes from index – 4, which will be 0th location.



$\therefore$ $t3 = t2$ [$t1$] will give starting element of array a *i.e.* at 0th location.

Similarly, $t5 = t4$ [$t1$] will give starting element of array b *i.e.* at 0th location.

According to Algorithm,

Statement (1) is a leader by Rule 1 (a) of Algorithm.

Statement (3) is also leader by Rule 1 (b).

As, statement (3) is a target of conditional goto at statement (11).

$\therefore$ Statement 1 and 2 will be in same block.

Statement 3 to 11 will form another basic block.

∴ **BLOCK 1**

1. prod = 0
2. $i = 1$

BLOCK 2

3. $t1 = 4 * i$
4. $t2$ = addr $(a) - 4$
5. $t3 = t2\ [t1]$
6. $t4$ = addr $(b) - 4$
7. $t5 = t4\ [t1]$
8. $t6 = t3 * t5$
9. prod = prod + $t6$
10. $i = i + 1$
11. If $i \leq 20$ goto (3)

11.4 FLOW GRAPH

A flow graph is a directed graph which shows the relationships between basic blocks.

(*a*) **Nodes** of the flow graph are represented by Basic blocks.

(*b*) **Edges** of flow graph are represented by directed arrows which tells the order of execution.

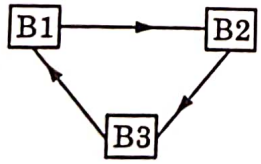
Directed edge from Block B1 to Block B2 means B2 will follow B1 during execution.

i.e. B1 is predecessor of B2.

Loops → Loop is a collection of nodes in the flow graph such that

1. All such nodes are **strongly connected** *i.e.* there is path from any node to any other node with in that loop.

e.g.



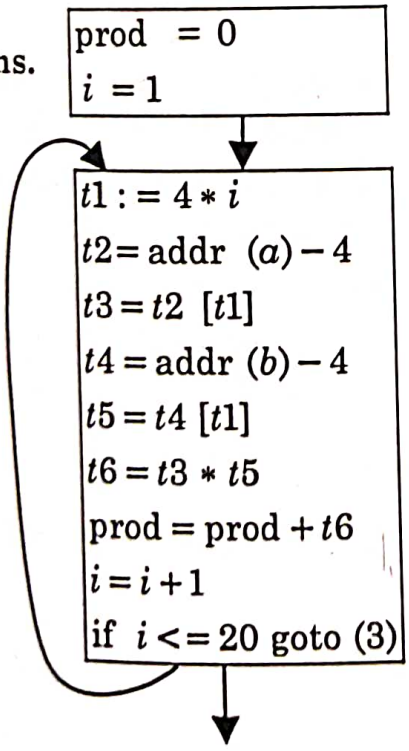
In this graph, if there is a path from Block B1 to B2 then there is also a path from B2 to B1.

2. The collection of nodes has **unique entry**. That means there is only one path from a node outside the loop to the node inside the loop.
3. If loop consist of only one node, then it is called **one node loop.**

Example 2. Construct the flow graph for the following three address code.

1. prod = 0
2. $i = 1$
3. $t1 = 4 * i$
4. $t2 = \text{addr}\ (a) - 4$
5. $t3 = t2\ [t1]$
6. $t4 = \text{addr}\ (b) - 4$
7. $t5 = t4\ [t1]$
8. $t6 = t3 * t5$
9. prod = prod + $t6$
10. $i = i + 1$
11. If $i <= 20$ goto (3).

Ans.



Block B1

Block B2

Example 3. Consider following program code :–

```
int f = 1 ;
for (i = 1 ; i <= 10 ; i++ )
```

$f = f * i$;

Design the flow graph for the above code.

Ans. First of all convert the code into three address code as :–

1. $f = 1$
2. $i = 1$
3. if $i <= 10$ goto (5)
4. goto (8)
5. $f = f * i$
6. $i = i + 1$
7. goto (3)
8. Ending statement

Leaders in three address statements are :–

→ Statement (1), as it is first statement.

→ Statement (3), as it is target of goto in statement (7).

→ Statement (4), because it immediately follows conditional goto statement (3).

→ Statement (5), as it follows goto statement (3).

→ Statement (8), as it follows goto statement (4).

∴ Blocks will be

Block B1 :

1	$f = 1$
2	$i = 1$

Block B2 :

3	If $i <= 10$ goto (5)

Block B3 :

4	goto (8)

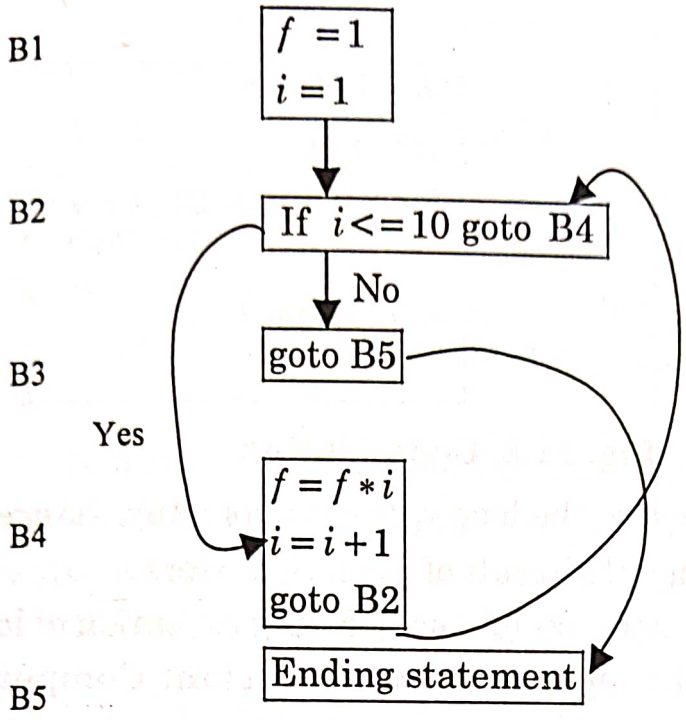
Block B4 :

5	$f = f * i$
6	$i = i + 1$
7	goto (3)

Block B5 :

8	Ending Statement

The Flow graph joining above blocks will be



11.5 LOOP OPTIMIZATION

Loop optimizations are important because many programs spend a large percent of time inside loops. Since, body of the loop is executed again and again, a small improvement in the execution of the loop body can result in a remarkable Overall improvement in the performance of the program. We also have the loops inside loops *i.e.* inner loops. An improvement in the innermost loop body is always targeted by the optimizing compilers. As we have discussed earlier 90 – 10 rule, in which 90% of program time is spent in executing 10% of the code *i.e.* inner loops. So, loops are the important candidates for optimization.

Following are the commonly used techniques of loop optimization :–

1. **Code Motion**
2. **Elimination of Induction Variables**
3. **Reduction in Strength**

11.5.1 Code Motion

Sometimes, there is a code inside a loop which can be moved outside the loop causing no change in the meaning of a program.

By doing this, the length of code inside loop will decrease, which leads to improvement in running time of a program. Obviously, loop will take less execution time to execute less number of statements in a loop.

For example :–

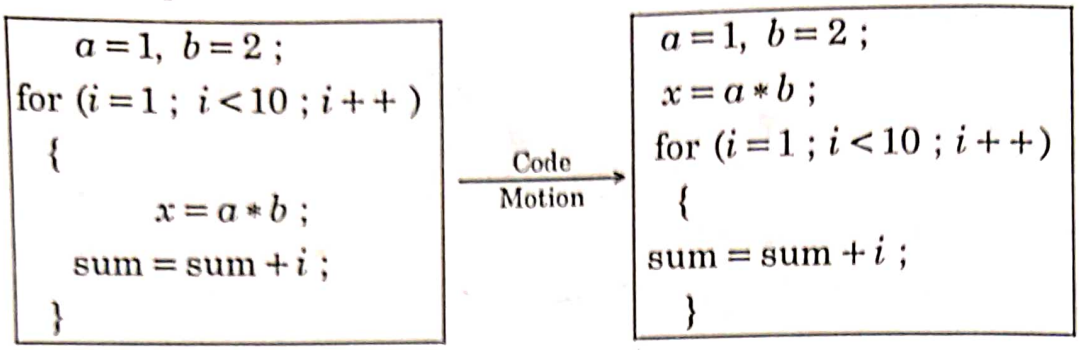


Fig. 11.3. Code Motion

As, keeping $x = a * b$ before the loop will not make any change in the meaning of program and will not change the result of execution. Here x, a, b are loop invariant variables which do not vary (change) during execution of loop. *i.e.*why code motion can also be called as **Eliminating Loop Invariant Computation.**

Example 4. Perform Code Motion *i.e.* Eliminate Loop invariant computation from the flow graph given in Example 2.

Ans. Flow graph obtained after applying code motion is shown below :–

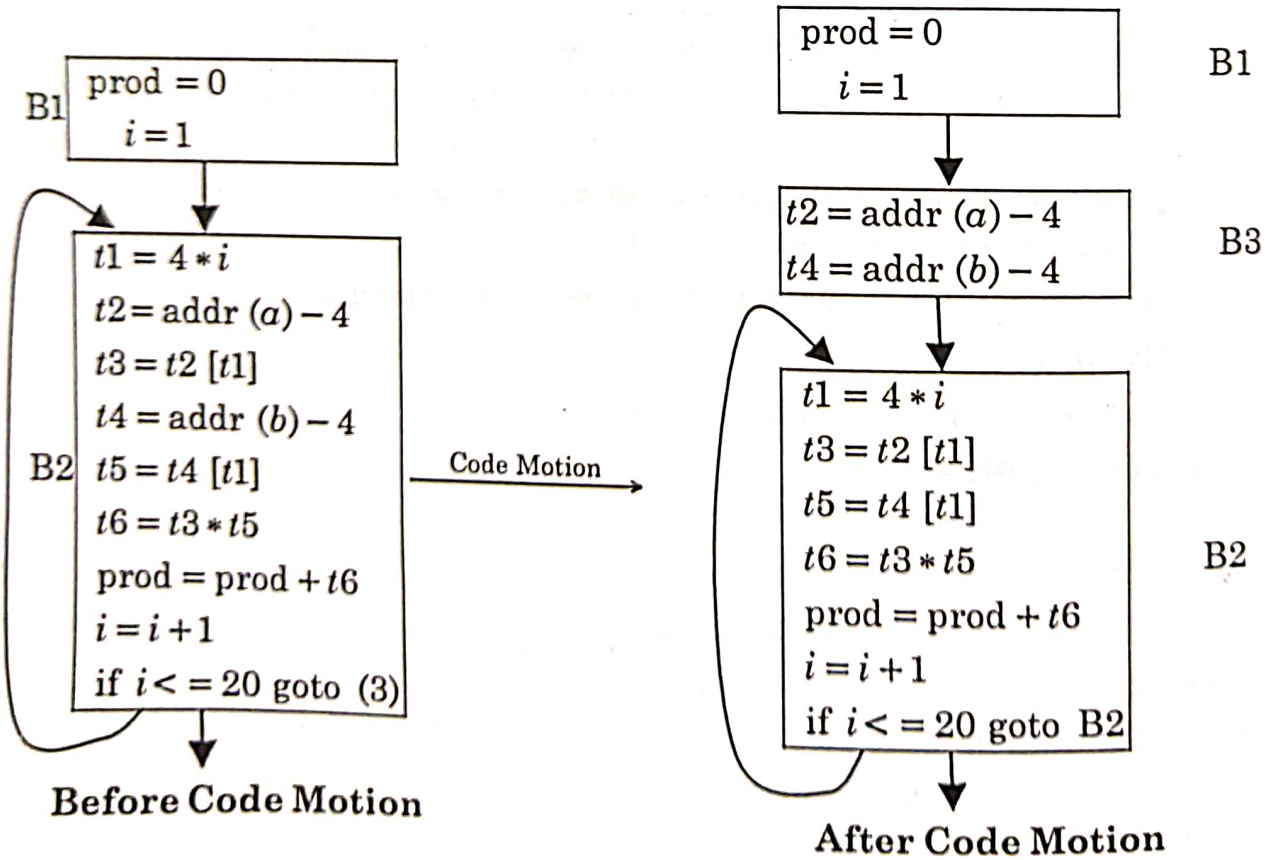


Fig. 11.4 : Code Motion

In Flow graph before code motion $t2 = \text{addr}(a) - 4$ and $t4 = \text{addr}(b) - 4$ are **op invariant computations.**

i.e., the value of $t2$ and $t4$ will remain same throughout the execution of loop in lock B2. Since, addr (a) & addr (b) are statically allocated arrays *i.e.* Their cation will remain same. Due to this $t2$ and $t4$ will have same value during the ecution of loop. So, it is better to remove statements $t2 = \text{addr}(a) - 4$ and $t4 = \text{addr}(b) - 4$ from loop, and keep them before the loop. It can be done by creating w Block B3 before B2.

→ We have to Replace goto (3) by goto B2 since, statement (3) *i.e.* $t1 = 4 * i$ ow becomes the leader of Block B2 after code motion. So, Block B2 which earlier ntains 9 statements. After code motion, it contains 7 statements. Decrease in tements in a loop will lead to decrease in execution time.

11.5.2 Induction Variable Elimination

Definition :– A variable i is called an **induction variable** of loop L if every me the variable i changes values *i.e* everytime i either gets incremented or cremented by some constant.

For example :– In Loop **for $i = 1$ to 100**

i is an induction variable. Elimination of this induction variable can decrease e total number of instructions in a loop. Consider following block B which has duction variable i because i gets incremented by 1 each time loop is executed.

Since, $t1 = 4 * i$.

Therefore, each increment in value of i leads to increment of $t1$ by 4.

$\therefore$ i & $t1$ are in locked state.

We can remove induction variable i which will decrease the number of structions of Block in a loop. So, it is better to increment $t1$ by 4 directly by $t1 = t1$ 4 instead of first incrementing i. ($i = i + 1$) and then substituting it in $t1 = 4 * i$.

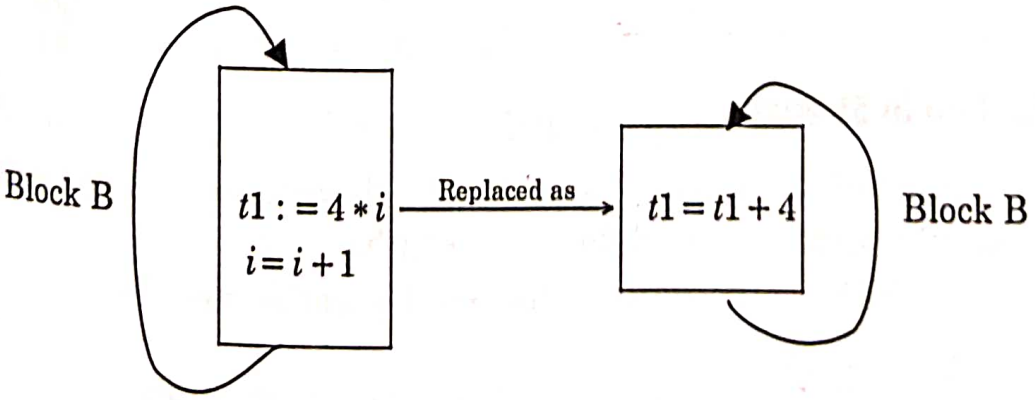


Fig. 11.5. Induction Variable Elimination

Example 5. Eliminate induction variable i from the flow graph obtained after code motion in previous example.

Ans.

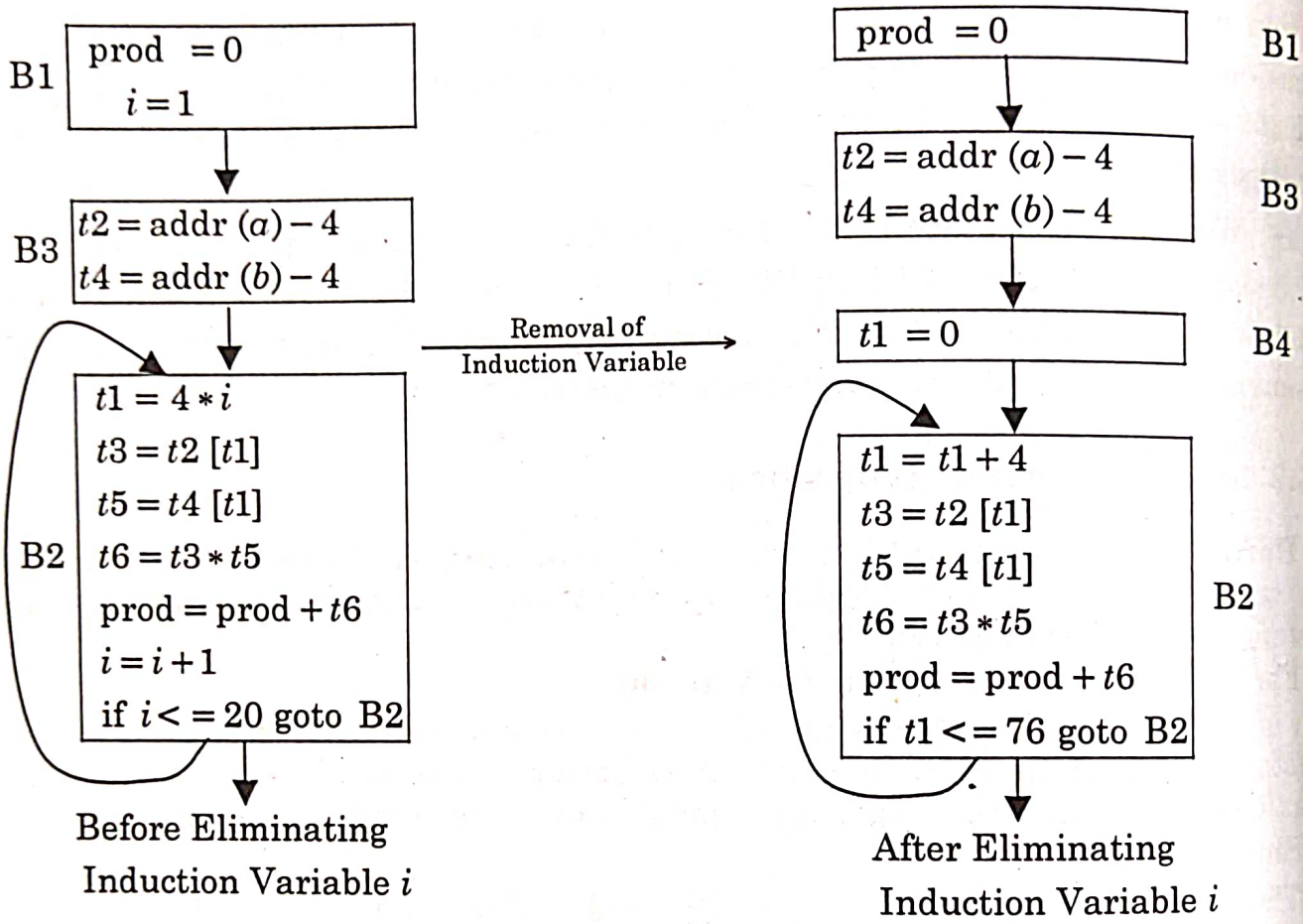


Fig. 11.6 : Induction Variable Elimination

Statement if $i <= 20$ **goto B2** will be be replaced with if $t1 <= 76$ goto B2 Because i was incremented by 1 each time. So, Block B2 will be executed 19 times. Therefore Each increment of 4 in $t1$ means $t1$ will be incremented upto $19 * 4 = 76$ times.

11.5.3 Reduction in Strength

The replacement of an expensive operation by cheaper one is called reduction in strength. Different operators have different strengths.

Addition operator has less strength than multiplication operator.

i.e. '+' is cheaper than the '*'.

So, '*'can be replaced by '+'operator.

For example :–

$$\boxed{\begin{array}{l} i = 1 \\ t1 = 4 * i \end{array}} \xrightarrow{\text{Replaced with}} \boxed{\begin{array}{l} t1 = 0 \\ t1 = t1 + 4 \end{array}}$$

Another Example :–

So, String concatenation operator (| |) can be replaced with length determination and addition operation. If S1, S2 are two strings.

$$\boxed{\text{length (S1 || S2)}} \xrightarrow[\text{With}]{\text{Replace}} \boxed{\text{length (S1)+length (S2)}}$$

Length determination & '+' operation are cheaper than | |.

11.6 DAG REPRESENTATION OF BASIC BLOCKS

Definition of DAG :– A directed Acyclic graph which represents a single basic block.

or

It is a directed graph with no cycles which shows how value computed by each statement in basic block is used in subsequent statements in the block.

DAG can have nodes with the following types of Label :–

1. **Leaf nodes** are labeled by identifiers, variable names and constants.
2. **Interior nodes** contain operators.
3. **Interior node** also contain computed values *i.e.* Result or identifier is attached to the operator.

Each Basic block of flow graph can be represented by DAG.

Example 6. Construct DAG for following three address statements.

$x = y + z$

$c = x + b$

Ans.

→ After executing statement (1), DAG will be figure (*i*).

→ After executing statement (2), complete DAG will be figure (*ii*).

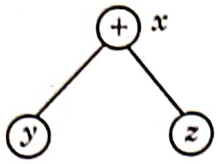


Fig.(*i*)

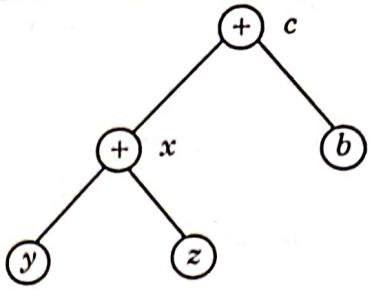


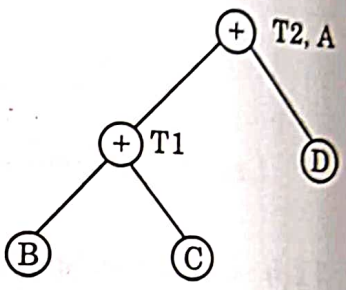
Fig.(*ii*)

Example 7. Construct DAG for following statement A = B + C + D.

Ans. First of all convert it into three address statement.

i.e. T1 = B + C
T2 = T1 + D
A = T2

DAG will be :–

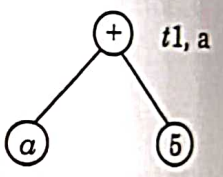


Example 8. Construct DAG for $a = a + 5$

Ans. Three address code will be

$t1 = a + 5$
$a = t1$

DAG will be :–



Example 9. Construct the DAG for following three Address Statements.

1. $t1 = 4 * i$
2. $t2 = \text{addr}\ (a) - 4$
3. $t3 = t2\ [t1]$
4. $t4 = 4 * i$
5. $t5 = \text{addr}\ (b) - 4$
6. $t6 = t5\ [t4]$
7. $t7 = t3 * t6$
8. $t8 = \text{prod} + t7$
9. $\text{prod} = t8$
10. $t9 = i + 1$
11. $i = t9$
12. if $i <= 20$ goto (1)

Ans. Steps to construct a DAG :-

→ Variables and constants in a block will act as **leaf nodes** of DAG.

For example prod, addr (*a*), addr (*b*), 4, *i*, 1, 20 will be leaf nodes.

→ **Interior nodes** will have the operators as their labels. The computed value or Results are attached with the operators on interior nodes.

For example :- t2 = addr (a) – 4

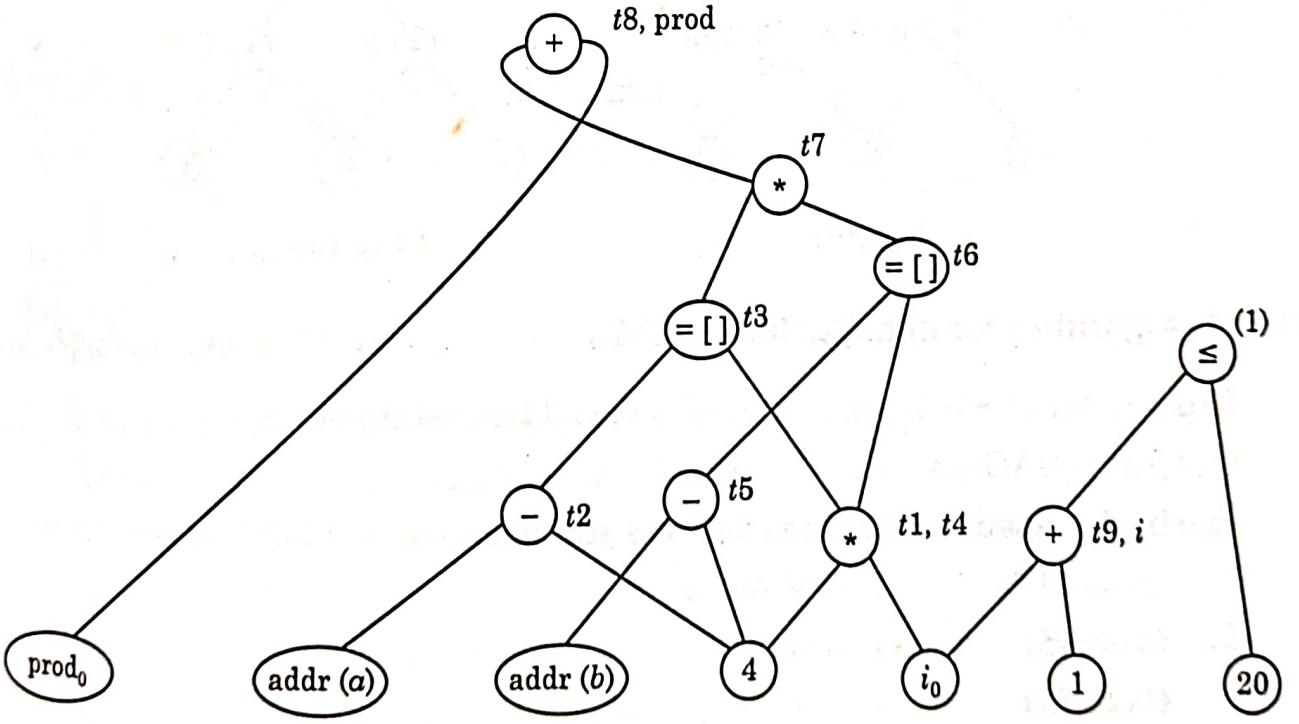
In this statement, addr (*a*) and 4 will be leaf nodes. The operator (–) applied on them will act as there parent node. t2 is attached with the Node (–).

→ Leaf nodes subscripted with 0 (*e.g.* $prod_0$, i_0) denote initial values of names which help to distinguish labels from attached identifiers in interior nodes.

→ If operator & variable are both children of same parent, then variable should be connected to its parent using R.H.S. edge.

For example :- Operator (*) and variable (prod) are both children of root node (+).

Therefore $prod_0$ should be connected by R.H.S. edge and * on L.H.S. edge of root node (+). ∴ Edges are tilted at root node because prod should be R.H.S.



Example 10. Construct DAG for following basic block.

D = B * C

E = A + B

B = B * C

A = E – D

Ans. Steps to Construct DAG :-

→ Statement (1) will generate first figure (*i*).

→ Statement (2) combined with (1) generate fig (*ii*).

→ Statement (3) will generate figure (*iii*).

As D and B contain same value *i.e.* B * C. Therefore, B is attached with variable D on node *. Node (B) at leaf level is changed to (B_0).

→ Statement (4) will generate figure (*iv*).

Figure (*iv*) is the required answer.

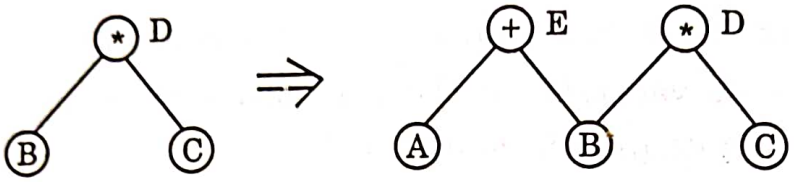


Fig. (*i*) **Fig. (*ii*)**

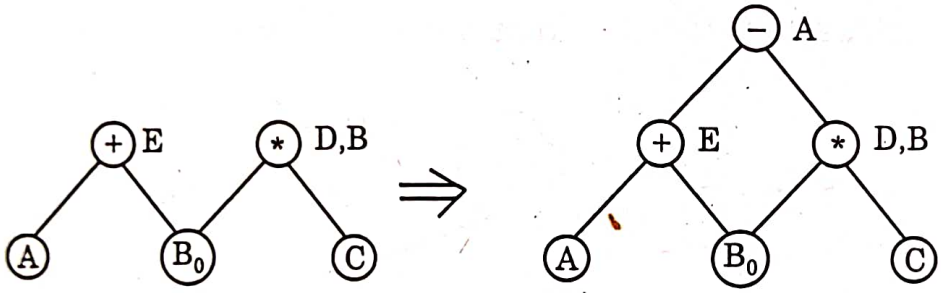


Fig. (*iii*) **Fig. (*iv*)**

11.6.1 Algorithm for construction of DAG

Input : Basic block consisting of three address statements.

Output : DAG

Method : Assume the three address statements are of following types :-

Case (1) $x = y$ op z

Case (2) $x =$ op y

Case (3) $x = y$

DAG for above statements can be constructed by following steps :-

Step 1 :- If node (y) is undefined, create leaf node labeled node (y).

In case (1), if node (z) is undefined create a leaf node labeled node (z).

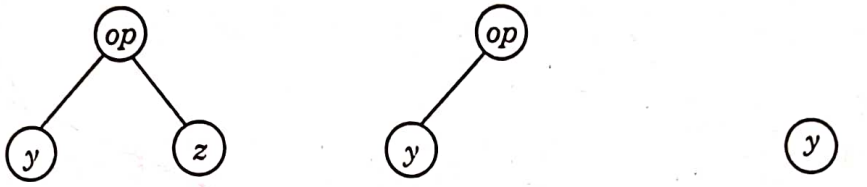
Case 1 **Case 2** **Case 3**

Step 2 :- In case (1), determine if there is node labeled op, whose left child is node (*y*) and whose right child is node (*z*). If not, create such a node. Let *n* be the new node created *i.e.* node (op).

In case (2), determine whether there is node labeled op, whose only child is node (*y*). If not create a node (op).

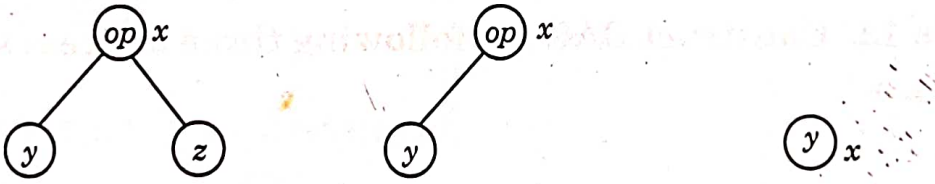
Let *n* be new node created *i.e.* node (op).

In case (3), Let *n* be node (*y*).



Case 1 Case 2 Case 3

Step 3 :- Append *x* to the list of attached identifiers for node *n* formed in step 2. Delete *x* from list of attached identifiers for node (*x*). Set node (*x*) to *n*.



Case 1 Case 2 Case 3

11.6.2 Applications of DAG

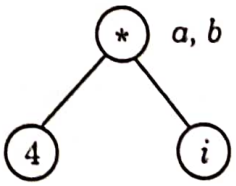
(1) It helps to determine common sub-expressions *i.e.*

For example, *a* and *b* are both equal to same expression.

$a = 4 * i$

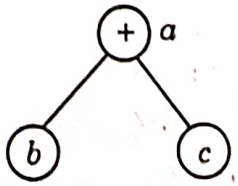
$b = 4 * i$

It can be converted into DAG



Here *a* and *b*, both attached to node *.

(2) DAG helps to determine which identifiers are used in the block *i.e.* Leaf nodes.



For example :- Statement $a = b + c$ is **defining** *a* and **using** *b* and *c*. When DAG for statement will be designed, *b* and *c* will be Leaf nodes.

(3) DAG helps to determine which statements compute values which could be used outside the block.

(4) It simplifies quadruples by removing common sub expressions. Because removing sub expressions lead to decrease in number of identifiers. So, less number of identifiers have to be entered into quadruple array.

Example 11. Construct DAG for following expressions

$(a - b) + c * (d / e)$.

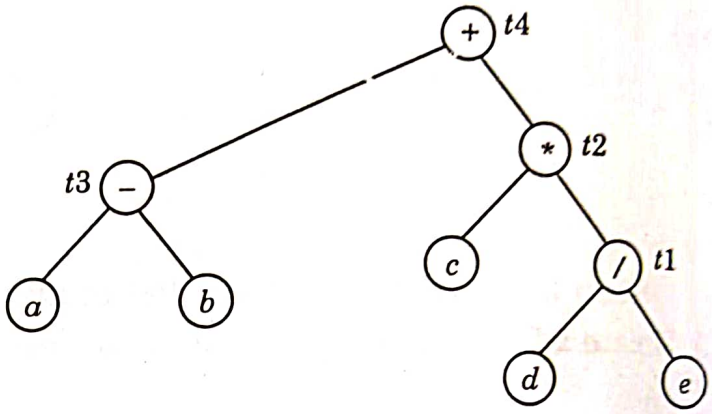
Ans. First of all, convert this expression into three address code.

$t1 = d / e$

$t2 = c * t1$

$t3 = a - b$

$t4 = t3 + t2$



Example 12. Construct DAG for following three address statements.

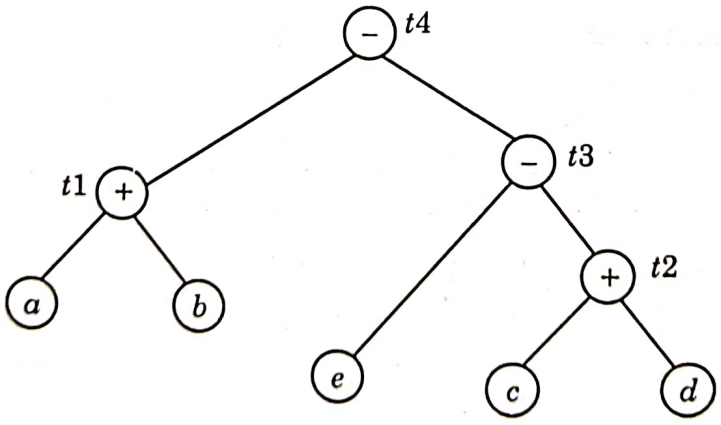
$t1 = a + b$

$t2 = c + d$

$t3 = e - t2$

$t4 = t1 - t3$

Ans.

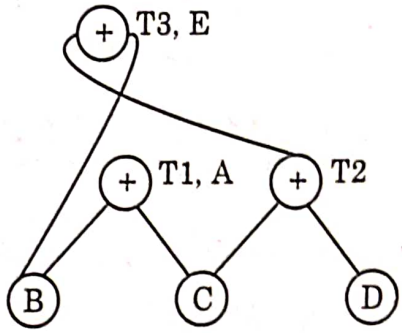


Example 13. Construct DAG for

A = B + C

E = C + D + B

Ans. First of all convert this code into three address code statement.



T1 = B + C
A = T1
T2 = C + D
T3 = T2 + B
E = T3

Root Node has two child identifier (B) and operator (+).

By Rule, identifier (B) should be on R.H.S of Root node. Therefore, Root node has tilted edges.

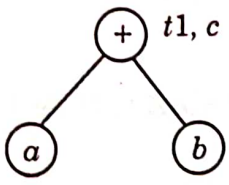
11.7 VALUE NUMBERS

- → Each node of a DAG is given a number to determine whether a node with particular children exists.
- → Hashing technique is used for this purpose.
- → Each node is given a value computed by basic block.

Value Number Method : Checking a node with particular children & particular operator using Hash Table.

For *e.g.*

Statement no.	Statement
1	$a = 5$
2	$b = 6$
3	$c = a + b$



Value Number	Symbol
1	*a*
2	*b*
3	*c*

If *a*, *b* are two nodes of DAG. We have to find an entry into hash table where a list of nodes having *a* and *b* as children can be found.

By hashing value *a* and *b*, we can obtain a pointer into hash table or Quadruple to reach at node having *a* and *b* as children.

i.e.

h (*a*) →
h (*b*) →

Operator	Operand 1	Operand 2	result
+	*a*	*b*	*c*
:	:	:	:
:	:	:	:

Quadruple

Here h is a hashing function.

Here $h(a)$ and $h(b)$ will point to the operator +. Since, + is parent of a and b.

11.8 ALGEBRAIC LAWS

There are various algebraic laws which can be applied on set of statements to improve the code generated from a DAG.

Algebraic Laws :-

1. **Commutative law**

   $a + b = b + a$

2. **Associative Law**

   $a + (b + c) = (a + b) + c$

For *e.g.*,

Consider the statements

$a = b + c$

$d = c + m + b$

The DAG for these statements without applying any commutative & associative law will be.

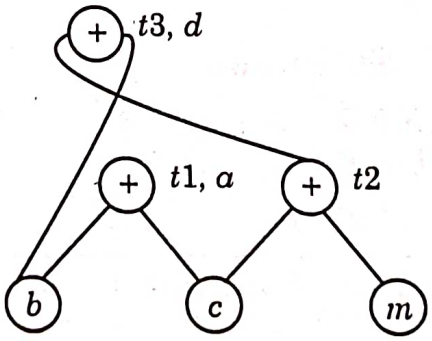


Fig. 11.7. DAG before applying commutative & associative law

On applying commutative & associative law

We can write $d = c + m + b$ as $d = b + c + m$

$\therefore$ The corresponding DAG will become

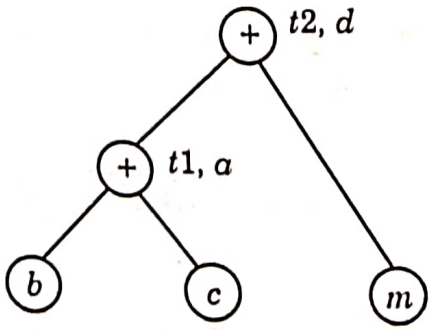


Fig. 11.8. DAG after commutative & associative law

∴ On using algebraic laws, code will be optimized and will use only 2 temporary variable ($t1$, $t2$) instead of 3 temporary variables ($t1$, $t2$, $t3$).

Limitations :-

→ Algebraic laws can change the value computed by an expression and can give over flow.

For *e.g.* If a, b, c are unsigned integer type lying in range.

$(a - c) + b$ can be written as $(a + b) - c$ using associative & commutative law.

But $(a + b) - c$ can produce an overflow error *i.e.* $a + b$ can give out of range value.

11.9 GLOBAL DATA FLOW ANALYSIS

Date Flow Analysis → In Data Flow Analysis, we analyze the flow of data from one block to another block of a program. It determines definition and use of data in a program.

The whole program is divided into blocks. Blocks are joined to make a flow graph. Finding all such informations can help to apply optimization called Data Flow Analysis.

Terms used in Global Data Flow Analysis :-

→ **Definition** of identifier a means an assignment to a or reading of value for a.

→ Use of identifier b means any occurrence of b as an operand.
For *e.g.* statement $a = b + c$ is said to **define** a and to **use** b **and** c.

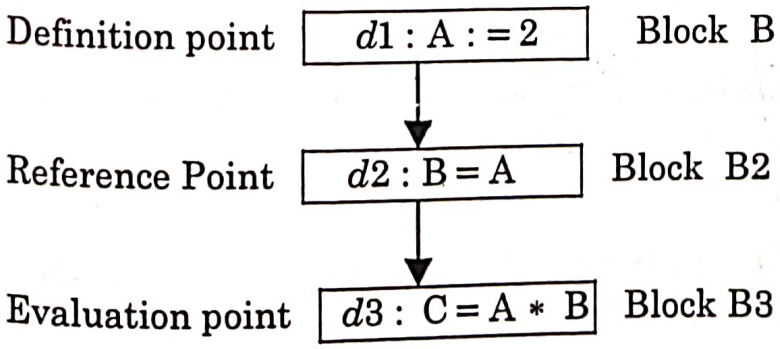
→ **Use definition (ud chaining) –** If identifier a is used at point p, then determining where the value of a was defined is called ud chaining.

→ **Point –** It is a position before or after any statement.

(*a*) **Definition Point –** A program point containing definition.

(*b*) **Reference point –** A program point at which a reference to a data item is made.

(*c*) **Evaluation point –** A program point at which some evaluating expression is given.



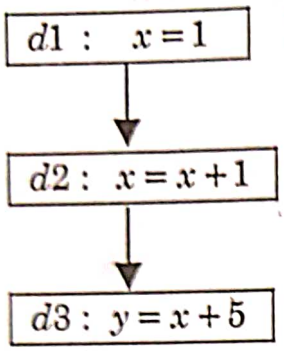
11.9.1 Reaching Definitions

→ Reaching Definition refers to determining the definitions that can reach a given point in a program.

→ A definition of variable x reaches at point p if there is path from that definition to p, such that no other definition of x appears on the path.

For example :– In given figure,,

Definition $d1$ is said to be a reaching definition for Block B2. But definition $d1$ is not a reaching definition in Block B3, because it is killed by definition $d2$ in Block B2. ($\because$ x is redefined in Block B2, its previous value will be lost)



11.9.2 Steps to Compute Reaching Definitions

1. Assign unique number to each definition *i.e.* $d1$, $d2$ ..... dn.
2. Compute gen [B]

   gen [B] – Set of generated definitions in Block B which reaches end of the Block B.
3. Compute kill [B]

   kill [B] – Set of definitions outside Block B that define same variables as are defined in Block B.
4. Compute in [B]

   in [B] – Set of definitions reaching the point just before first statement of Block B.
5. Compute out [B]

   out [B] – Set of definitions reaching the point just after last statement of B.

11.9.3 Data Flow Equations

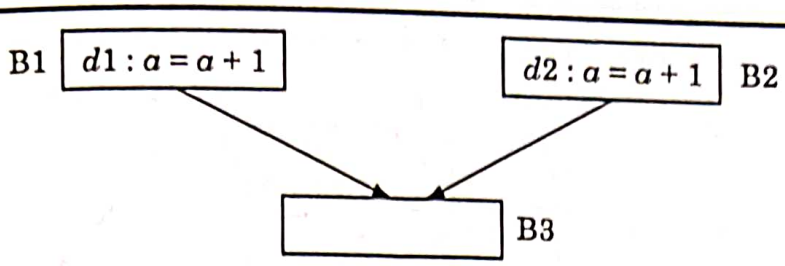
These equation represent the relationship between in's and out's of Block B.

(*a*) out [B] = gen [B] $\cup$ in [B] – kill [B]

(*b*) in [B] = $\bigcup_{P}$ out [P]

where P = Predecessor of B

For *e.g.*



Here, in [B3] = out [B1] ∪ out [B2]

where B1, B2 are Predecessor of B3.

11.9.4 Algorithm to compute Reaching Definitions

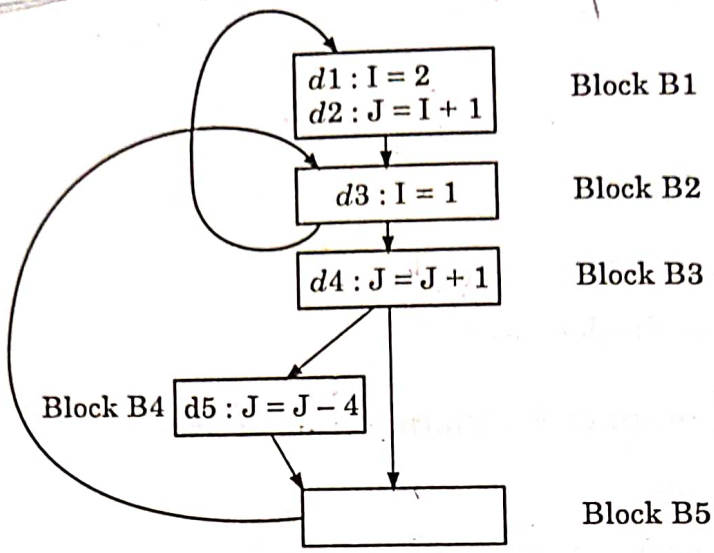
Input :- Flow graph

Output :- in [B] and out [B] for each Block B.

Method :-

1. Compute gen [B] and kill [B] for each Block B of flow graph.
2. for each Block B.
   {
3. in [B] = ϕ
4. out [B] = gen [B]
   }
5. change = true ;
6. while (change)
   {
7. change = false ;
8. for each Block B
   {
9. newin = $\bigcup_{P}$ out [P], where P is Predecessor of Block B.
10. if (newin ≠ in [B])
11. {
    change = true
12. in [B] = newin
13. out [B] = gen [B] ∪ in [B] – kill [B]
    }
   }
   }

Example 14. Calculate in [B] and out [B] for given flow graph.



Ans. We can represent gen [B] and kill [B] by bit vectors. *i.e.* we can represent each definition (*di*) by 0's and 1's.

Applying line 1 of algorithm to compute gen [B] and kill [B]

Block	gen [B]	kill [B]
B1	gen [B1] = {*d*1, *d*2 } = 11000	kill [B1] = {*d*3, *d*4, *d*5 } = 00111
B2	gen [B2] = {*d*3} = 00100	kill [B2] = {*d*1} = 10000
B3	gen [B3] = {*d*4} = 00010	kill [B3] = {*d*2, *d*5} = 01001
B4	gen [B4] = {*d*5} = 00001	kill [B4] = {*d*2, *d*4} = 01010
B5	gen [B5] = ϕ = 00000	kill [B5] = ϕ = 00000

Applying Line (3) and (4) of algorithm

Initialize in [B] = ϕ and out [B] = gen [B] for all Blocks B, ϕ will be represented as 00000.

Initialization		
Block	**in [B]**	**out [B]**
B1	00000	11000
B2	00000	00100
B3	00000	00010
B4	00000	00001
B5	00000	00000

Applying lines (8) to (13) of algorithm to get Pass 1 on each Block

Pass 1

Block B1

∵ Predecessor of B1 is B2.

∴ in [B1] = out [B2] = 00100

out [B1] = gen [B1] ∪ (in [B1] – kill [B1])

= 11000 + (00100 – 00111) = 11000 + (00000)

∴ out [B1] = 11000

Block B2

∵ B1 and B5 are predecessor to B2.

∴ in [B2] = out [B1] + out [B5] = 11000 + 00000

∴ in [B2] = 11000

out [B2] = gen [B2] ∪ (in [B2] – kill [B2]) = 00100 + (11000 – 10000)

∴ out [B2] = 00100 + (01000) = 01100

Block B3

∵ B2 is predecessor of B3

∴ in [B3] = out [B2]

∴ in [B3] = 01100

out [B3] = gen [B3] ∪ (in [B3] – kill [B3]) = 00010 + (01100 – 01001)

= 00010 + (00100)

∴ out [B3] = 00110

Block B4

∵ B3 is Predecessor of B4

∴ in [B4] = out [B3]

∴ in [B4] = 00110

out [B4] = gen [B4] ∪ (in [B4] – kill [B4]) = 00001 + (00110 – 01010)

= 00001 + (00100)

∴ out [B4] = 00101

Block B5

in [B5] = out [B3] + out [B4] = 00110 + 00101

∴ in [B5] = 00111

out [B5] = gen [B5] ∪ (in [B5] – kill [B5]) = 00000 + (00111 – 0000) = 00111

Pass 1		
	in [B]	**out [B]**
B1	00100	11000
B2	11000	01100
B3	01100	00110
B4	00110	00101
B5	00111	00111

Applying next iteration *i.e.* considering that while loop will be executed more than one time. So Pass 2 will be computed next.

Pass 2

Block B1

in [B1] = out [B2] = 01100

out [B1] = gen [B1] ∪ (in [B1] – kill [B1])

= 11000 + (01100 – 00111)

∴ out [B1] = 11000

Block B2

in [B2] = out [B1] + out [B5]

∴ in [B2] = 11000 + 00111 = 11111

out [B2] = gen [B2] ∪ (in [B2] – kill [B2]) 00100 + (11111 – 10000)

∴ out [B2] = 01111

Block B3

∵ B2 is predecessor to B3

∴ in [B3] = out [B2]

∴ in [B3] = 01111

out [B3] = gen [B3] ∪ (in [B3] – kill [B3]) = 00010 + (01111 – 01001)

∴ out [B3] = 00110

Block B4

∵ B3 is Predecessor of B4

∴ in [B4] = out [B3]

∴ in [B4] = 00110

out [B4] = gen [B4] ∪ (in [B4] – kill [B4])

∴ out [B4] = 00001 + (00110 – 01010) = 00101

Block B5

int [B5] = out [B3] + out [B4]

∴ in [B5] = 00110 + 00101 = 00111

out [B5] = gen [B5] ∪ (in [B5] – kill [B5])

∴ out [B5] = 00000 + (00111 – 00000) = 00111

Pass 2		
	in [B]	**out [B]**
B1	01100	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

Applying next iteration *i.e.* while loop of algo, *i.e.* Pass 3.

Pass 3

Block B1

in [B1] = out [B2] = 01111

out [B1] = gen [B1] ∪ (in [B1] – kill [B1])

∴ out [B1] = 11000 + (01111 – 00111) = 11000

Block B2

in [B2] = out [B1] + out [B5]

∴ in [B2] = 11000 + 00111 = 11111

out [B2] = gen [B2] ∪ (in [B2] – kill [B2])

∴ out [B2] = 00100 + (11111 – 10000) = 01111

Block B3

∴ in [B3] = out [B2] = 01111

out [B3] = gen [B3] ∪ (in [B3] – kill [B3])

∴ out [B3] = 00010 + (01111 – 01001) = 00110

Block B4

in [B4] = out [B3] = 00110

out [B4] = gen [B4] ∪ (in [B4] – kill [B4])

∴ out [B4] = 00001 + (00110 – 01010) = 00101

Block B5

in [B5] = out [B3] + out [B4]

∴ in [B5] = 00110 + 00101 = 00111

out [B5] = gen [B5] ∪ (in [B5] – kill [B5])

∴ out [B5] = 00000 + (00111 – 00000) = 00111

Pass 3		
	in [B]	**out [B]**
B1	01111	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

Applying next iteration *i.e.* Pass 4, will give same in and out values.

∴ Pass 3 is final answer.

Example 15. Calculate use definition (ud chaining) for variable I and J in previous example.

Ans. In Block B2

$d1$: I = 2

$d2$: J = I + 1

$\because$ $d2$ uses I, which follows definition of I in $d1$.

$\therefore$ ud-chain for I in $d2 = \{d1\}$

In Block B3, d4 is using J. But there is no definition of J before it in same Block.

$\therefore$ We have to find in [B3].

$\therefore$ in [B3] = { $d2$, $d3$, $d4$, $d5$ }

But, out of { $d2$, $d3$, $d4$, $d5$ }, $d3$ has definition of I *i.e.* I = 1.

$\therefore$ Ud-chain for J in $d4 = \{d2, d4, d5\}$

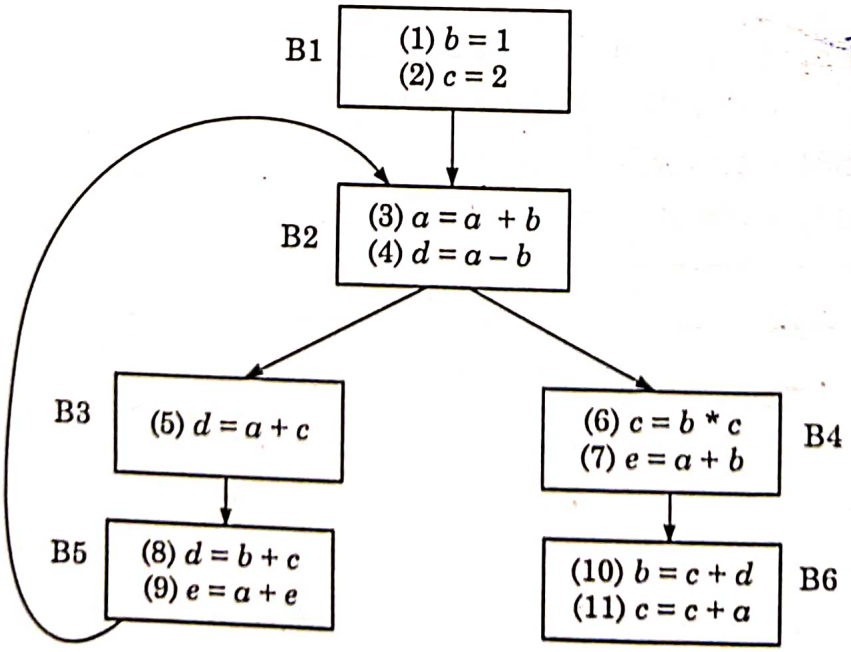
In Block B4, $d5$ is also using J which is not preceded by any definition of J.

$\because$ in [B4] = { $d3$, $d4$ }

But $d3$ is definition of I.

$\therefore$ Ud-chain for J in $d5 = \{d4\}$

Example 16. Compute gen and kill for following blocks in flow graph.



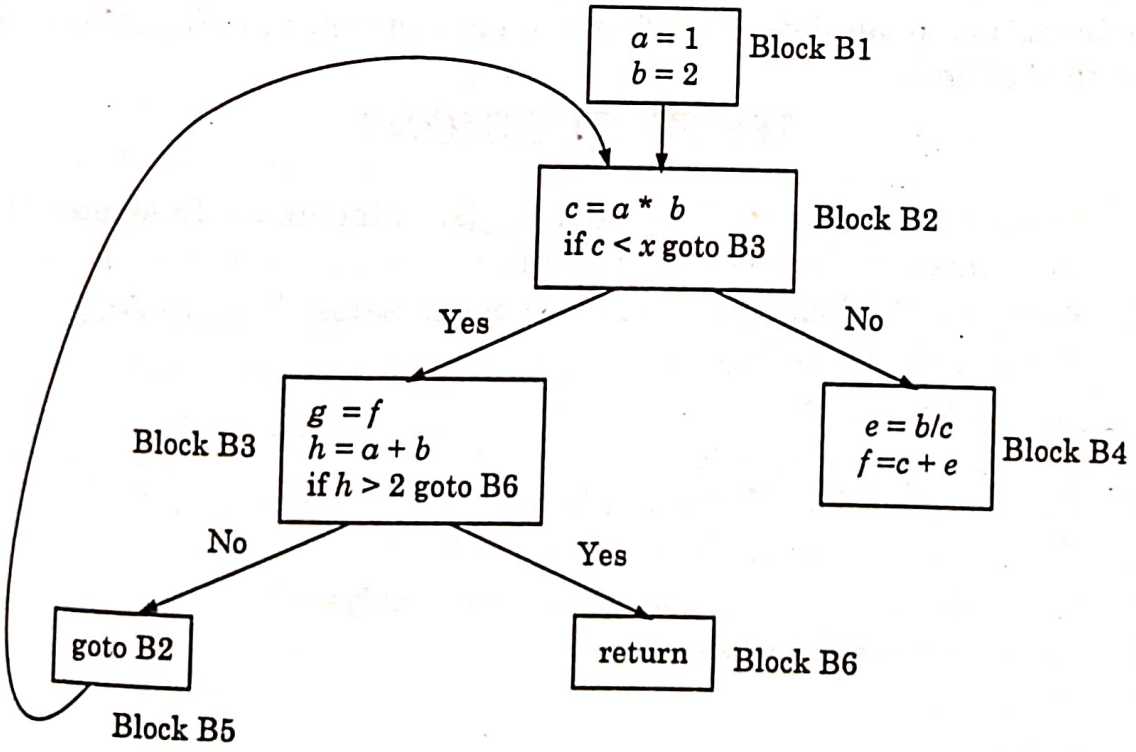
Ans.

Block	gen	kill
B1	{ 1, 2 }	{6, 10, 11}
B2	{ 3, 4 }	{ 5, 8 }
B3	{5}	{4, 8}
B4	{6, 7}	{2, 9, 11}
B5	{8, 9}	{4, 5, 7}
B6	{10, 11}	{1, 2, 6}

Example 17. Draw the flow graph for following three-address code statements :-

```
    a = 1
    b = 2
L1: c = a * b
    if c < x goto L2
    e = b/c
    f = c + e
L2: g = f
    h = a + b
    If h > 2 goto L3
    goto L1
L3: return
```

Ans.



KEY POINTS TO REMEMBER

Code Optimization – It improves execution efficiency of the object code.

⇒ **Sources of Optimization**

(*a*) Local optimization
(*b*) Loop optimization
(*c*) Data Flow Analysis
(*d*) Algorithmic optimization

⇒ **Basic Blocks** – A sequence of statements in which flow of control enters at beginning and when entered are executed in a sequence.

⇒ **Flow graph** – A directed graph which shows relationships between basic blocks.

⇒ **Loop optimization**

(*a*) Code Motion
(*b*) Elimination of induction variables
(*c*) Reduction in Strength

⇒ **DAG** – Directed Acyclic graph.

⇒ **Value Number Method** → Each node of DAG is assigned a number to know whether a node with particular childrens exist.

⇒ **Global Data Flow Analysis** → It performs optimization by determining information about definition & use of data in program. It performs analysis on flow of data.

REVIEW QUESTIONS

Q.1. Explain machine Dependent & Machine Independent Code Optimization Techniques in detail.
Q.2. What are the Principal sources of optimization ?
Q.3. Explain the following terms : –
(*a*) Code Motion
(*b*) Induction Variable
Q.4. What are DAGS ? Write an algorithm for constructing DAG.
Q.5. What are various applications of DAGS ?
Q.6. Write short note on Global data Flow Analysis ?
Q.7. Construct DAG for basic block.

D = B * C
E = A + B
B = B * C
A = E – D