07/10/14

UNIT-7

Data Flow Analysis

- Flow graph
- Data flow equation
- Global Optimization.
- Redundant Sub expression elimination
- Induction variable elimination
- Live variable Analysis
- Copy propogation.

7M

Flow graph :-

A flow graph is a directed graph in which the flow control information is added to the basic block.

→ The nodes to the flow graph are represented by basic blocks.

→ The basic block whose leader is the first statement is called initial block.

→ There is a directed edge from block B_1 to block B_2. If B_2 immediately follows B_1. In the given sequence. you can say that B_1 is a predesessor of B_2

For example:- Consider 3-address code for

```
Begin
PROD:=0
I:=1
do
  begin
PROD:= PROD+A[I]*B[I];
  I:= I+1
  END
while I<=20
        END.
```

Sol The 3-address code is

1) PROD := 0
2) i := 1
3) loop : = t1 := 4*i

4) $t_2 := A[t_1]$

5) $t_3 := 4 * i$

6) $t_4 := B[t_3]$

7) $t_5 := t_2 * t_4$

8) $t_6 := PROD + t_5$

9) $PROD := t_6$

10) $t_7 := i + 1$

11) $i := t_7$

12) $i <= 20$ goto loop 3.

The flow graph for the above code can be drawn as follows

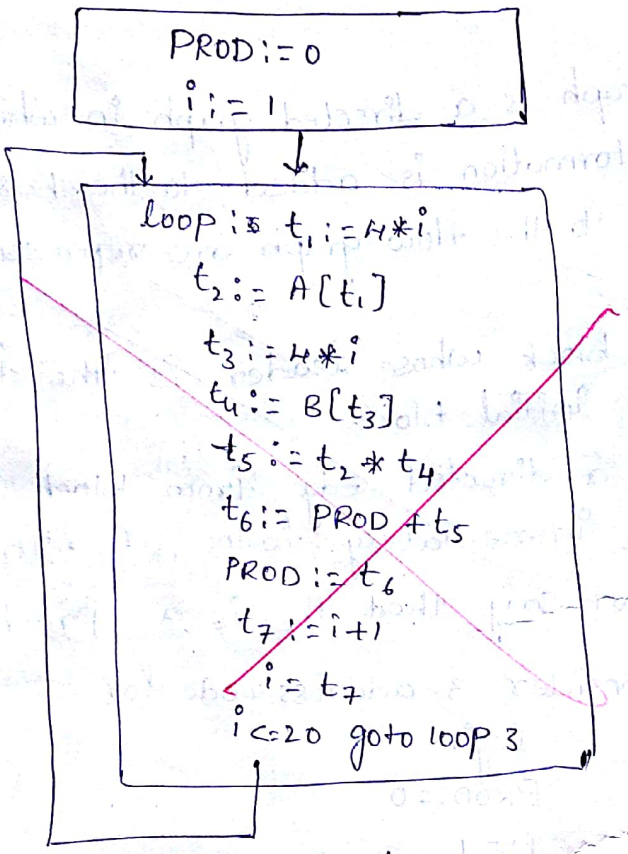


fig: flow graph

In the above flow graph, we have 2 basic blocks B_1 & B_2

Here B_1, Basic block is a Initial block & it transmit the data to the basic block B_2.

→ Loops in flow graph:-

Let us get introduced with some common technologies being used for loop in flow graph.

1) Dominator:

In a flow graph a node 'd' dominates 'n' if every path to node 'n' from initial node goes through 'd'. This can be denoted as d domn.

Every initial node dominates all the remaining nodes in the flow graph. Similarly every node dominates its self.

Eg:-

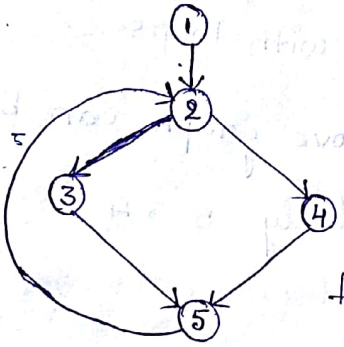


fig: flow graph

In above flow graph, node 1 is initial node & it dominates every node as it is a initial node.

→ Node 2 dominates 3, 4, & 5 as there exist only one path from initial node to node 2 which is going through 2 (1-2-3) similarly for node 4 only path exist is 1-2-4 and for node 5 the only path existing 1-2-3-5 (or) 1-2-3-4 which is going through the node 2.

→ Node 3 dominates itself similarly node 4 dominates itself the reason is that for going to remaining node 5. node 5 dominates no node.

2) Natural loops:-

Loop in a flow graph can be denoted by $n \to d$ such that "d domn" these edges are called back edges and for a loop there can be more than one back edges. If there is $p \to q$ then q is head & p is tail & head dominates tail.

Eg:-

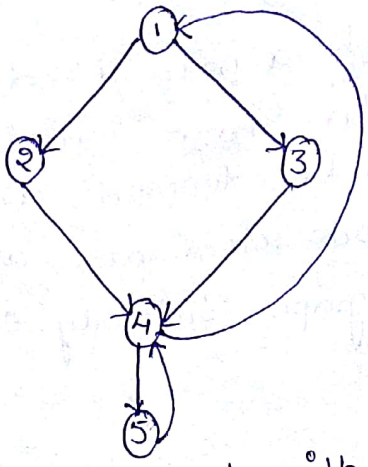


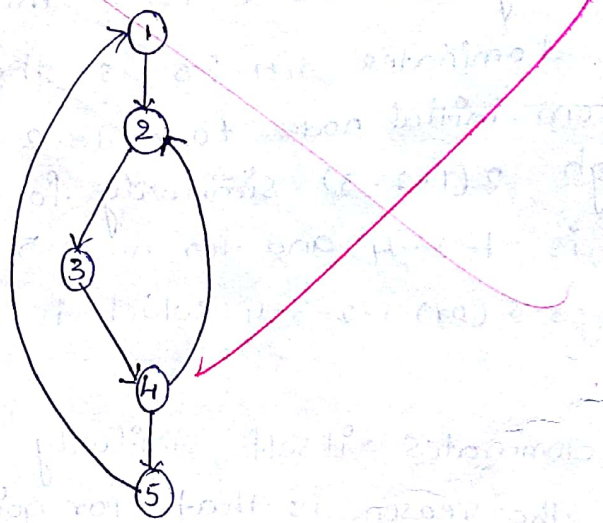
fig: flow graph with loops

The loops in the above graph can be denoted by '4 → 1' i.e 1 dom 4. Similarly 5 → 4 i.e., 4 dom 5

3) Inner loops :-

The inner loop is a loop that contains no other loop.

Eg:-



Here the inner loop is 4 → 2 i.e., edge given by 2-3-4.

4) Reducible flow graph :-

The reducible flow graph is a flow graph in which there are two types of edges i.e., forward edges & backward edges. These edges have following properties.

1. The forward edge from an acyclic graph
2. The backward edges are such edges whose head dominates their tail.

Eg:-

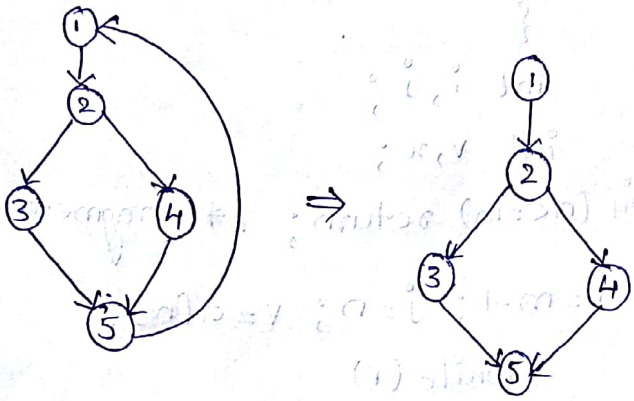


Fig: flow graph

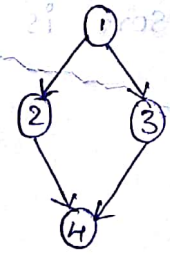
The above flow graph is reducible. we can reduce this graph by removing the back edge from 3 to 2. Similarly by removing the back edge from 5 to 1.

5) Non Reducible flow graph:-

A non reducible flow graph is a flow graph is a flow graph in which

1. There are no back edges
2. Forward edges may produce cycle in the graph.

Eg:-



IAM*

Eg:- Draw the flow graph by using quick sort(c co

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

Sol The 3-address code for quick sort is as follows.

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. t_3 < v goto step 5
9. j = j-1
10. t_4 := 4*j
11. t_5 := a[t_4]

12. $t_5 > v$ goto step 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. ~~$a(t_4) := t_9$~~

20. $t_{10} := 4 * j$

21. $a[t_{10}] > v$

22. goto step 5

23. $t_{11} := 4 * i$

24. $x := a[t_{11}]$

25. $t_{12} := 4 * i$

26. $t_{13} := 4 * n$

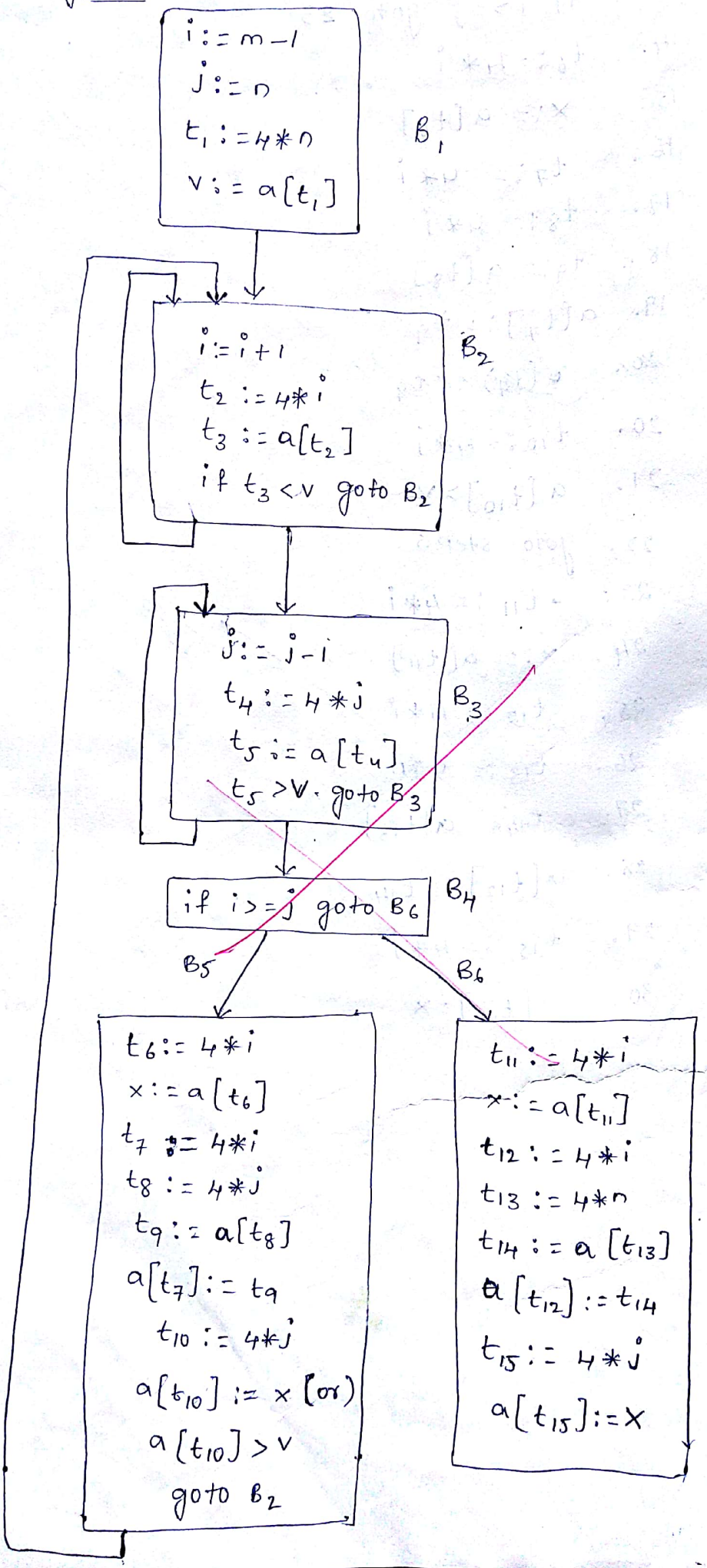
27. $t_{14} := a[t_{13}]$

28. $a[t_{12}] := t_{14}$

29. $t_{15} := 4 * j$

30. $a[t_{15}] = x$.

Flow graph :-



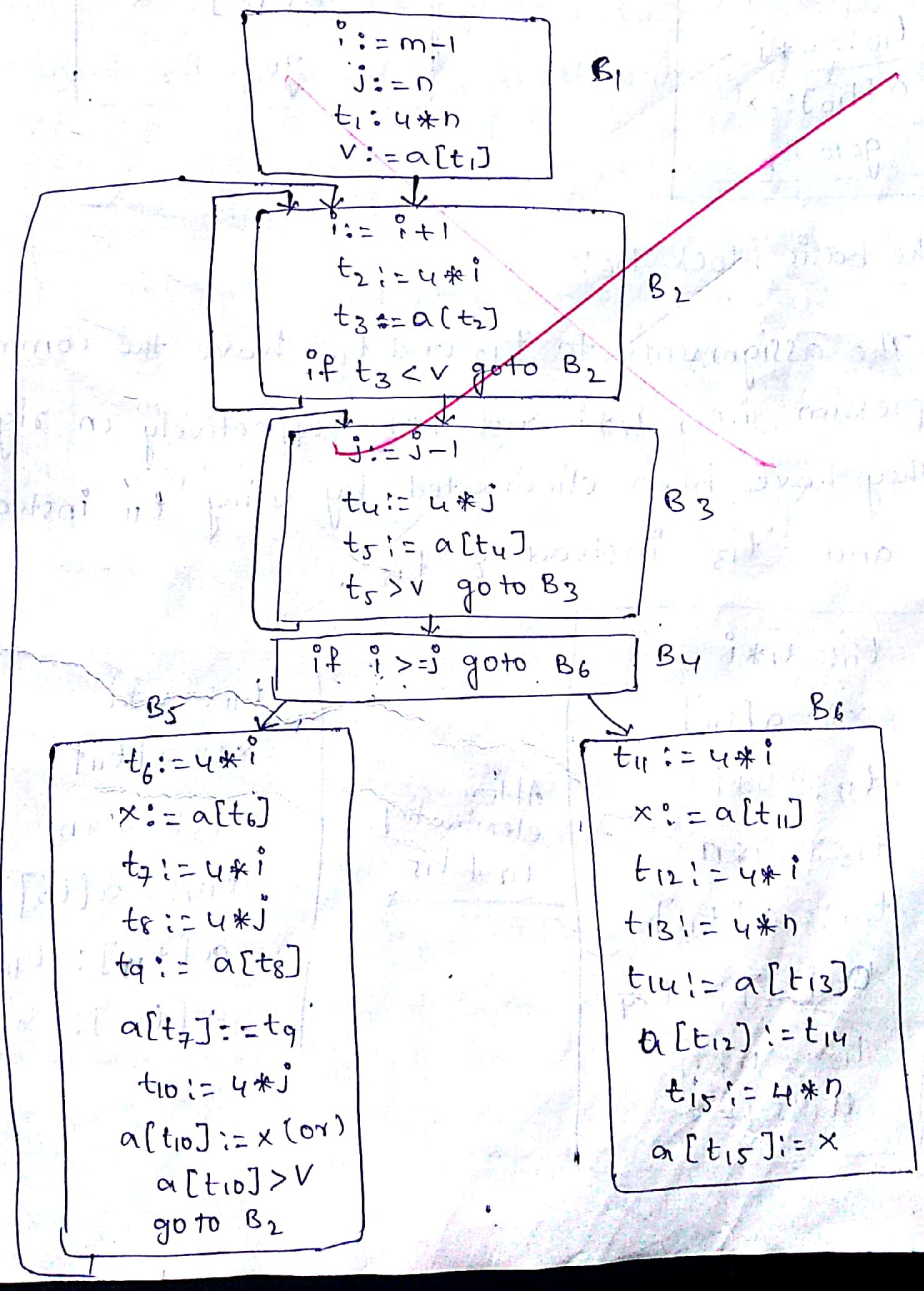
Local optimization (or) Function preserving Transformations:-

Here we mainly concentrate on

1) redundant common sub expression elimination.
2) Copy Propogation.
3) Dead code elimination.
4) Constant folding

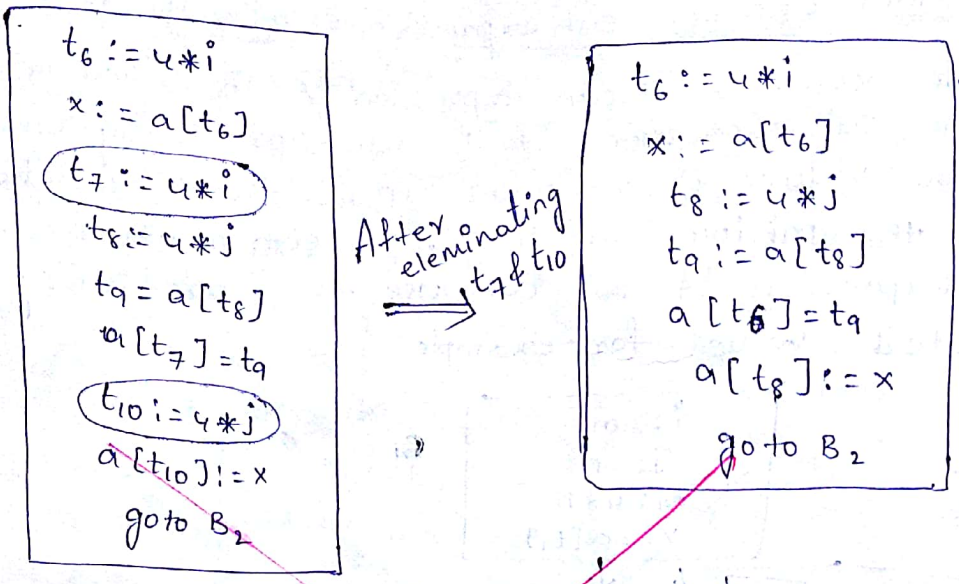
Redundant common sub expression elimination:-

An occurence of an expression 'E' is called a common sub expression if 'E' was previously computed and the values of variables in 'E' have not changed since the previous computition. We can avoid recomputing the expression if we can use the previously computed value for example



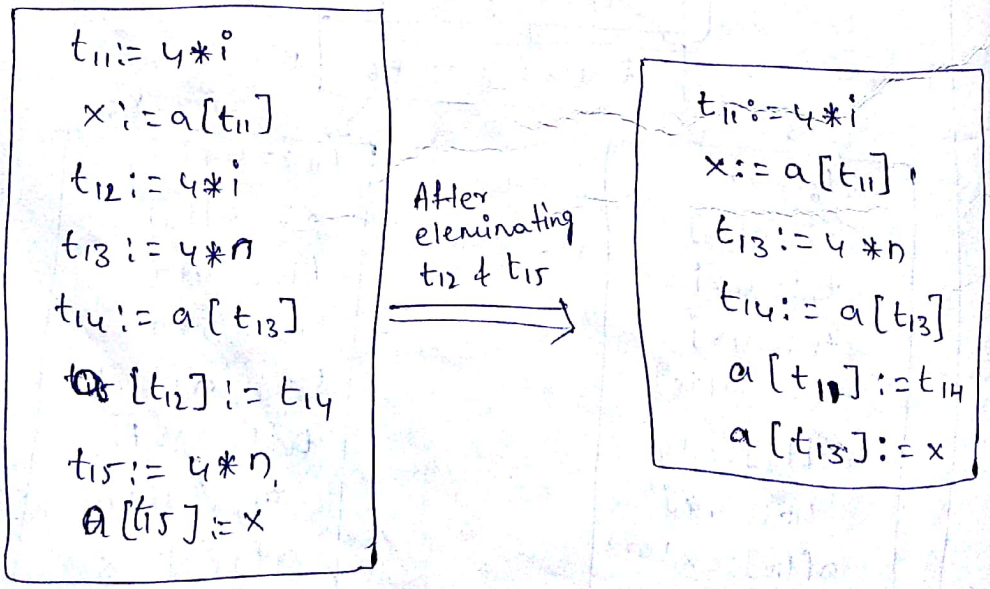
From the basic block B_5 :-

The assignment to t_7 and t_{10} have the common sub expression i.e., $4*i$ and $4*j$ respectively on right side. They have been eliminated by using 't_6' instead of 't_7' and 't_8' instead of 't_{10}'.



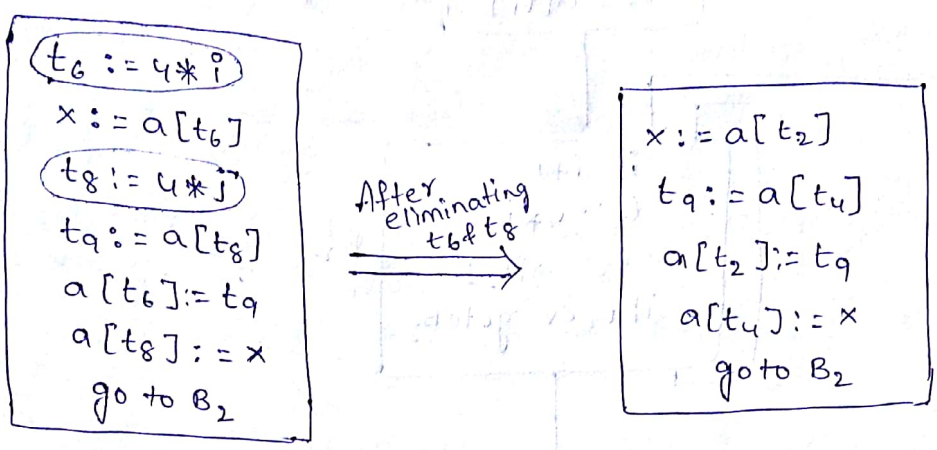
From the basic block B_6 :-

The assignments to t_{12} and t_{15} have the common sub expression i.e., $4*i$ and $4*n$ respectively on right side. They have been eliminated by using 't_{11}' instead of 't_{12}' and 't_{13}' instead of 't_{15}'.

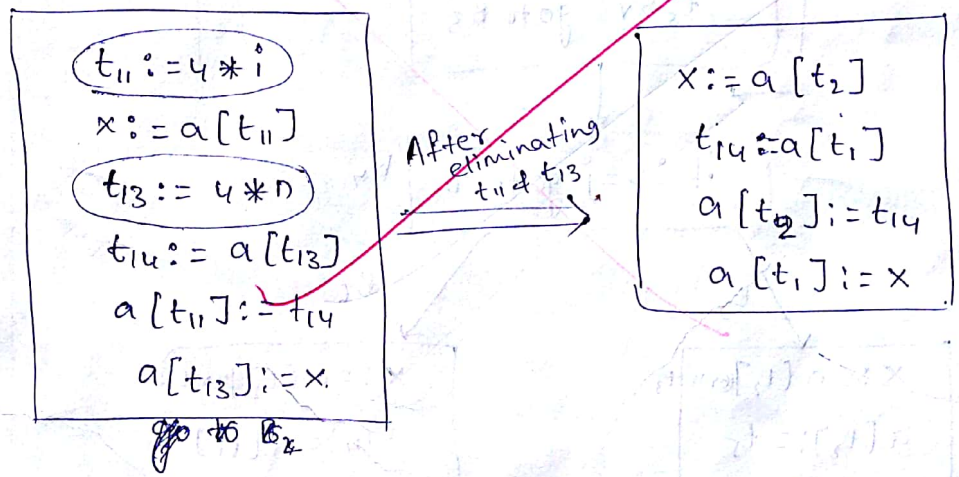


→ After local common sub expressions are eliminated; B_5 still evaluates $4*i$ & $4*j$. They have eliminated by using t_2 instead of t_6 & t_4 instead of t_8.

→ After eleminating t_6 and t_8 the basic block B_5 can be written as

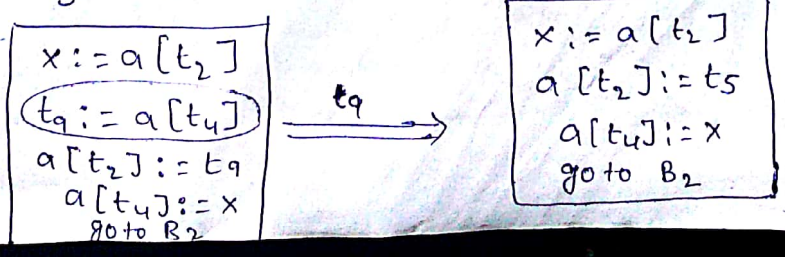


The common sub expression t_{11} & t_{13} have been eliminated by using t_2 instead of t_{11} & t_1 instead of t_{13}

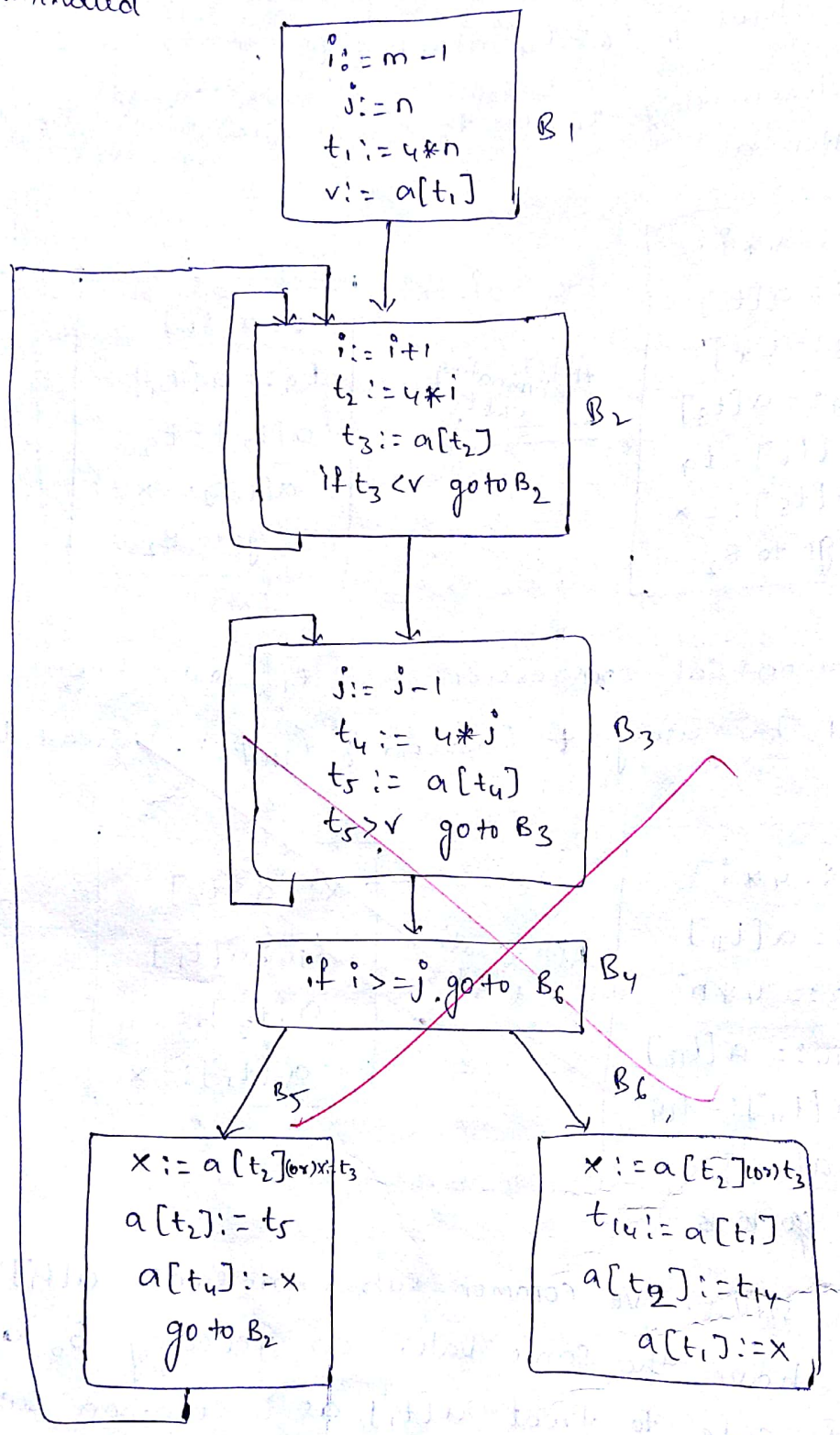


B_5 & B_6 still have common sub expressions $a[t_1]$ may not have the same value on reaching B_6 and it is not safe to treat $a[t_1]$ as a common sub expression.

→ The common sub expression t_9 has been eliminated by using t_5 instead t_9



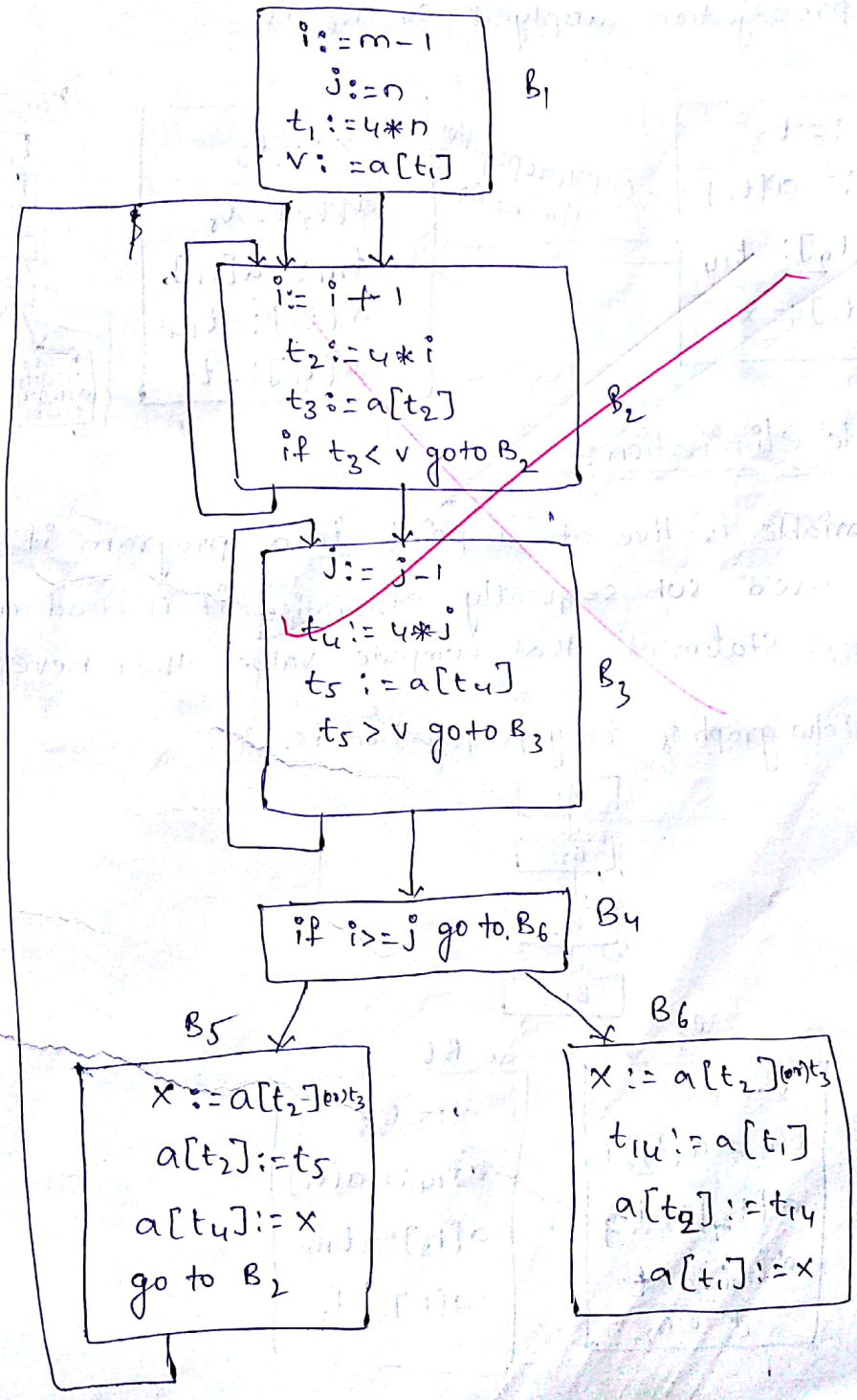
The flow graph after common sub expressions are eliminated



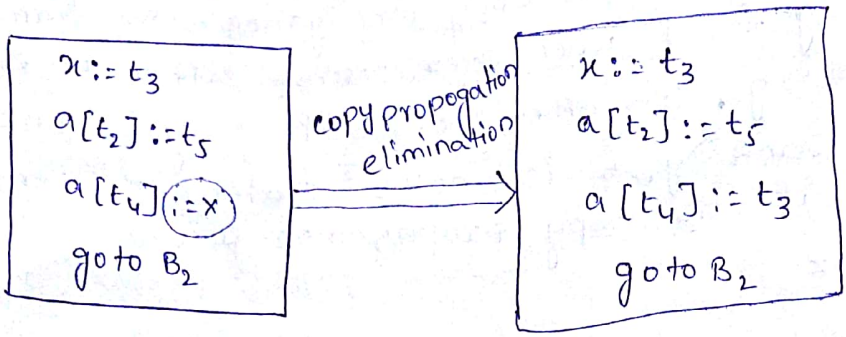
Copy propogation :-

The form $f := g$ called copy statements (or) copies the idea behind the copy propagation transformation is to use g for f when ever possible after the copy statement $f := g$. In other words if $x := y$ is one statement and $y := z$ is another statement then if and only if the copy propagation can be written as $x := z$

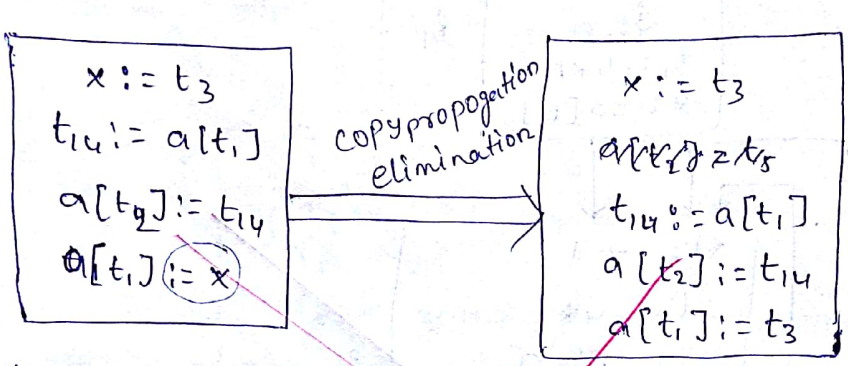
Eg:- Consider the flow graph after elimination of common sub expression is



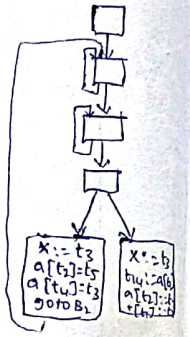
The assigment $x := t_3$ in block B_5 is a copy statement and copy propogation applyed to B_5 is



The assigment $x := t_3$ in block B_6 is a copy statement and copy propogation applyed to B_6 is



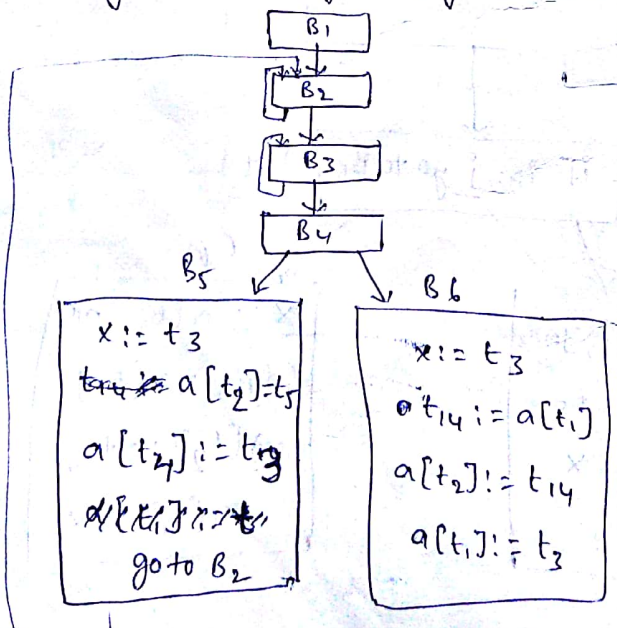
flow graph



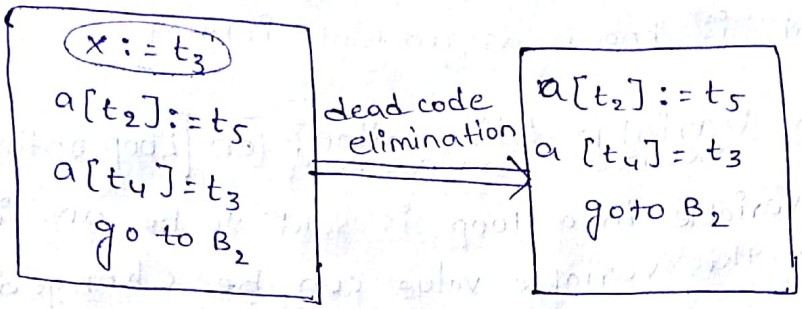
Dead Code elimination :-

A variable is live at a point in a program if its value is used sub sequently otherwise it is dead at the point. i.e., Statements that compute values that never get used

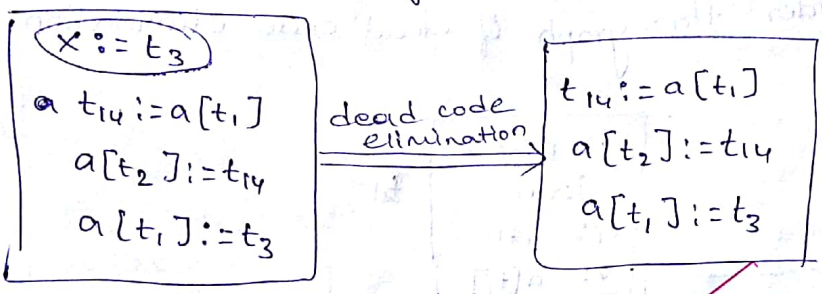
Eg:- Folw graph of copy propogation is.



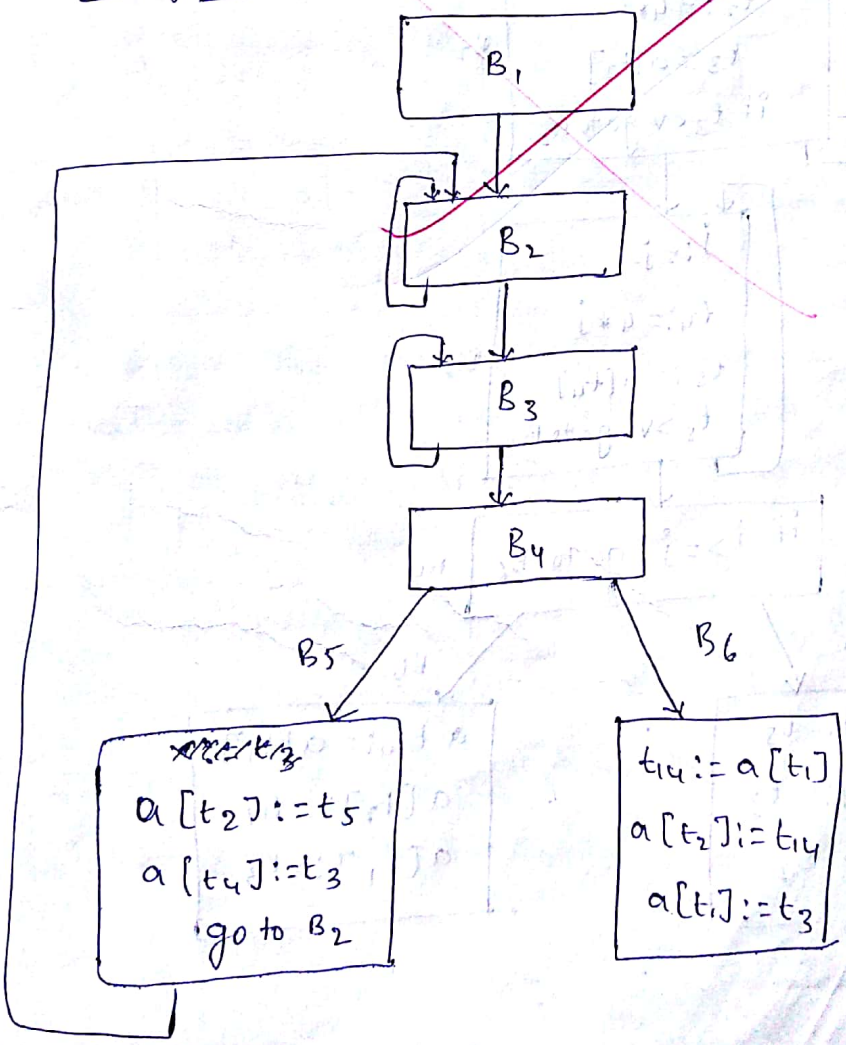
The assigments $x := t_3$ in Block B_5 is a dead code. Dead code elimination is applyed to B_5 is



The assigments $x := t_3$ in Block B_6 is a dead code. Dead code elimination is applyed to B_6 is



Flow graph:-



Constant folding :-

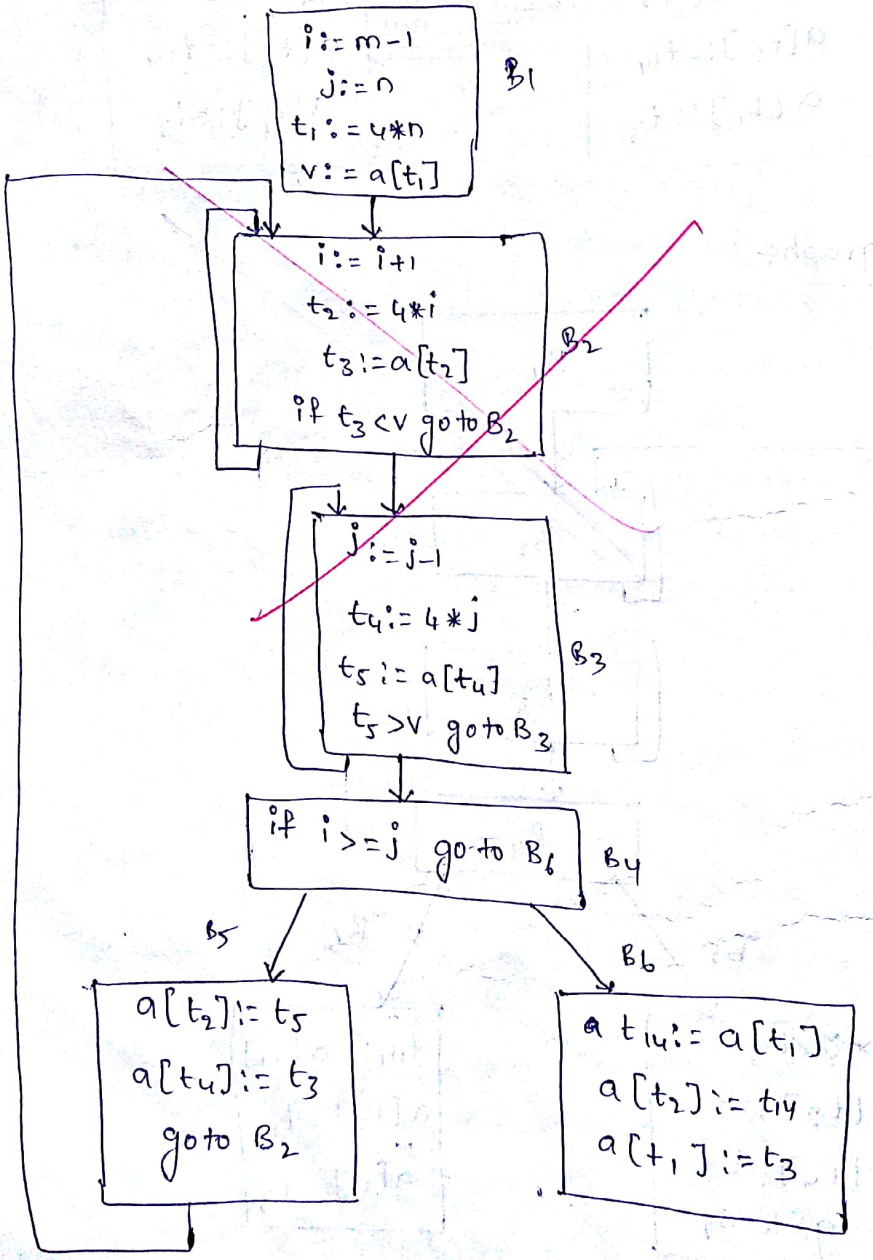
The submition of values for names whose values are constant is known as constant folding.

***imp

Induction Variable Elimination :- (or) [Loop optimization]

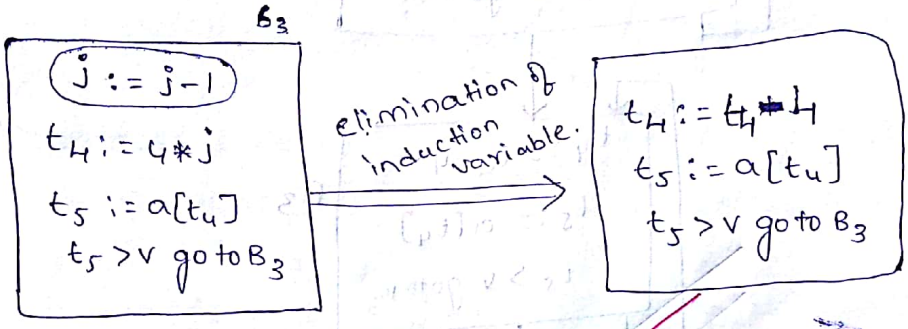
A variable in a loop is said to be an induction variable, if this variable value can be changed by using some constant. Here, changing of the variable is may be increment (or) decrement.

Eg:- Consider flow graph of dead code elimination. is

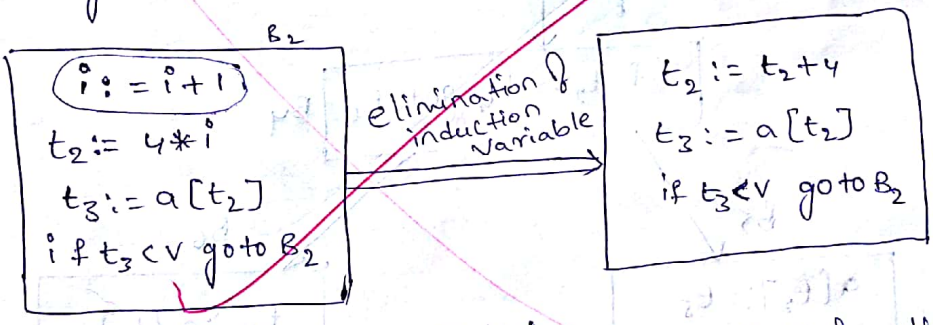


The values of j and t_4, every time the value of 'j' decreases by one, that of t_4 decreases by 4 because of 4*j is assigned to t_4. Therefore replace the assignment $t_4 = 4*j$ by $t_4 = t_4 - 4$.

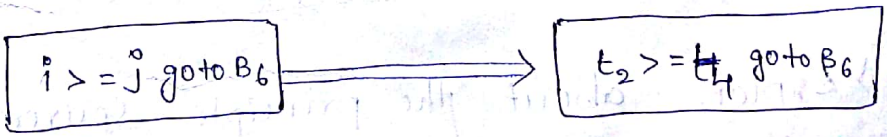
The only problem is that t_4 does not have a value when we enter block B_3 for the first time since we must maintain the relationship $t_4 = 4*j$ on entry to the block B_3, we place an initialization of t_4 at the end of block where 'j' it self is initialized



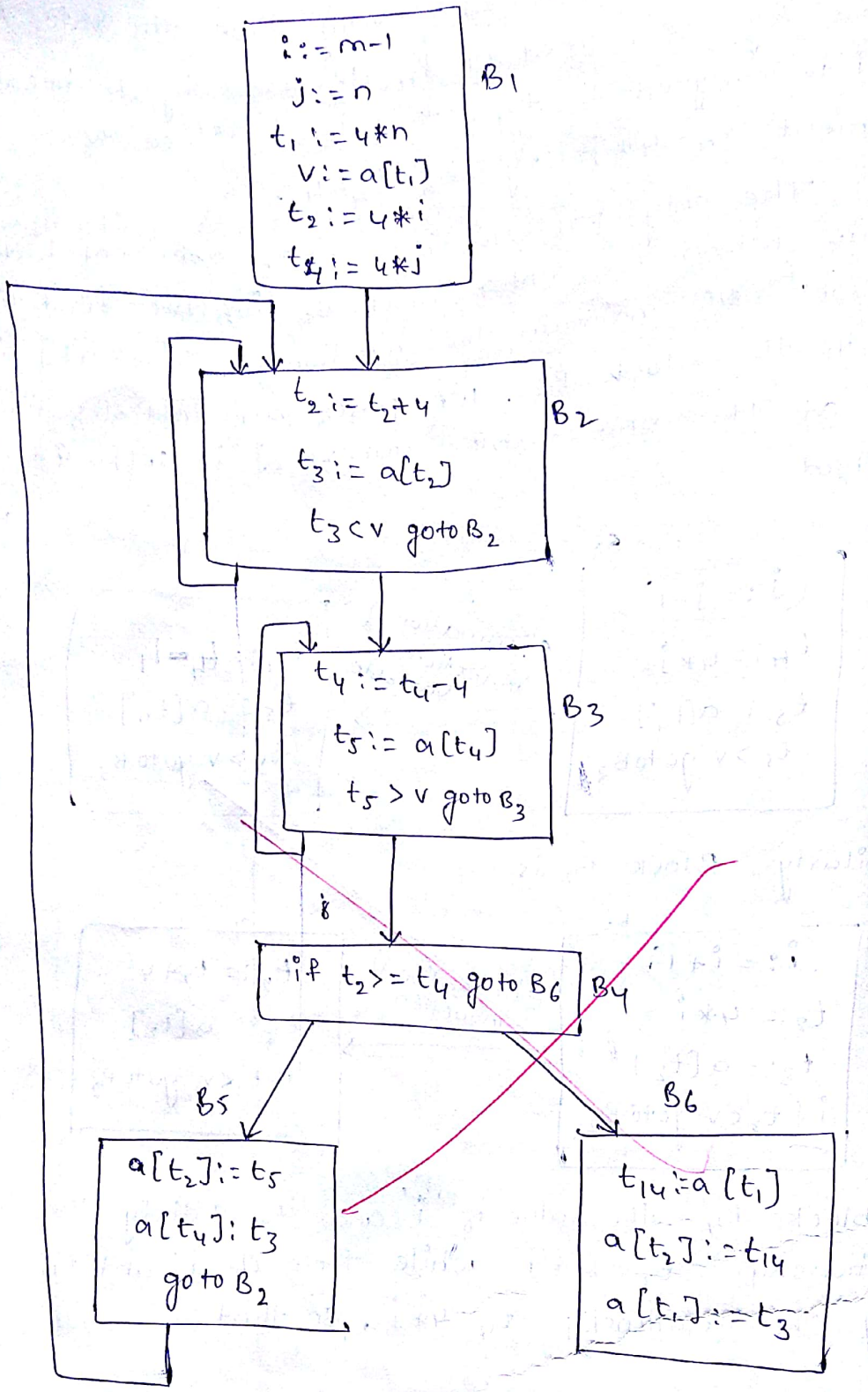
Similarly Block B_2 is



In block B_4 the value of 'i' and t_2 satisfy the relationship $t_2 = 4*i$, while those of 'j' and t_4 satisfy the relationship $t_4 = 4*j$, so that



∴ The required flow graph of induction variable elimination is.



Note :- 1) Explain about the principle sources of optimization (or)

2) Explain about local optimization & loop optimization

Write 1) 3-address code

2) Basic block

3) flow graph

4) Elemination of common sub-expression of

```
begin
PROD = 0
I := 1
do
  begin
PROD := PROD + A[i] * B[i]
  I := I + 1
  END
while I <= 20
  END.
```

Sol 1) 3-address code

1) PROD := 0
2) i := 1
3) loop: t_1 := 4 * i
4) t_2 := A[t_1]
5) t_3 := 4 * i
6) t_4 := B[t_3]
7) t_5 := t_2 * t_4
8) t_6 := PROD + t_5
9) PROD := t_6
10) t_7 := i + 1
11) i := t_7
12) i <= 20 goto loop (3)

2)

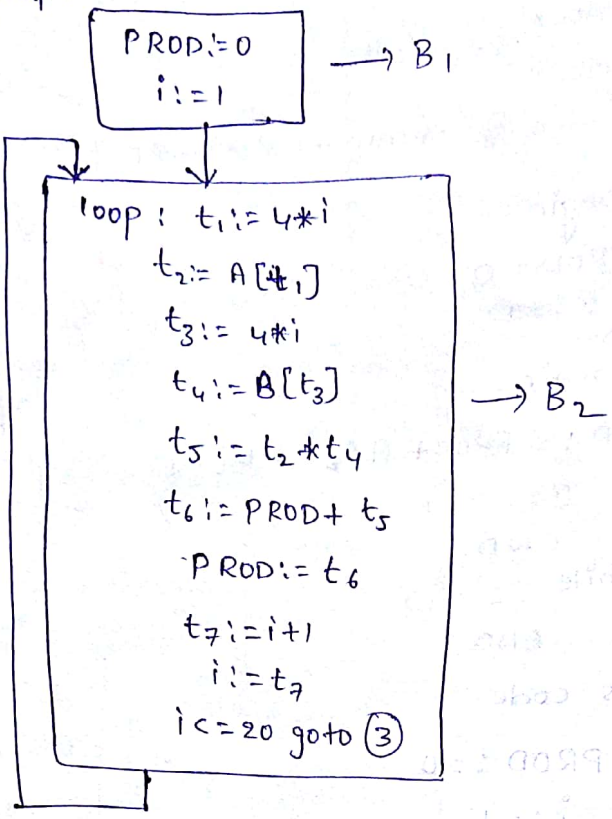
B_1

```
PROD := 0
i := 0
```

B_2

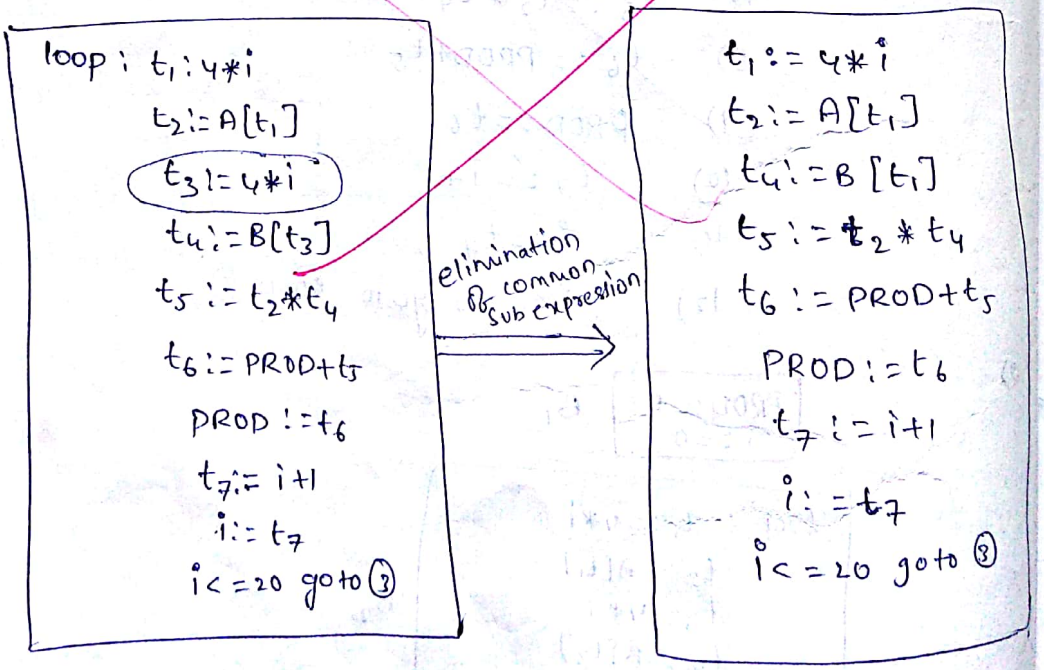
```
loop : t1 := 4*i
       t2 := A[t1]
       t3 := 4*i
       t4 := B[t3]
       t5 := t2 * t4
       t6 := PROD + t5
       PROD := t6
       t7 := i+1
       i := t7
i <= 20  go to loop (3)
```

3) Flow graph



4) elimination common sub expressions

From basic block B_2 t_1 and t_3 are common terms i.e., $t_1 = t_3$ so that we can eliminate t_3 i.e., $t_3 = 4*i$, we get



Data flow equation :-

Data flow equations are the equations representing the expressions that are appearing in the flow graph. This equations are useful for computing live variable and reaching definition.

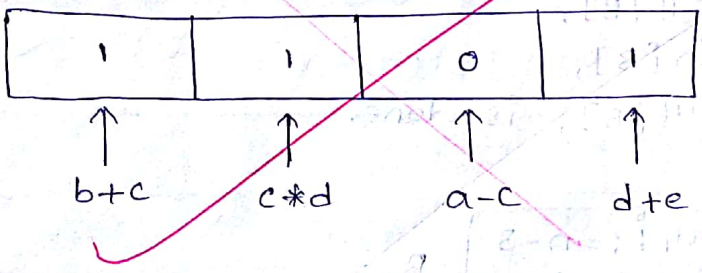
→ Data flow equations can be computed by using forward (or) backward substitutions.

Representing data flow information :-

The data flow information can be represented by either set of properties (or) using bit vector with each bit representing a property.

Eg :- a := b+c
d := c*d
e := a-c
f := d+e

The set of properties for the statements can be represented



here
a := b+c := 1+1 = 1
d := c*d := 1*1 = 1
e := a-c := 1-1 = 0
f := d+e := 1+1 = 1

Therefore [b+c , c*d , d+e]

Data flow equations for programming construct :-

The data flow equation written as

$$out[s] = gen[s] \cup [in[s] - Kill[s]]$$

where → "s" = statement for which data flow equation can be written.

→ "out" represents an output i.e., the information generated at the end of statement.

→ "gen" represents the information generated within the statement.

→ "in" represents gathering information before entering in the loop.

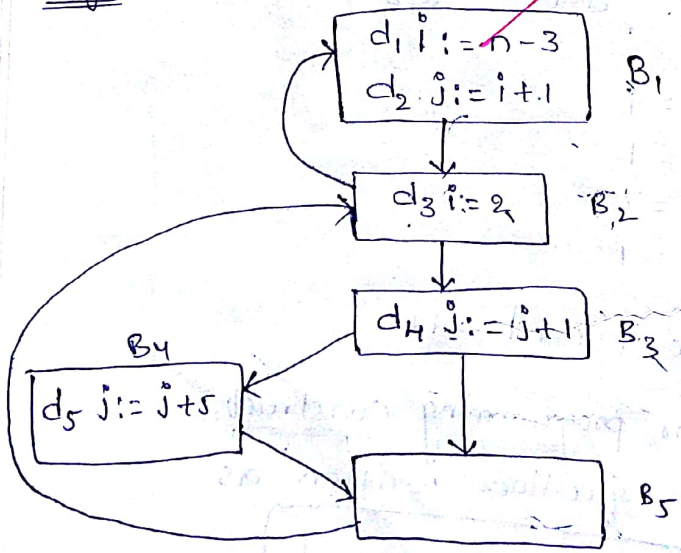
→ "kill" represents the information which is killed with in the control flow.

Reaching definition :-

In this section we will discuss how to obtain the solution to data flow equation using simple and efficient iterative algorithm by using this algorithm the computation of

gen [B],
kill [B],
in [B],
out [B], is done.

Eg:-



Block	gen [B]	kill [B]
B_1	$gen[B_1] = \{d_1, d_2\} = \{11000\}$	$kill[B_1] = \{d_3, d_4, d_5\} = \{00111\}$
B_2	$gen[B_2] = \{d_3\} = \{00100\}$	$kill[B_2] = \{d_1\} = \{10000\}$
B_3	$gen[B_3] = \{d_4\} = \{00010\}$	$kill[B_3] = \{d_2, d_5\} = \{01001\}$
B_4	$gen[B_4] = \{d_5\} = \{00001\}$	$kill[B_4] = \{d_2, d_4\} = \{01010\}$
B_5	$gen[B_5] = \{\phi\} = \{00000\}$	$kill[B_5] = \{\phi\} = \{00000\}$

According to data flow equation we initialize $in[B] = \phi$, $out[B] = gen[B]$ for all basic blocks we get,

Initialization		
Block	in [B]	out [B]
B_1	0 0 0 0 0	{11000}
B_2	0 0 0 0 0	{00100}
B_3	0 0 0 0 0	{00010}
B_4	0 0 0 0 0	{00001}
B_5	0 0 0 0 0	{00000}

From the flow graph predecessor of B_1 is B_2

$\therefore$ $in[B_1] = out[B_2]$

$= \{00100\}$

$out[B_1] = gen[B_1] \cup (\{in[B_1] - kill[B_1]\})$

$= 11000 \cup (\{00100\} - \{00111\})$

$= 11000 \cup (00000)$

$out[B_1] = \{11000\}$

00100
00111
01000

Now consider block B_2

$in[B_2] = out[B_1] + out[B_5]$

$= 11000 + 00000$

$= \{11000\}$

$out[B_2] = gen[B_2] \cup (\{in[B_2] - kill[B_2]\})$

$= 00100 \cup (\{11000 - 10000\})$

$= 00100 \cup (01000)$

$out[B_2] = \{01100\}$

Consider block B_3

$in[B_3] = out[B_2]$

$= \{01100\}$

$out[B_3] = gen[B_3] \cup (\{in[B_3] - kill[B_3]\})$

$= 00010 \cup (01100 - 01001)$

$= 00010 \cup 00100$

$out[B_3] = \{00110\}$

Similar $in[B_4] = out[B_3]$

$= \{00110\}$

$$out[B_4] = gen[B_4] \cup (\{in[B_4] - kill[B_4]\})$$

$$= 00001 \cup (00110 - 01010)$$

$$= 00001 \cup 00100$$

$$out[B_4] = \{00101\}$$

similarly

$$in[B_5] = out[B_3] + out[B_4]$$

$$= 00110 + 00101$$

$$= 00111$$

$$out[B_5] = gen[B_5] \cup (\{in[B_5] - kill[B_5]\})$$

$$= \{00000\} \cup (00111 - 00000)$$

$$= 00000 \cup 00111$$

$$out[B_5] = \{00111\}$$

Then the next two iterations can be computed in this way as we are considering this while loop will execute more than one.

PASS 1		
Block	in[B]	out[B]
B_1	{00100}	{11000}
B_2	{11000}	{01100}
B_3	{01100}	{00110}
B_4	{00110}	{00101}
B_5	{00111}	{00111}

Similarly pass2, pass3 can be calculated in above fashion.

PASS 2		
Block	in[B]	out[B]
B_1	01100	11000
B_2	11111	01111
B_3	01111	00110
B_4	00110	00101
B_5	00111	00111

PASS 3		
Block	in[B]	out[B]
B_1	01111	11000
B_2	11111	01111
B_3	01111	00110
B_4	00110	00101
B_5	00111	00111

After pass 3 same iterations are comming for in[B] and out[B] hence pass 3 is finall iteration.

Live Variable Analysis:-

Live variable is a variable it is used to further after its declaration. Before understanding how to compute data flow equations we will learn two important terms i.e.,

i) use[s] = is a set of variables used by 's'

ii) def[s] = is a set of variable defined by 's'

Eg:-

$x := y + z$

Here use = y, z

def = x

Eg:-

$x := x + 1$

Here use = x

def = x

Let us see an algorithm for computing an data flow equations using Live variable Analysis.

for all i , in[i] = ϕ and out[i] = ϕ

repeat untill no change

for all i

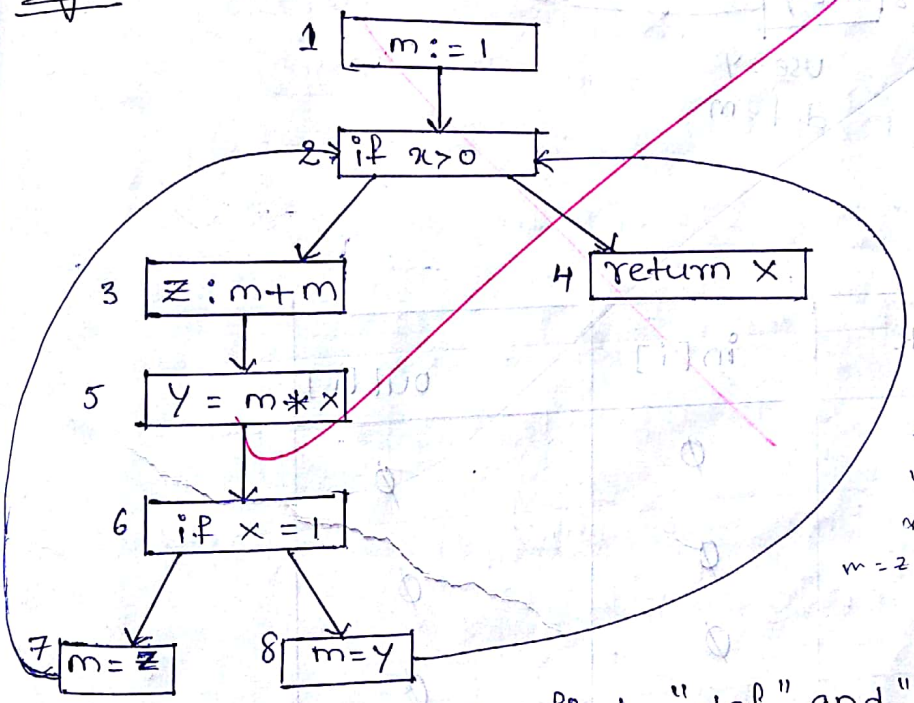
$$out[i] = \bigcup_{i' \in succ[i]} in[i']$$

$$in[i] = use[i] \cup [out[i] - def[i]]$$

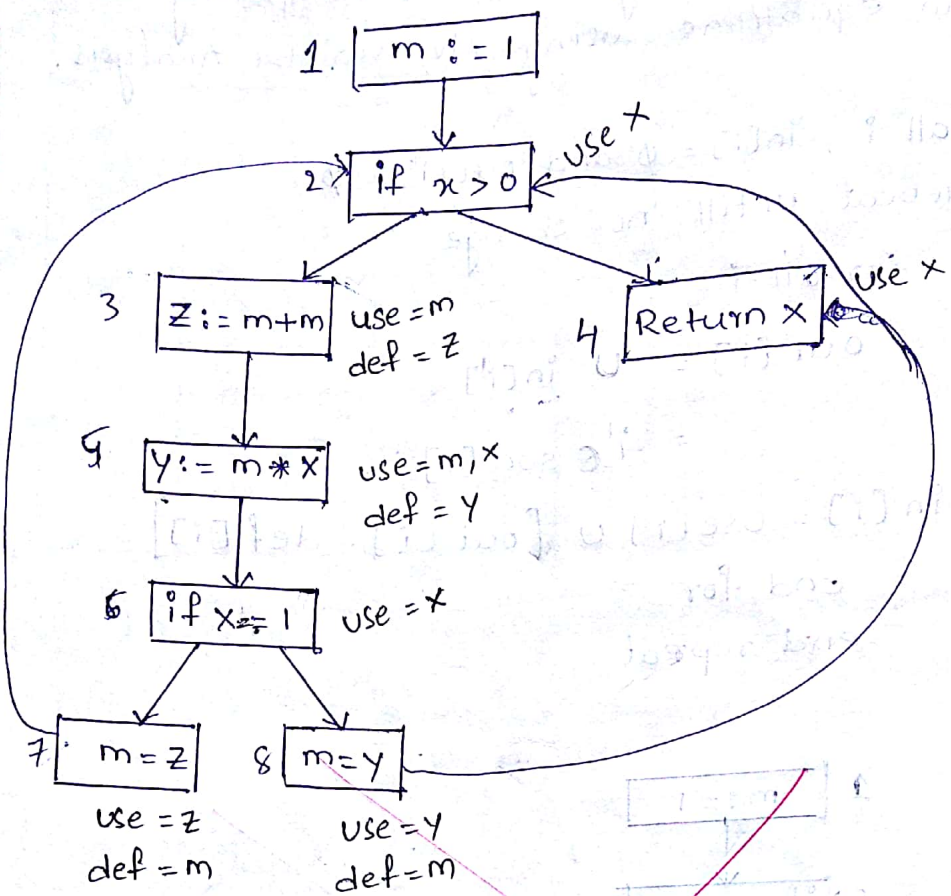
end for

end repeat

Eg:-



At first we will find "def" and "use" for each block in the given above flow graph.



Initially

Block	in[i]	out[i]
1	∅	∅
2	∅	∅
3	∅	∅
4	∅	∅
5	∅	∅
6	∅	∅
7	∅	∅
8	∅	∅

Now we compute out[i] = ∪ in[i'] where 'i' is successor node of 'i'.

$$in[i] = use[i] \cup [out[i] - def[i]]$$

$$out[2] = in[3] \cup in[4]$$

$$= \phi \cup \phi$$

$$\boxed{out[2] = \phi}$$

$$in[2] = use[2] \cup [out[2] - def[2]]$$

$$= x \cup [\phi - \phi]$$

$$\boxed{in[2] = x}$$

$$out[3] = in[5]$$

$$\boxed{out[3] = \phi}$$

$$in[3] = use[3] \cup [out[3] - def[3]]$$

$$= m \cup [\phi - z]$$

$$\boxed{in[3] = m}$$

Since complement z can be eliminated

$$\boxed{out[4] = \phi}$$

$$in[4] = use[4] \cup [out[4] - def[4]]$$

$$= x \cup [\phi - \phi]$$

$$\boxed{in[4] = x}$$

$$out[5] = in[6]$$

$$\boxed{out[5] = \phi}$$

$$in[5] = use[5] \cup [out[5] - def[5]]$$

$$= (m, x) \cup [\phi - y]$$

$$\boxed{in[5] = m, x}$$

$out[6] = in[7] \cup in[8]$

$= \emptyset \cup \emptyset$

$\boxed{Out[6] = \emptyset}$

$in[6] = use[6] \cup [out[6] - def[6]]$

$= x \cup [\emptyset - \emptyset]$

$\boxed{in[6] = x}$

$out[7] = in[2]$

$\boxed{out[7] = \text{X}}$

$in[7] = use[7] \cup [out[7] - def[7]]$

$= z \cup [\text{X} - m]$

$\boxed{in[7] = x, z}$

$out[8] = in[4]$

$\boxed{out[8] = \text{X}}$

$in[8] = use[8] \cup [out[8] - def[8]]$

$= y \cup [x - m]$

$\boxed{in[8] = y, x}$ After 1st pass we get

block	in[i]	Out[i]
1	$\emptyset$	$\emptyset$
2	x	$\emptyset$
3	m	$\emptyset$
4	x	$\emptyset$
5	m, x	$\emptyset$
6	x	$\emptyset$
7	x, z	x
8	x, y	x

Now we computed for pass 2

$out[1] = in[2]$

$= x$

& $in[1] = use[1] \cup [out[1] - def[1]]$

$= \emptyset \cup [x - \emptyset]$

$= x$

$out[2] = in[3] \cup in[4]$

$= m \cup x$

$= m, x$

$in[2] = use[2] \cup [out[2] - def[2]]$

$= x \cup [m, x - \emptyset]$

$= x, m$

$out[3] = in[5]$

$= m, x$

$in[3] = use[3] \cup [out[3] - def[3]]$

$= m \cup [m, x - z]$

$= m, x$

$out[4] = \emptyset$

$in[4] = use[4] \cup [out[4] - def[4]]$

$= x \cup [\emptyset - \emptyset]$

$= x$

Since $in[4]$ & $out[4]$ is same as pass 1 Hence we will not consider $in[4]$ & $out[4]$ again.

$out[5] = in[6]$

$= x$

$in[5] = use[5] \cup [out[5] - def[5]]$

$= m, x \cup [x - y]$

$= m, x.$

$$out[6] = in[7] \cup in[8]$$

$$= x, z \cup x, y$$

$$= x, y, z$$

$$in[6] = use[6] \cup [out[6] - def[6]]$$

$$= x \cup [x, y, z - \phi]$$

$$= x, y, z$$

$$out[7] = in[2]$$

$$= x, m$$

$$in[7] = use[7] \cup [out[7] - def[7]]$$

$$= z \cup [m, x - m]$$

$$= x, z$$

$$out[8] = in[2]$$

$$= x, m$$

$$in[8] = use[8] \cup [out[8] - def[8]]$$

$$= y \cup [x, m - m]$$

$$= x, y$$

PASS 2

	in[i]	out[i]
1.	x	x
2.	x, m	x, m
3.	x, m	x, m
4.	x	ϕ
5.	x, m	x
6.	x, y, z	x, y, z
7.	x, z	x, m
8.	x, y	m, x

PASS 3

$out[1] = in[2]$

$= x, m$

$in[1] = use[1] \cup [out[1] - def[1]]$

$= \emptyset \cup [x, m - m]$

$= x$

$out[2] = in[3] \cup in[4]$

$= x, m \cup x$

$= x, m$

$in[2] = use[2] \cup [out[2] - def[2]]$

$= x \cup [x, m - \emptyset]$

$= x, m$

$out[3] = in[5]$

$= x, m$

$in[3] = use[3] \cup [out[3] - def[3]]$

$= m \cup [m, x - z]$

$= x, m$

$out[4] = \emptyset$

$in[4] = use[4] \cup [out[4] - def[4]]$

$= x \cup [\emptyset - \emptyset]$

$= x$

$out[5] = in[6]$

$= x, y, z$

$in[5] = use[5] \cup [out[5] - def[5]]$

$= x \cup [x, y, z - y]$

$= x, z$

$$out[6] = in[7] \cup in[8]$$
$$= x, z \cup x, y$$
$$= x, y, z$$
$$in[6] = use[6] \cup [out(6) - def[6]]$$
$$= x \cup [x, y, z - \emptyset]$$
$$= x, y, z$$
$$out[7] = in[2]$$
$$= x, m$$
$$in[7] = use[7] \cup [out[7] - def[7])$$
$$= z \cup [x, m - m]$$
$$= x, z$$

out[8], in[8] is same as pass 2

Pass 3

1.	x	m, x
5.	m, y, z	x, y, z

Remaining all are same we will compute pass 4 we get only one changed value

$$out[3] = in[5]$$
$$m, x, z$$
$$in[3] = use[3] \cup [out[3] - def[3]]$$
$$= m \cup [m, x, z - z]$$
$$= m, x$$

in[1] and in[5] are similar to that of privious pass. After pass 4 we get.

3.	m, x.	m, x, z

After this pass we get same values as privious then stop the procedure we get.

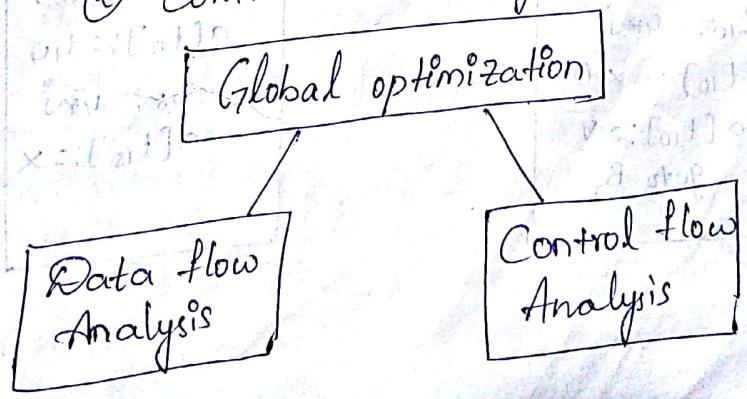
Block	in[i]	out[i]
1.	x	m, x
2.	m, x	m, x
3.	m, x	m, x, z
4.	x	∅
5.	m, x, z	x, y, z
6.	x, y, z	x, y, z
7.	m, x	x, z
8.	x, y	m, x

Global Optimization :-

Local optimization :- The local optimization can optimize the code with in the basic block where as in global optimization we can optimize the code out of the basic block & with in the basic block.

In global optimization we impliment two types of analysis

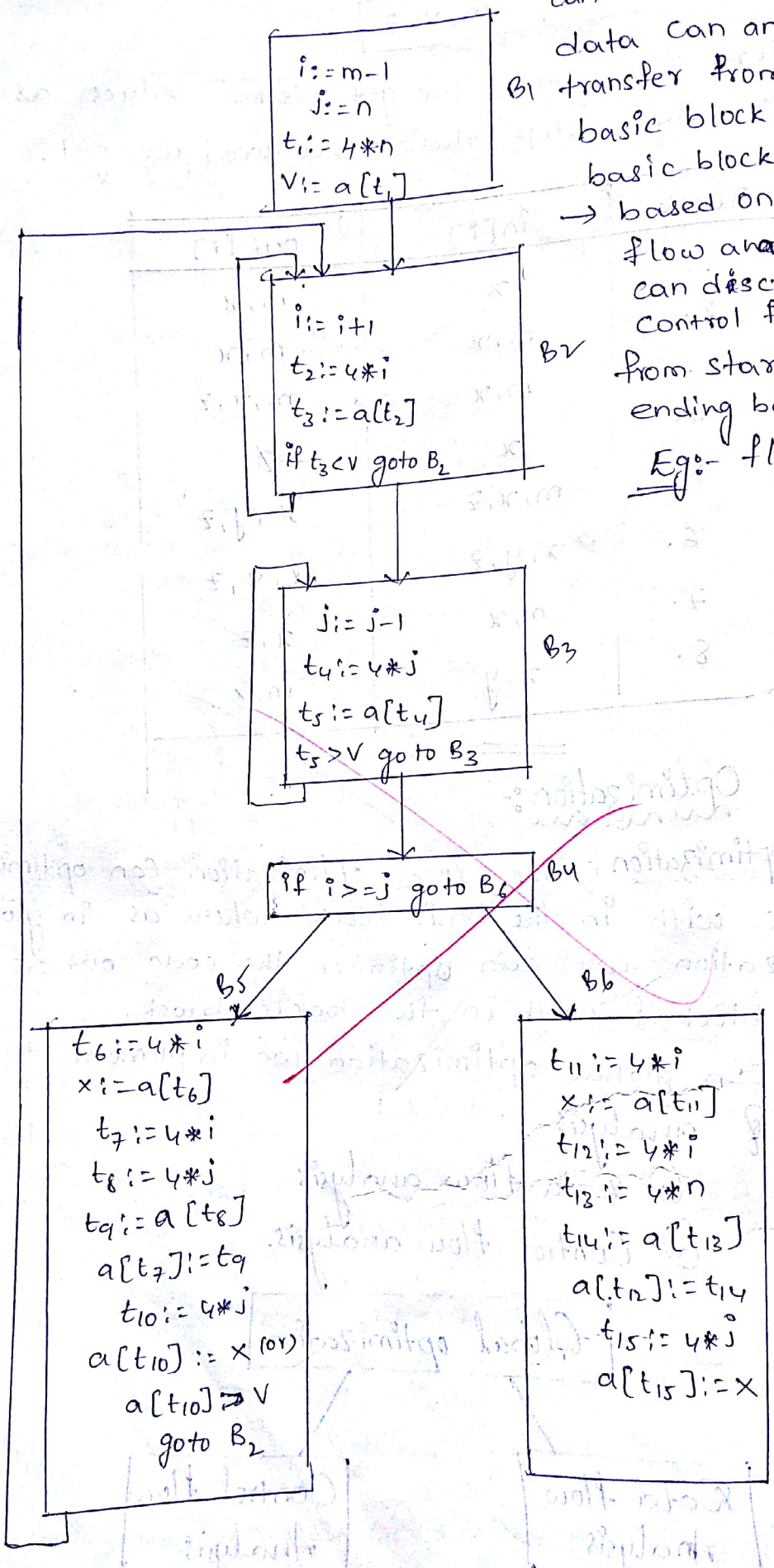
1. Data flow analysis
2. Control flow analysis.



→ In data flow analysis can be described how data can analyze to transfer from one basic block to another basic block.

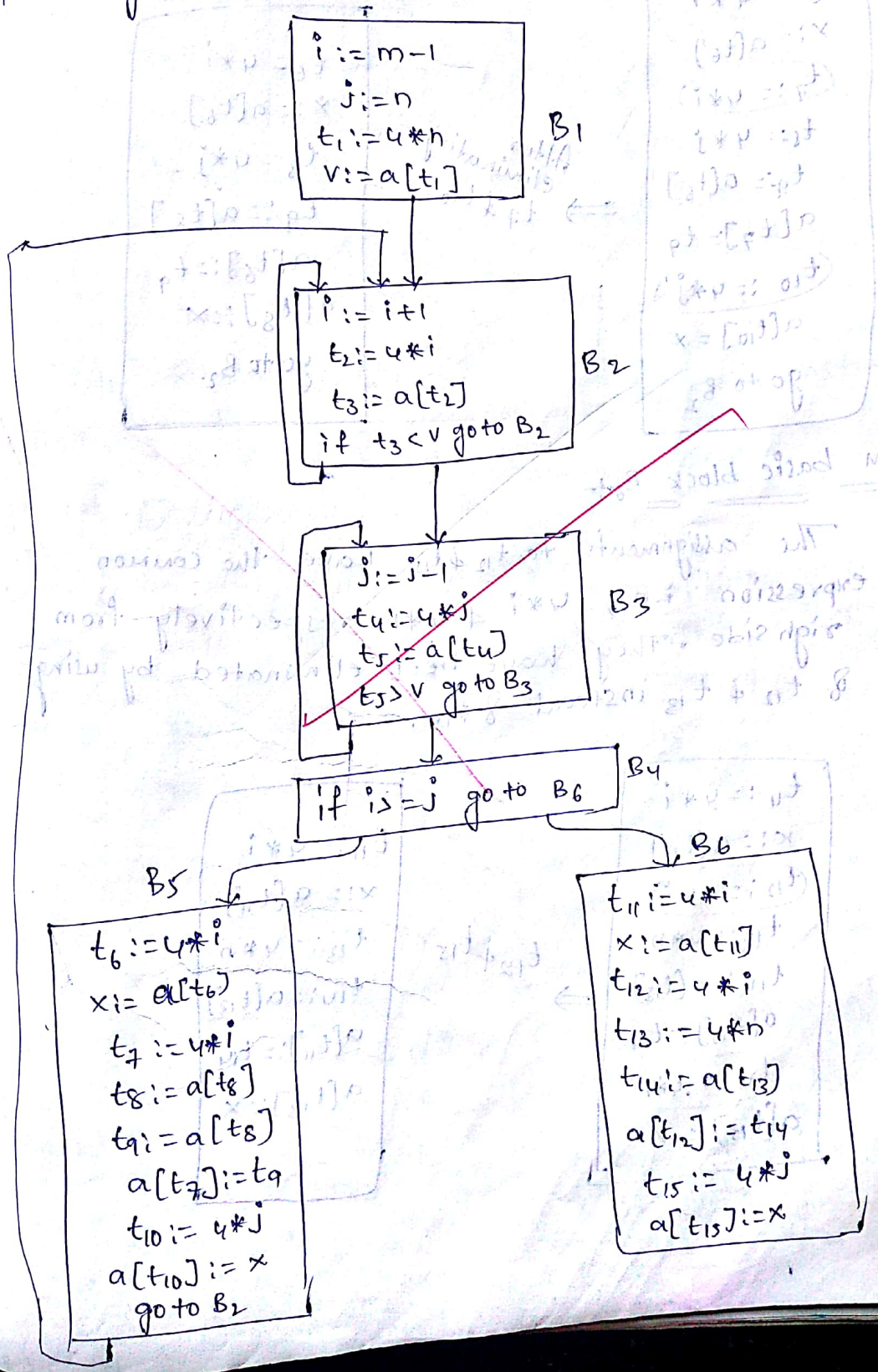
→ based on the data flow analysis we can describe the control flow analysis from starting to ending basic block.

Eg:- flowgraph.



Redundant common sub expression elimination :-

An occurance of an expression 'E' is called a common sub expression, if 'E' was previously computed and the values of variables in 'E' have not changed since the previous computation. we can avoid recomputing the expression if we can use the previously computed value for example



From the basic block B_5

The assignments t
Sub expression i.e., 4*i
right side. They have b
instead of t_1 and t_8 in.

$t_6 := 4 * i$

$x := a[t_6]$

$t_7 := 4 * i$

$t_8 := 4 * j$

$t_9 := a[t_8]$

$a[t_7] := t_9$

$t_{10} := 4 * j$

$a[t_{10}] = x$

go to B_2

Afte
eli
t

from basic block B_6 :-

After local common sub expressions are eliminated B_5 still evaluate $4*i$ and $4*j$. They have eliminated by using t_2 instead of t_6 and t_4 instead of t_8 after eliminating t_6 and t_8 the basic block B_5 can be written as.

$t_6 := 4*i$
$x := a[t_6]$
$t_8 := 4*j$
$t_9 := a[t_8]$
$a[t_6] := t_9$
$a[t_8] := x$
goto B_2

t_6 & t_8 ⟹

$x = a[t_2]$
$t_9 := a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B_2

The common sub expression t_{11} & t_{13} have been eliminated by using t_2 instead of t_{11} & t_{14} instead of t_{13}

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

t_{11} & t_{13} ⟹

$x = a[t_2]$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = x$

B_5 & B_6 still have common sub expression $a[t_1]$ may not have the same value on reaching B_6 and it is not safe to treat $a[t_1]$ as a common sub expression

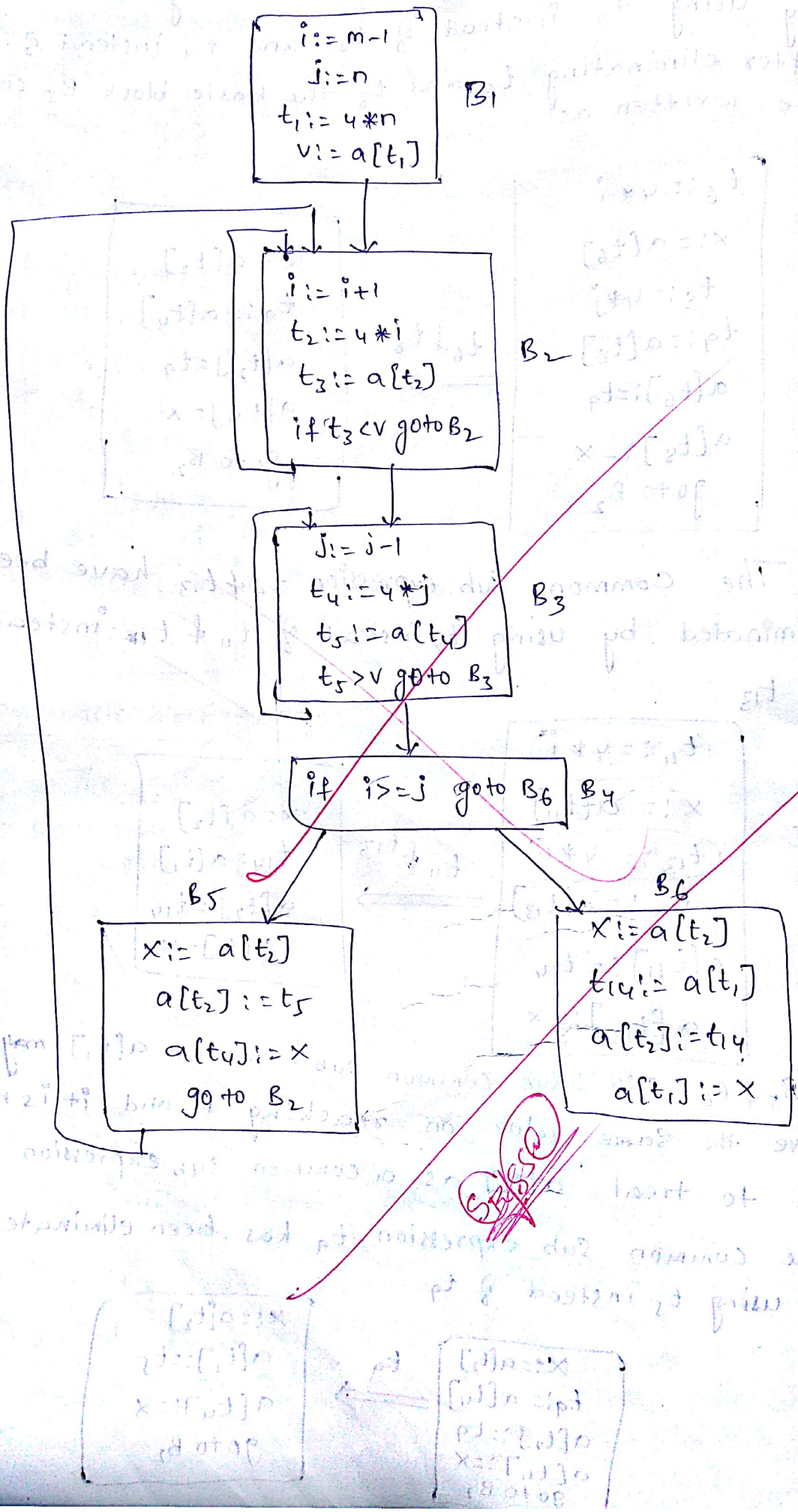
↳ The common sub expression t_9 has been eliminated by using t_5 instead of t_9

$x := a[t_2]$
$t_9 := a[t_4]$
$a[t_2] := t_9$
$a[t_4] := x$
goto B_2

t_9 ⟹

$x := a[t_2]$
$a[t_2] := t_5$
$a[t_4] := x$
goto B_2

The flow graph after common sub expression are eliminated.



22/10/2014

UNIT-8

Object Code Generation

→ object code forms
→ Machine dependent code optimization
→ Register allocation & assignment, generic code generation algorithm
→ DAG for register allocation.

Introduction to code generation:-

The final phase of compiler is code generation. It takes intermediate code representation of the source program as the input and produces a target program (or) object program as it's output.

The semantic representation as shown in bellow.

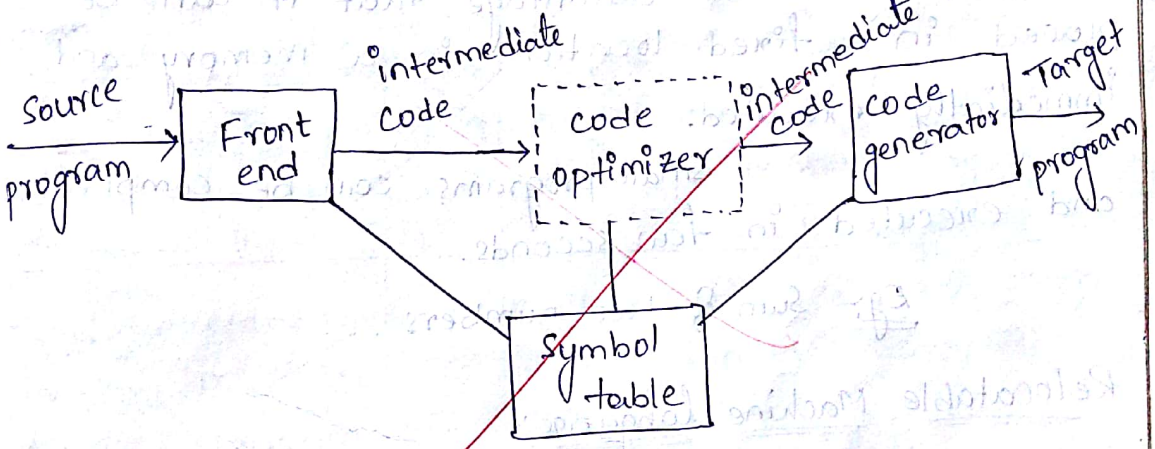


Fig:- position of code generation.

In the above diagram we mainly concentrate on front end and back end.

* Here source program act as front end and target program act as back end.
* In source program we are using different types of programming languages like C, C++, java, PHP, .Net, C#, HTML, CSS, SOAP, SOA, DHTML, XML...

COBAL, PASCAL, Ada, perl, phython.

* In target program nothing but backend here we are using different types of data bases like MYSQL, Oracle 9i, Oracle 10g, SQL, *AMPP, WAMP, LAMP, SQL yogo

Object code forms:-

The output of the code generator is target program (or) object program. The object program may takes for many forms such as

1. Absolute machine language
2. relocatable machine language
3. Assembly language

Absolute machine language:-

producing the absolute machine language as output has the advantage that it can be placed in a fixed location in a memory and immediatly executed.

Small programs can be compiled and executed in few seconds.

Eg:- Sum of two numbers ...

Relocatable Machine language:-

producing the relocatable machine langug as output allows sub programs. And all this sub programs to be compiled separatly.

→ It provides flexibility to compile sub routines separatly and to call previously compiled programs from an object module.

→ If the target machine does not handle relocation automatically.

Assembly language:-

producing an assembly language program as output makes the process of code generation. In this assembly language we use symbolic instructions and macro facilities of the assembler and to generate the target code.

→ Assembly language nothing but machine level language it generates the binary numbers 0 & 1.

→ The advantage for this machine is small memory where compiler must use several passes.

→ The main intension of assembly language is producing high level language as output simplifies code generation even further.

Machine dependent code generation and generic code generation algorithm:-

Machine dependent code generation:-

Any compiler (or) programmer can convert the source program into target program. Target program nothing but machine level language. In machine dependent code generation we are using intermediate code representation as input and target program as output.

→ After complition of target program it generates binary numbers i.e., 0's & 1's.

→ In this we implement micro processor commands and their registers.

Eg:- MOV : it is used to move source to destination.

ADD : it is used to Add source to destination.

MUL : it is used to multipy source to destination

SUB : it is used to substract source from destination.

→ The source and destination fields are not long enough to hold memory address, so certain bit patterns in this field specify that following an instruction contain operands and address.

→ Source & destination of an instruction are specified by combining registers and memory locations with address mode.

→ The address modes together with their assembly language forms and associated cost are as shown in bellow table.

Mode	Form	Address	Address cost.
Absolute	M	M	1
Register	R	R	0
indexed	C(R)	C+ contents (R)	1
Indirect register	*R	contents (R)	0
Indirect indexed	*C(R)	contents (C+ contents (R))	1
Literal	#C	Source to bea constant	1

Eg:- 1. Mov R_0, M ; store the contents of register 'R_0' into memory location 'M'

2. Mov 4(R_0), M : stores the value contents (4 + contents of R_0) into memory location of 'M'

3. Mov *4(R_0), M : Stores the value contents (contents(4+ contents (R_0))) into the memory location of 'M'

4. Mov $1, R_0 : Loads the constant 1 into register R_0

Instruction Cost :- The cost of an instruction to be one plus the cost associated with source and destination address modes. This cost corresponds to length of instructions.

→ For the most machines and for the most instructions, the time taken to fetch an instruction from memory. and we implement 3-address code statement that is $a = b + c$.

This statement can be implemented by many different instruction they are.

	1+address mode cost	Instruction cost
1) MOV b, R_0	1+1	2
2) ADD c, R_0	1+1	2
3) MOV R_0, a	1+1	2
	Total cost is	6

Generic Code generation algorithm :-

DM IT) Write generate code for the following expression

$$x := (a+b) * (c-d) + ((e/f) * (a+b))$$

Sol) First write the 3 address code statement for above expression.

$t_1 := a + b$

$t_2 := c - d$

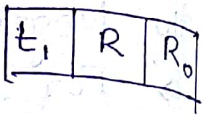
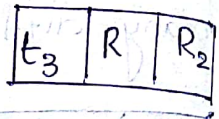
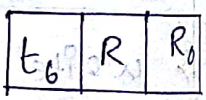
$t_3 := e/f$

$t_4 := t_1 * t_2$

$t_5 := t_3 * t_1$

$t_6 := t_4 + t_5$

using code generation algorithm the sequence target code can be generated as shown in bellow.

Three address Code	Target code Sequence	Register descriptor(R)	operand descriptor.
$t_1 := a+b$	Mov a, R_0 Add b, R_0	Empty R_0 contain t_1	
$t_2 := c-d$	MOV c, R_1 SUb d, R_1	R_1 contain c R_1 contain t_2	
$t_3 := e/f$	Mov e, R_2 div f, R_2	R_2 contain e R_2 contain t_3	
$t_4 := t_1 * t_2$	MUL R_0, R_1	R_0 contain t_1 R_1 contain t_2 R_1 contain t_4	
$t_5 := t_3 * t_1$	MUL R_2, R_0	R_2 contain t_3 R_0 contain t_1 R_0 contain t_5	
$t_6 := t_4 + t_5$	Add R_1, R_0	R_1 contain t_4 R_0 contain t_5 R_0 contain t_6	

2) Generate the code sequence using code generation algorithm for the following expression.

$W := (A-B)+(A-C)+(A-C)$

Sol:- 3-address code for the given expression

$t_1 := A-B$

$t_2 := A-C$

$t_3 := t_1+t_2$

$t_4 := t_3+t_2$

$w := t_4$

Three address code	Target code sequence	register descriptor	operand descriptor
$t_1 := A - B$	Mov A, R_0 SUB B, R_0	Empty R_0 contain t_1	t_1 | R | R_0
$t_2 := A - C$	Mov A, R_1 SUB C, R_1	R_1 contain A R_1 contain t_2	t_2 | R | R_1
$t_3 := t_1 + t_2$	Add R_0, R_1	R_0 contain t_1 R_1 contain t_2 R_1 contain t_3	t_3 | R | R_1
$t_4 := t_3 + t_2$	Add R_1, R_1	R_1 contain t_4	t_4 | R | R_1
$w := t_4$	Mov R_1, w	w contain t_4	

3) generate the code sequence using code generation algorithm for following expression.

$$P := ((x+y) + (z+w)) * ((a+b) + (c-d))$$

Sol

3-address code

$t_1 := x + y$
$t_2 := z + w$
$t_3 := a + b$
$t_4 := c - d$
$t_5 := t_1 + t_2$
$t_6 := t_3 + t_4$
$t_7 := t_5 * t_6$
$p := t_7$.

Three address code	Target code sequence	Register discriptor	operand discriptor
$t_1 := x+y$	Mov x, R_0 add y, R_0	Empty R_0 contain t_1	t_1 R R_0
$t_2 := z+w$	Mov z, R_1 add w, R_1	R_1 contain z R_1 contain t_2	t_2 R R_1
$t_3 := a+b$	Mov a, R_2 add b, R_2	R_2 contain a R_2 contain t_3	t_3 R R_2
$t_4 := c-d$	MOV c, R_3 Sub d, R_3	R_3 contain c R_3 contain t_4	t_4 R R_3
$t_5 := t_1+t_2$	add R_0, R_1	R_0 contain t_1 R_1 contain t_2 R_1 contain t_5	t_5 R R_1
$t_6 := t_3+t_4$	add R_2, R_3	R_2 contain t_3 R_3 contain t_4 R_3 contain t_6	t_6 R R_3
$t_7 := t_5 * t_6$	add R_1, R_3	R_1 contain t_5 R_3 contain t_6 R_3 contain t_7	t_7 R R_3
$P := t_7$	MOV P	P contain t_7	

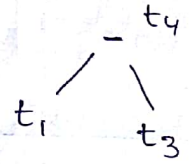
Code generation using DAG :-

1) Compute DAG for following 3-address statement considering DAG as an example & explain the process of code generation (only 2 registors)

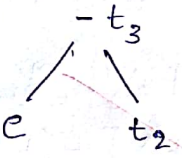
$t_1 : a+b$
$t_2 : c+d$
$t_3 : e-t_2$
$t_4 : t_1-t_3$

Sol We will construct DAG for $t_1 : a+b$
$t_2 : c+d$
$t_3 : e-t_2$
$t_4 = t_1-t_3$

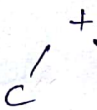
$t_4 = t_1 - t_3$



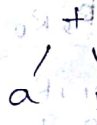
$t_3 = e - t_2$

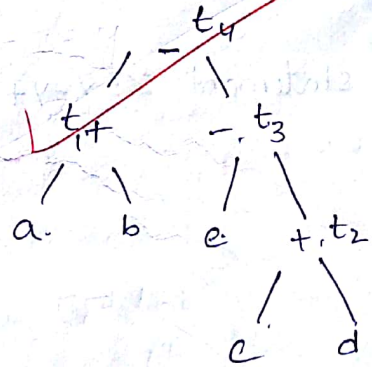


$t_2 = c+d$



$t_1 : = a+b$





For the above order of evaluation we generate generation of code shown in bellow.

```
MOV a, R0      → t1
ADD b, R0
MOV c, R1      → t2
ADD d, R1
```

```
MOV R0, t1
---------
MOV e, R0      → t3
SUB R1, R0
---------
MOV t1, R1
---------
SUB R0, R1
MOV R1, t4
```

Now we change the sequence of computition as.

$t_2 : c + d$

$t_3 : e - t_2$

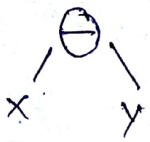
$t_1 : a + b$

$t_4 = t_1 - t_3$

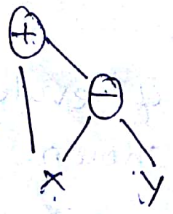
```
MOV c, R0
ADD d, R0      → t2
---------
MOV e, R1
SUB R0, R1     → t3
---------
MOV a, R0
ADD b, R0      → t1
---------
SUB R1, R0
MOV R0, t4     → t4
```

2) Show the DAG for statement $Z = x - y + x * y * u - v / w + x + v$

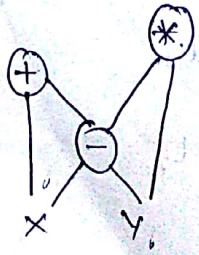
Sol 1) $x - y$



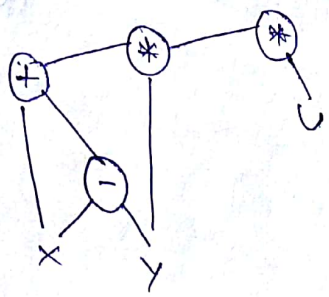
2) $x - y + x$



3) $x - y + x * y$

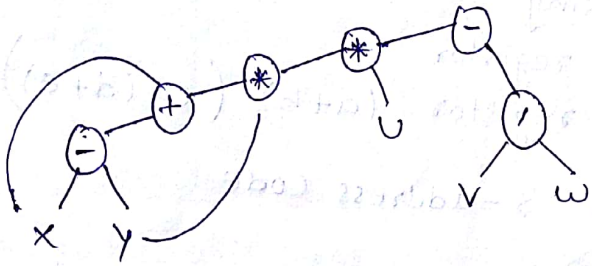


4) x - y + x * y * v

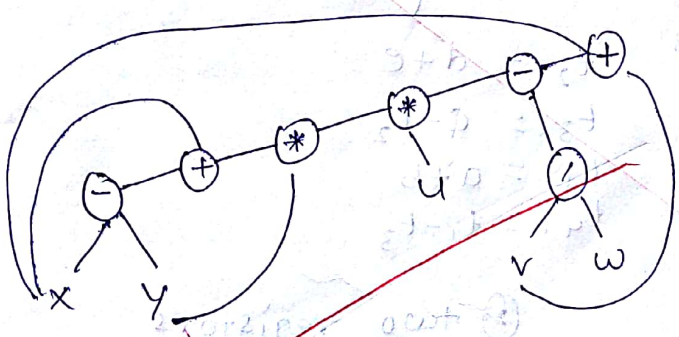


5) v/w

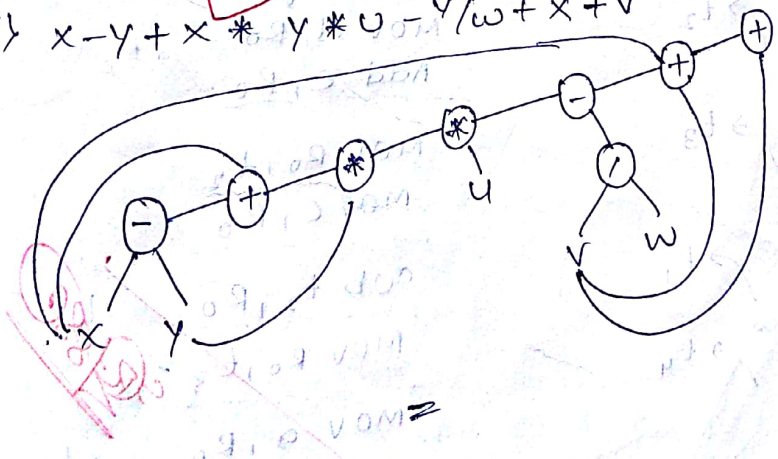
6) x - y + x * y * v - v/w



7) x - y + x * y * v - v/w + x



8) x - y + x * y * v - v/w + x + v



3) generate code for following statements $a = b+c$
$d = a+e$
(only 1 registor)

Sol

Given $a = b+c$
$d = a+e$

```
Mov b, R0
Add c, R0
---------
MOV a, R0
Add e, R0
---------
Mov R0, d
```

4) generate code for following expression obtain optimal code using
1) only 1 registor
2) only 2 registor $(a+b)-(c-(d+e))$

Sol

First we write 3-address code

$t_1 = a+b$
$t_2 = d+e$
$t_3 = c-t_2$
$t_4 = t_1-t_3$

we change the sequence

$t_2 := d+e$
$t_3 := c-t_2$
$t_1 := a+b$
$t_4 := t_1-t_3$

(1) One registor

```
Mov d, R0   → t2
add e, R0
---------
Mov c, R1   → t3
Sub R0, R1
---------
Mov a, R0   → t1
Add b, R0
---------
Sub R1, R0  → t4
```

(2) two registors

```
MOV d, R0
Add e, R0   → t2
---------
MOV R0, t2
MOV c, R0
---------
Sub t2, R0  → t3
---------
MOV R0, t3
---------
MOV a, R0   → t1
Add b, R0
---------
Sub t3, R0  → t4
```

Register allocation and assignment :-

* Global Register Allocation
* Usage counts
* Register Assignment for outer loops
* Register Allocation by graph coloring