chapter-4

Code optimization

This phase will take intermediate code as i/p and produces optimized code as an output:

→ Code optimization phase algorithm can often reduce execution time & memory utilization.

→ During the optimization phase even if we apply many more no. of code optimization techniques there is no guarentee that generated code will be completely optimized.

→ That's why it is called code improvement phase.

→ While applying the code optimization mechanism, we should follow the following constraints.

(1) Apply any no. of code optimization techniques. But we should preserve the meaning of the program. i.e., the target code should ensure symantic equivalence with source pgm.

(2) If we apply any code optimization mechanism then we should get minimum benefit either in the form of reducing execution time or in the form of reducing memory utilization

(3) Without having proper target, we should not apply any code optimization mechanism

→ Optimizations are classified into 2 categories
1) Machine independent optimization
2) " Dependent "

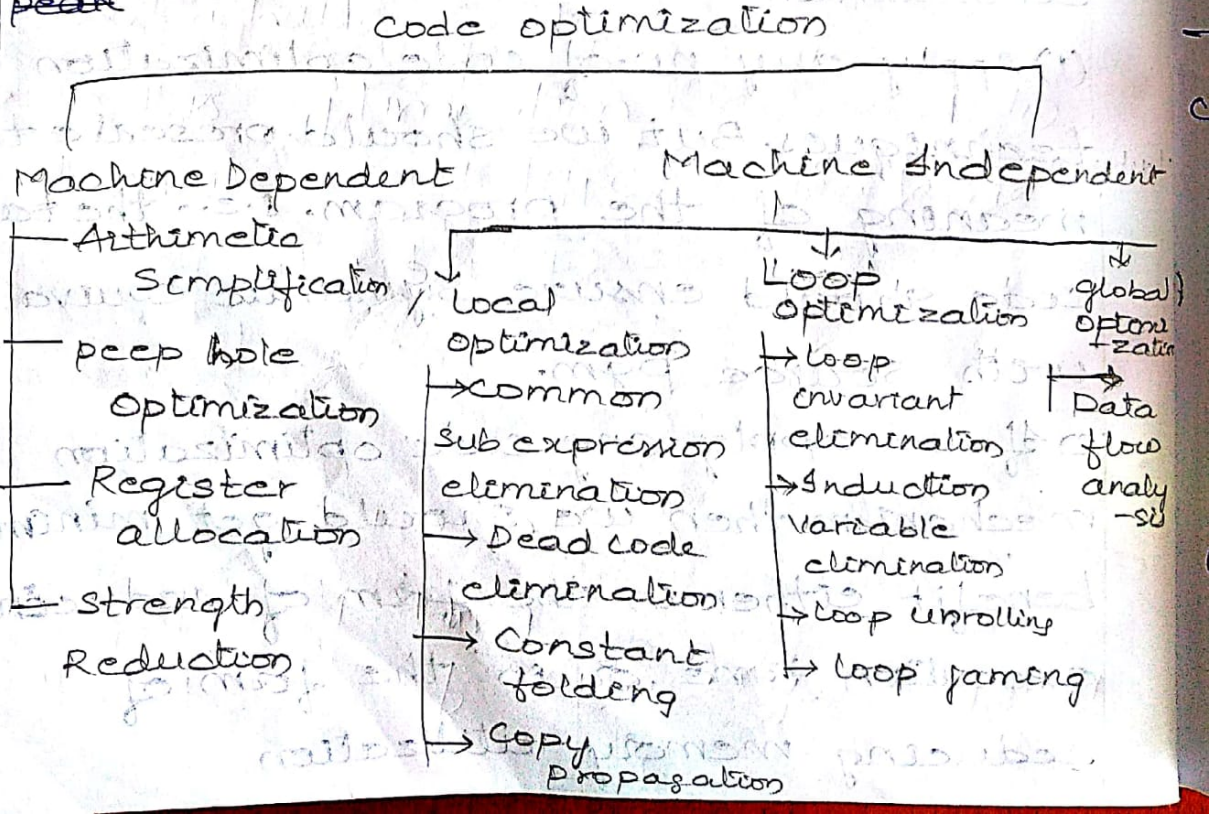
Machine Independent Optimization:

→ These techniques donot consider the features of target machine

Ex: Constant folding, Dead code elimination, Constant propagation ---

Machine Dependent Optimization: These techniques require specific information related to target machine.

Ex: Register allocation, strength reduction, peep hole optimization ---



Local optimization: A transformation of a program is called local if it is performed only inside the statements of basic block →otherwise it is called global.

① Common sub expression elimination:

→Occurence of an expression 'E' is called common sub expression if E was previously computed and the value of variables in E have not changed since the previous computation.

→We can avoid recomputing the expression if we use previously computed values.

Ex: t1=4*i
t2=a[t1]
t3=4*j
t4=4*i
t5=n
t6=b[t4]+t5

→The code can be optimized by using common sub expression elimination as

t1=4*i
t2=a[t1]
t3=4*j
t5=n
t6=b[t1]+t5

② Copy propagation:-

Assignments of the form f:=g is called copy statement. It increases the execution

time by creating more no of variables even though they are not used.

Example:- $x = pi$

$A = x * r * r$

→ The optimization using copy propagation can be done by using original variable instead of temporary variables.

ex:- $A = pi * r * r$

here the variable x is eliminated as shown above.

③ Deadlode Elimination :- In programs some block of code may not execute at anytime. Such block of code is called Deadcode (or) Unreachable Code.

→ In this optimization we need to eliminate such type of deadcode.

ex:-
```
final int i = 10;
if (i != 10)
{
    -----
    -----
    Statements
    -------
}
```

④ Constant Folding

→ The procer of replacing compiler

evaluated - expression results in place of particular variable at runtime is called constant folding

Example:-

```
#define a 50
void main()
{ int i=10;
  i=i+a;
  i=i*a;
}
```

```
Void main()
{ int i=10;
  i=i+50;
  i=i*50;
}
```

Loop optimization

① Loop invariant Computation Elimination:-

This optimization is also called as 'Code motion'. In this optimization we should eliminate the loop invariant Computations from the loop.

→ Loop invariant Computations should not vary their results from iteration to iteration.

Example:-

```
int j=50;
for(int i=0; i<10; i++)
{ j=50;
  ----
}
```

```
int j=50
for(int i=0; i<10;
        i++)
{ ----
}
```

(2) Induction Variable Elimination :-

Value of induction variable ~~doesn't~~ increase / decrease its value with a constant length from iteration to iteration.

→ In this optimization we should elimina such type of induction variables & we need to manage the logic by using loop variables.

Ex:-

```
int i=0;
for(int j=0;j<10;j++)
{ printf("%d",i);
  i=i+10;
}
```

```
for(j=0;j<100;
                j+10)
{ Printf("%d",j);
}
```

(3) Loop Unrolling :- The main purpose of this optimization is to reduce no of test conditions in the loop.

→ In this optimization. we write the same loop code twice in order to reduce no of tests.

Ex:-

```
for(i=0;i<10;i++)
a[i]=50;
```

```
for(i=0;i<10;i++)
{ a[i]=50;
  i++;
} a[i]=50;
```

(4) Loop Jamming :- The main purpose of this optimization is to reduce the loops.

→ It is the process of merging loops inorder to decrease the program code execution.

example:-

```
for(int i=0; i<10; i++)
{ for(int j=0; j<10; j++)
    a[i][j]=20;
}
for(k=0; k<10; k++)
  b[k] = 30;
```

```
for(int i=0; i<10; i++)
{ for(int j=0; j<10; j++)
    a[i][j]=20;
  b[i]=30;
}
```

Machine Dependent optimization Techniques

(1) Arithmetic Simplification :-

```
ADD 5,R0
ADD 10,R0   => ADD 15,R0
```

(2) peep hole optimization

There are 2 types of peephole optimiz-ation.

(1) Elimination of redundent loads :-

```
LOAD R0,a;
```

LOAD a, R0

(2) Reducing the goto instructions

Case 1

100) goto 105
=
105) goto 110
=
110) =

⇒

100) goto 110
=
110) =

Case 2

100) if then goto 105
=
105) if then goto 110
=
110) =

⇒

100) if then goto 110
=
110) =

Case 3

100) goto 105
=
105) if then goto 110
=
110) =

⇒

100) if then goto 110
=
110) =

Case 4:

100) if then goto 105
=
105) goto 110
=
110) =

⇒

100) if then goto 110
=
110) =

Using Machine Idioms:-

→ The target machine may have H/w instructions to implement certain specific operations effectively.

→ for example :- Some machines have auto-increment / auto-decrement addressing modes. These modes improve the quality of code.

$a = a + 1$

⇓

INC a

Global optimization

Data flow Analysis

Basic Block:- A basic block is a set of consecutive statements that are executed sequentially.

→ once the control enters into block then every statement in basic block is executed one after other before leaving the block.

Flow Graph :- A Graphical Representation of three-address code is called Flow Graph.

→ flow-Graph shows the ~~diff~~ relation b/n basic blocks and its preceeding and successor block.

→ The nodes in flow graphs represent a single basic block & edges represent the flow of control.

Procedure to identify Basic Block

→ for given 3-address code identify the leader statements & group the leader statements with the statements upto the next leader.

→ To identify the leader use the following rules.

(1) first statement in a program is a leader.

(2) Any statement that is the target of conditional / unconditional statement is a leader statement.

(3) Any statement that is immediately follows a conditional / unconditional statement is a leader statement.

example:-

```
main()
{   int i=0, n=10;
    int a[n];
    while (i<=(n-1))
    {  a[i]=i*1;
       i=i+1;
    }
  return 0;
}
```

3-address code for above code

1) i := 0
2) n := 10

B_1 → leader

3) t_1 := n-1
4) if (i > t_1) goto 12

B_2 → leader

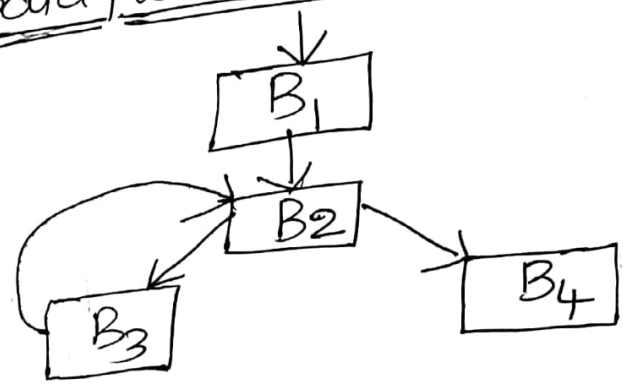
5) t_2 = i * i
6) t_3 := 4 * i
7) t_4 := a[t_3]
8) t_4 := t_2
9) t_5 := i+1
10) i := t_5
11) goto (3)

B_3 → leader

12) return

B_4

Data flow Graph



② 1) i = 1 — ①
2) j = 1 — ②

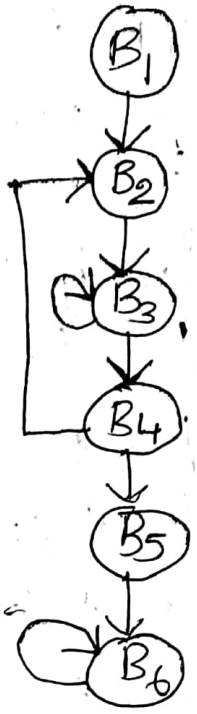
3) t_1 = 10 * i — ③
4) t_2 = t_1 + j
5) t_3 = 8 * t_2
6) t_4 = t_3 - 88
7) a[t_4] = 0.0
8) j = j + 1;
9) if (j <= i) goto 3

10) i = i+1
11) if (i<=10) goto 2

12) i = 1

13) $t_5 = i-1$
14) $t_6 = 8 * t_5$
16) a[16] = 1.0
17) i = i+1
18) if (i<=10) goto 13

④



⑤

⑥

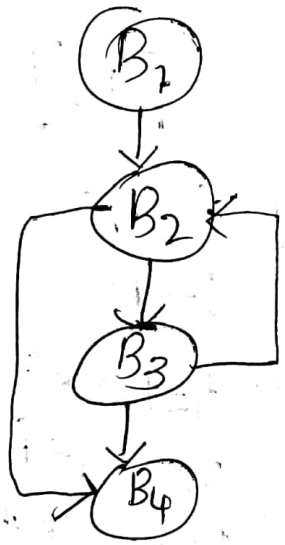
③
```
fact(x)
{
   f = 1
   for(i=2; i<x; i++)
      f = f * i;
   return f;
}
```

1) f = 1
2) i = 2
— B_1

3) if i > x goto 9 — B_2

4) $t_1 = f * i$
5) $f = t_1$
6) $t_2 = i++$
7) $i = t_2$
8) goto 3
— B_3

9) return f — B_4



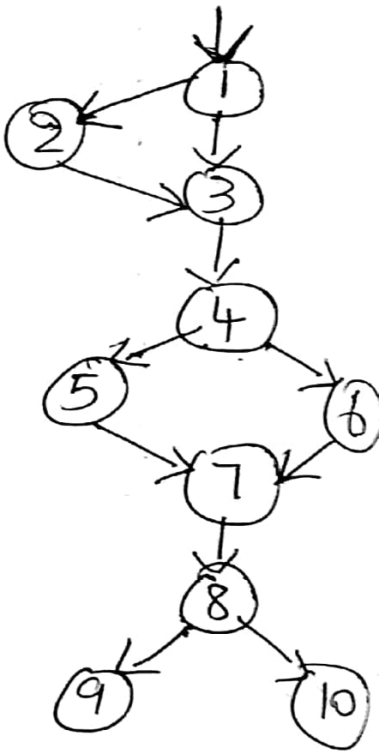
(4)
```
int i
i = 1
while (a<10)
{ if (x>y)
    a = x+y
  else
    a = x-y
}
```

1) i = 1 ——— (1)
2) if (a>10) goto 9 - (2)
3) if (x<y) goto 6 (3)
4) $t_1 = x+y$ — (6)
5) ~~$t_1 = x+y$~~ $a = t_1$
6) $t_2 = x-y$ — (5)
7) $a = t_2$
8) goto 2
9) next — (4)

(5)
```
c = 0
do
{ if (a<b) then n = n+1
  else
    n = n-1
  c++
} while (c<5);
```

Dominators :- "A" node "K" dominates node 'R' if every path from the start node to node R goes through K we say that node K is a dominator of R.

① Identify domintor set of each node of the following flow graph & Construct dominator.



dom(1) = {1}

dom(2) = {1, 2}

dom(3) = {1, 3}

dom(4) = {1, 3, 4}

dom(5) = {1, 3, 4, 5}

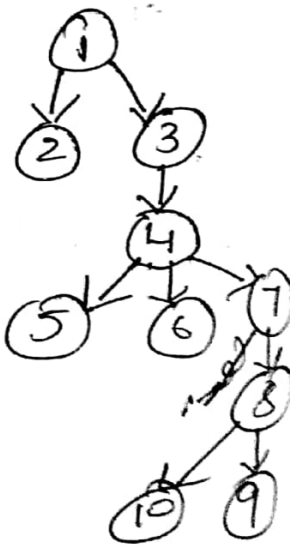
dom(6) = {1, 3, 4, 6}

dom(7) = {1, 3, 4, 7}

dom(8) = {1, 3, 4, 7, 8}

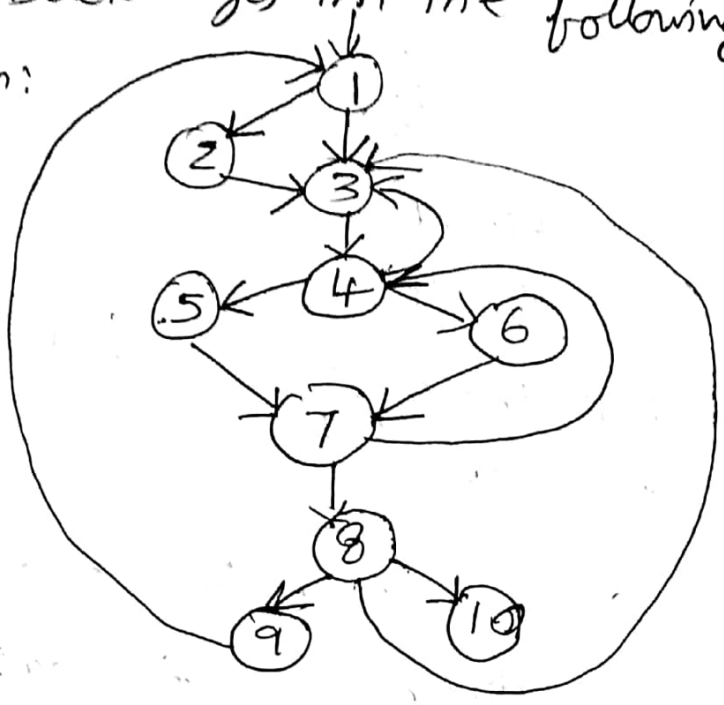
dom(9) = {1, 3, 4, 7, 8, 9}

dom(10) = {1, 3, 4, 7, 8, 10}



Back Edge :- $b \rightarrow a$ is a backedge if and only if a dominates b

ex:- Identify the back edges in the following data flow Graph:



Natural Loop :-

There are 2 essential properties of loop

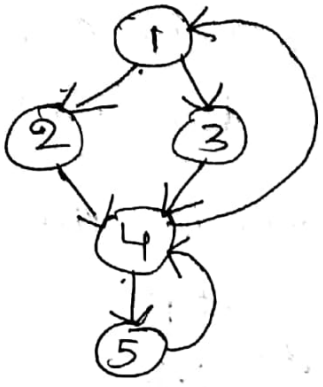
① A loop must have a single entry Point called header. This entry Point dominates all the nodes in the loop.

② There must be atleast one way to iterate the loop that is atleast one path back to the header.

Loop in the flow graph can be denoted by $n \rightarrow d$

example:-

(1)



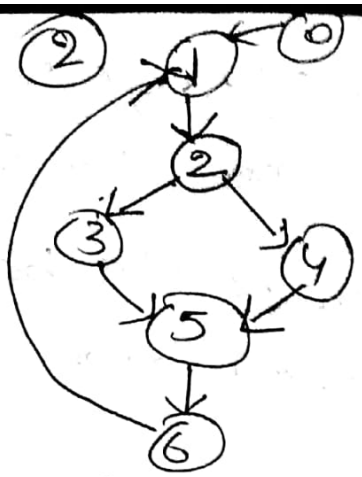
4 -> 1

5 -> 4

4-1-{4,2,3,1}

5-4-{5,4}

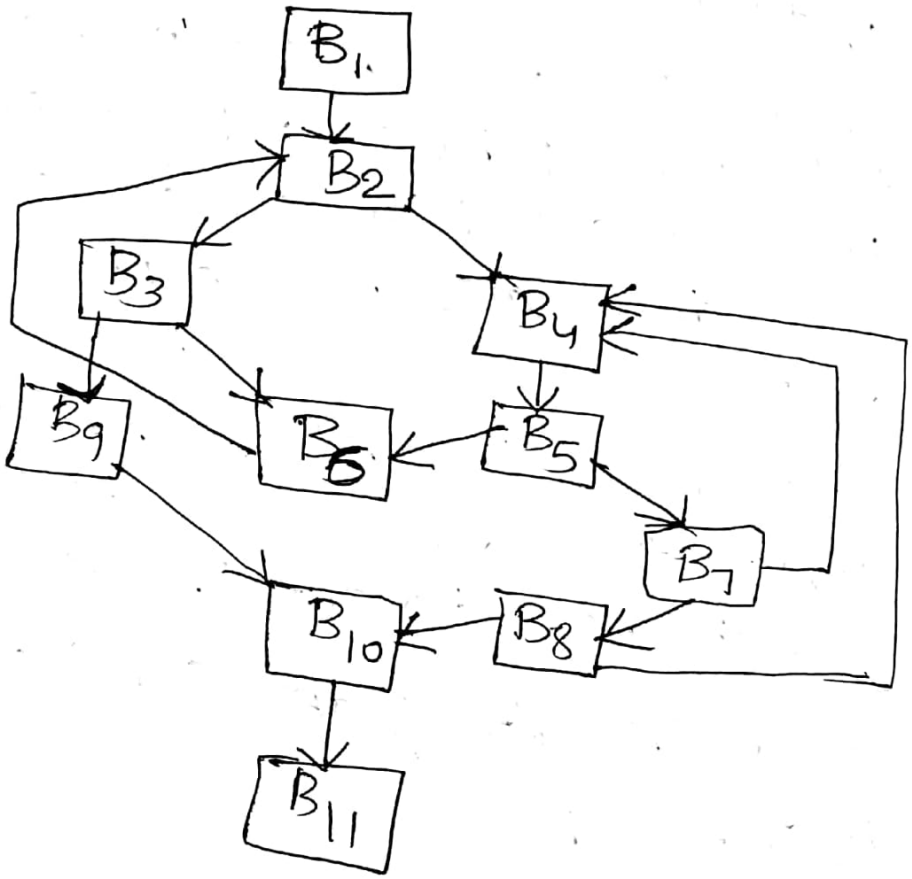
(2)



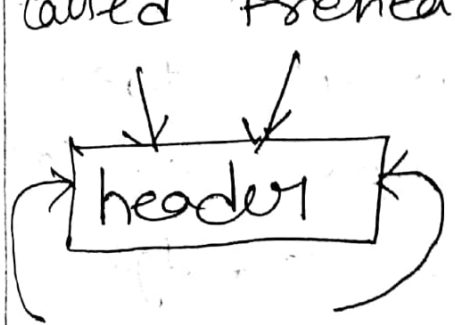
6 -> 1

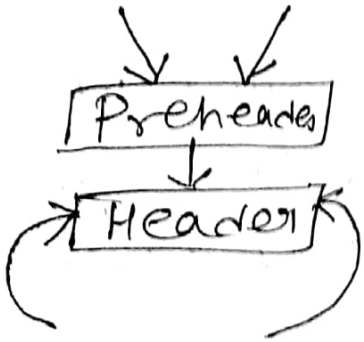
6-1-{6,5,4,3,2,1}

Consider the following flowgraph & find the dominators for each basic block. Detect all the loops in graph.

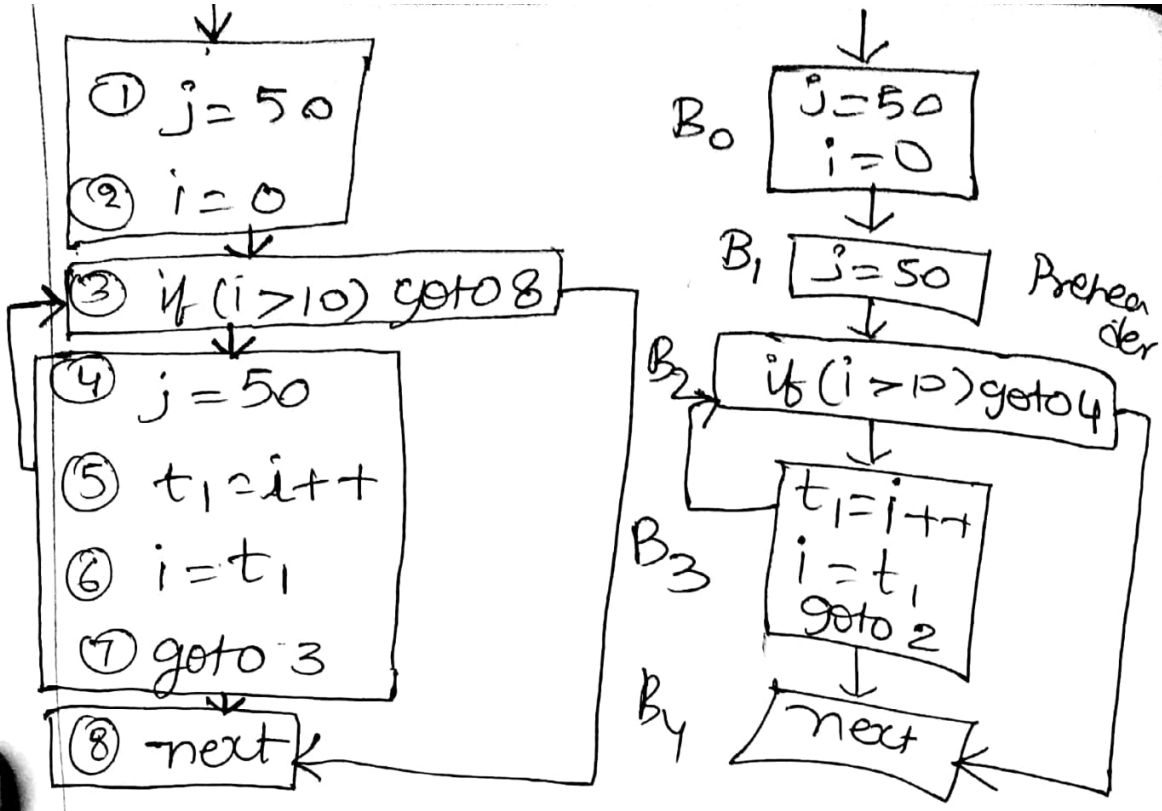


Preheader :- The first statement of the loop is called header. Several transformations required as to move loop invariant statements before the head. Therefore, we create a new block before the header is called Preheader.





ex:-
```
j=50;
for(int i=0; i<10; i++)
{ j=50;
}
```



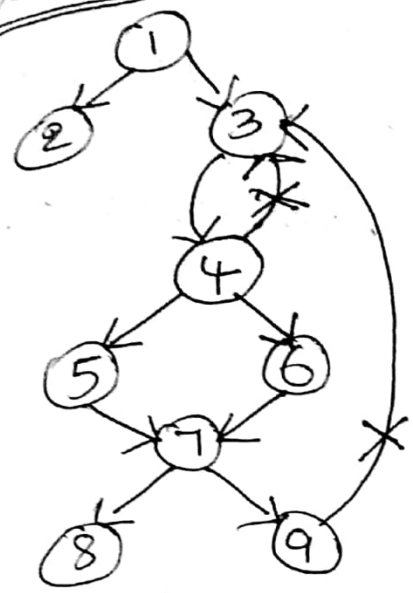
Reducible Flow Graph :- These graphs follow the property that there are no jumps statements into the middle of loops from outside. The only entry point is through its header.

→ A flow Graph 'G' is reducible if and only if we can partion the edges into two disjoint groups often called forward edges and back edges.

→ To check whether the graph is reducible remove the back edges in the flow graph if the resultent graph is acyclic then it is reducible.

All the for, while, do while statements are reducible.

Examples:-



Reducible flow graph

irreducible flow graph.

→ optimization techniques like code motion, induction variable elimination, etc.. can't be directly applied to irreducible graphs.

Global Data Flow Optimization

In Global data flow optimization we analyse flow of information throughout the program.

Data flow properties:-

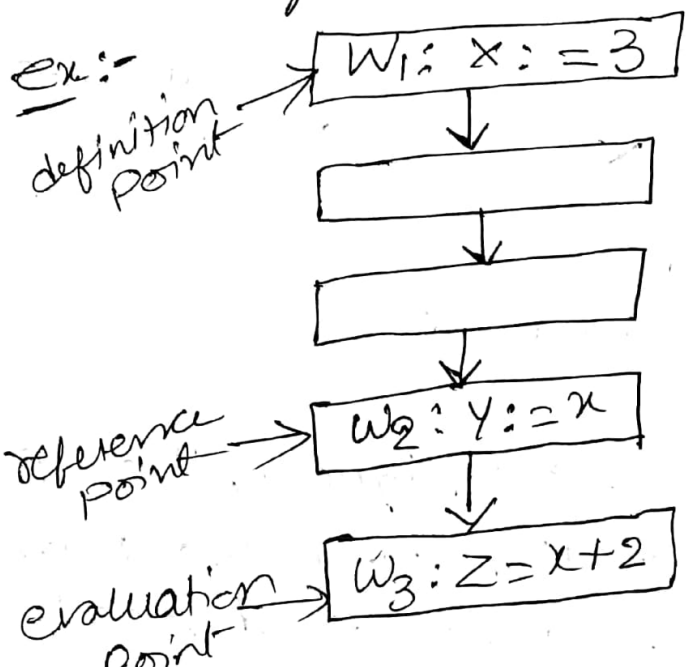
① Point:- It is a label of a statement in a program.

② Reference point:- It is a point at which a reference to a data item is made

③ Evaluation Point :- A program Point at which some evaluation expression is given.

④ Definition point :- It is a program Point at which a variable x is assigned with some value.

ex :-



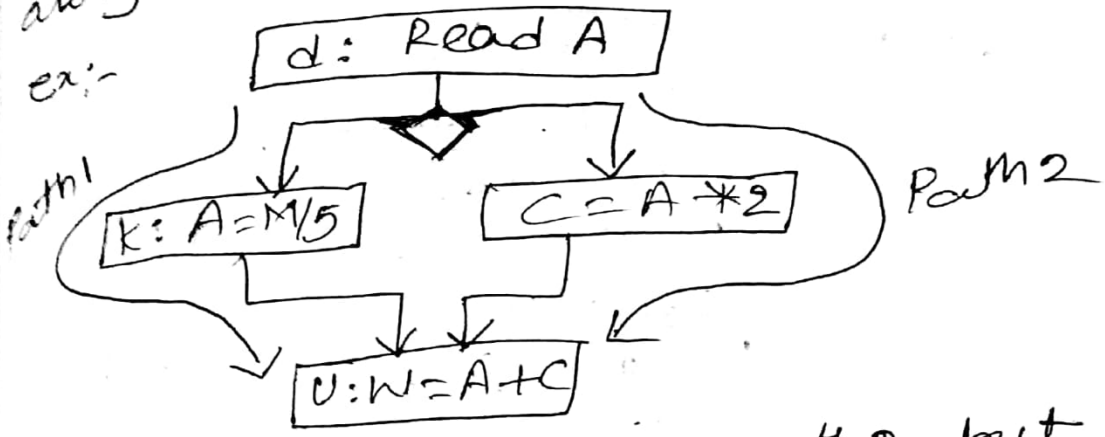
Path :- A path from P_1 to P_n is a sequence of $P_1, P_2 \ldots P_n$ such that for each i b/n $i = 1$ to $n-1$ is either.

(i) P_i is immediate preceding statement of P_{i+1} in the same block (or)

(ii) P_i is end of some block & P_{i+1} is beginning of successor block.

Reaching Definition :- We say a definition 'd' reaches a point 'p' if there is immediate path from d' to 'p' such that 'd' is not killed along the path.

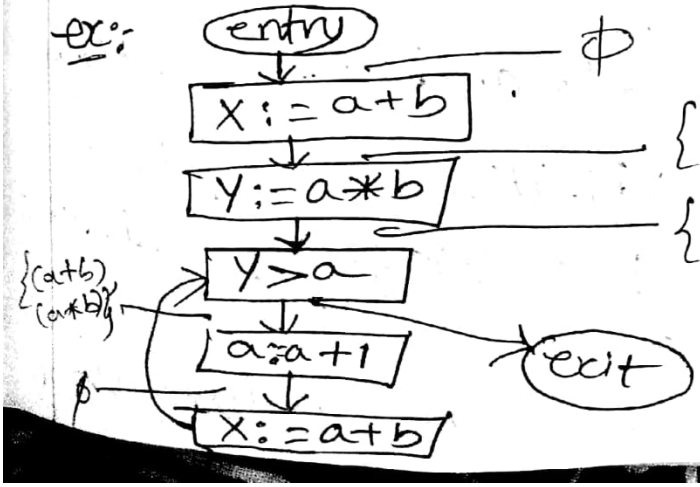
ex:-



'd' reaches 'U' along the path 2 but 'd' doesnot reach along path1 because A is reassigned with some other value at point K.

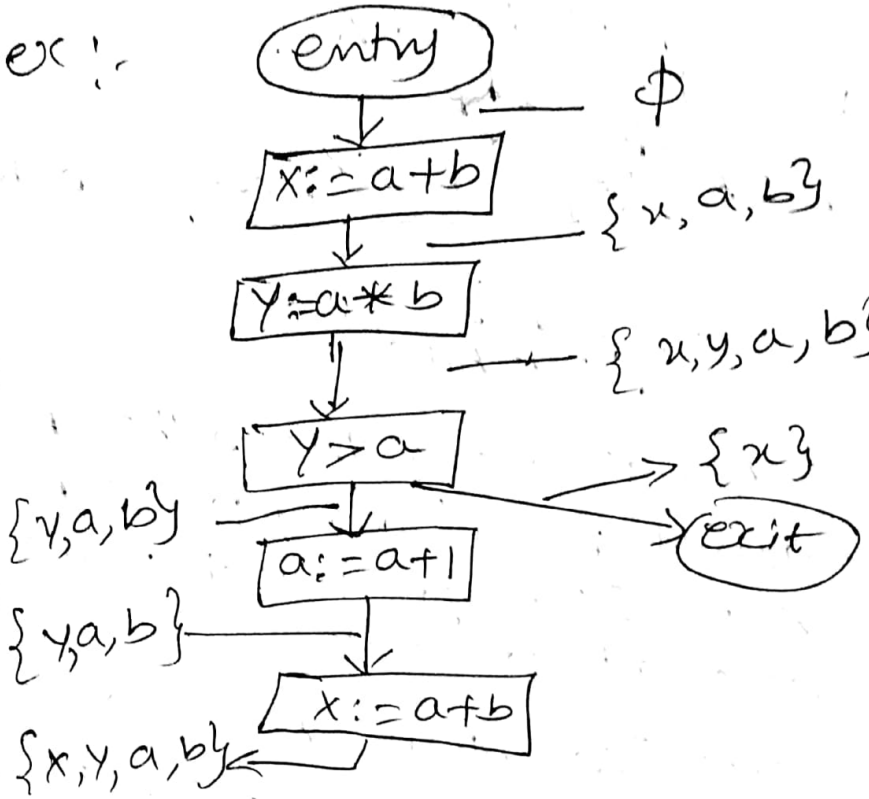
Available Expressions:- An expression 'e' is available at program point 'p' if e is computed on every path to P & the value of e has not changed since the last time 'e' was computed on the path to 'P'.

ex:-



Live Variable :- A variable 'v' is live at program point 'p' if v will be used on some execution path originating from p before v is over written.

ex :-



Very Busy Expression :- An expression 'e' is very busy at point 'p' if on every path from 'p' expression 'e' is evaluated before the value of expression change.

Applications Of Global data flow analysis

① Reaching definitions analysis is used to find the usage of uninitialized variables.

② Available expression analysis used

to avoid recomputing expressions.

(3) live variable analysis used to allo -cate registers effectively.

(4) very busy expresion analysis used to reduce code size.

Data flow equation :- Data flow information can be collected by setting of & solving system of equati -ons that relate information at various points in a program.

→ A data flow equation is in form

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$

where $IN[B] = \bigcup_{P} OUT[P]$

P is predessor of Block B.

→ This equation states that at end of a block information is either generate -d within a block (or) enters at the begining & is not killed as the ctrl flows through the block.

IN[B] - set of definitions of each reachable variable at the begin -ing of block b.

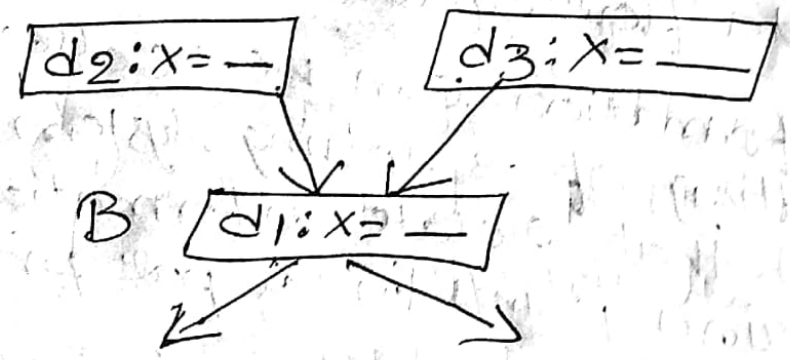
OUT[B] - set of definitions of each reachable variable at the

end of block b.

KILL[B] :- All definitions of a variable x defined in block b.

GEN[B] :- Set of definitions of newly generated variables in block b.

ex :-



GEN[B] = {d1}

IN[B] = {d2 d3}

KILL[B] = {d2, d3}

OUT[B] = GEN[B] ∪ (IN[B] − KILL[B])

= {d1}

Data flow analysis of structured Programs

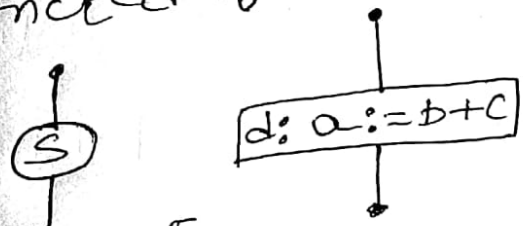
flow graphs for ctrl flow construct such as do while statements have single begining point at which control enters & single end point at which control leaves when execution of statement is over.

(1) Generate flow graph for following expressions & also mention dataflow

equations for reaching definitions.

$S \rightarrow id := E$
$\quad | \; S ; S$
$\quad | \; if \; E \; then \; S \; else \; S$
$\quad | \; do \; S \; while \; E$
$E \rightarrow id + id$
$\quad | \; id$

expressions in this language are similar to those intermediate code but flow graphs for statements of restricted forms



$S \rightarrow id := E$

$GEN[S] = \{d\}$

$KILL[S] = Da - \{d\}$

$OUT[S] = GEN[S] + (IN[S] - KILL[S])$

~~$S \rightarrow id := E$~~ $GEN[S] = GEN[S_2] \cup (GEN[S_1] - KILL[S_2])$

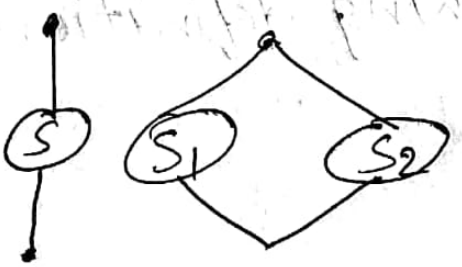
$KILL[S] = KILL[S_2] \cup (KILL[S_1] - GEN[S_2])$

$IN[S_1] = IN[S]$

$IN[S_2] = OUT[S_1]$

$OUT[S] = OUT[S_2]$

$S \rightarrow S ; S$



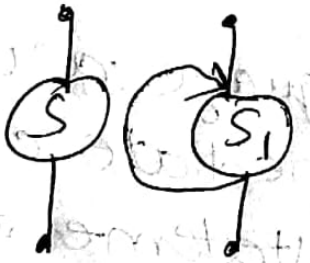
$GEN[S] = GEN[S_1] \cup GEN[S_2]$

$KILL[S] = KILL[S_1] \cap KILL[S_2]$

$IN[S_1] = IN[S]$

$IN[S_2] = IN[S]$

$OUT[S] = OUT[S_1] \cup OUT[S_2]$



$GEN[S] = GEN[S_1]$

$KILL[S] = KILL[S_1]$

$IN[S_1] = IN[S] \cup GEN[S]$

$OUT[S] = OUT[S_1]$.