12/3/18

UNIT - 5

Code optimization:

→ Takes i/p as o/p of intermediate code.

→ Resulting pgm takes less space & less execution time

→ Construct - group of statements
|
equivalent efficient

→ The time Space should be more than the time should be code.

→ principle source of optimization

(1) Inner Loops
(Loop variants)

→ identify the statements and move point to the outside is called "Loop variants".

(2) → Efficient use of Registers and instructions

(3) → Common sub expressions

Eg: A[J+1] = B[J+1]

P: J+1

A[P]: B[P]

(1) For Loops:-

```
Begin
    prod := 0
    I = 1
do
 begin
    Prod = prod + A(I) * B(I)
      I = I + 1
  end
 while I ≤ 20
end
```

Three address code:-

1. prod := 0 ✓
2. I := 1
3. $T_1 = 4 * I$ ✓
4. $T_2 = addr(A) - 4$
5. $T_3 = T_2[T_1]$
6. $T_4 = addr(B) - 4$
7. $T_5 = T_4[T_1]$
8. $T_6 = T_3 * T_5$
9. $prod = prod + T_6$

* each A_i occupy 4 bytes.

Eg:- $T_2[0]$

↑4 bytes

$T_2[4]$

↑4 bytes

$T_2[8]$

→ $T_2[0]$

↓ base ↳ offset

10. I = I + 1

11. IF (I ≤ 20) GO TO 3

→ divide the pgm into basic blocks & construct the flow graph.

In next page:

→ Basic blocks is a group of statements when entered at the beginning (or) executed in the sequence without halt (or) branch

Basic block

first Identify the leader

1) first stmt is a leader

2) Any stmt target of conditional (or) unconditional jump is a leader.

3) The stmt which immediately follows the unconditional is a leader

Block: (B_1, B_2)

→ Start with the leader and upto (or) but not including the next leader is basic block.

Flowgraph (In next page)

→ These basic blocks are the nodes of the graph. (Flow graph)

$B_1 \rightarrow B_2$

Basic block:

→ A sequence of consequtive stmts which when entered at the begining can executed in sequence without halt can any possibility of branch.

Leader:

→ The first stmt is a leader.

→ Any stmt which is the target of conditional can unconditional goto is a leader.

→ Any stmt which immediately follows the unconditional is a leader.

block: (B_1, B_2 --)

→ Consists of Leader and all the stmts upto and but no including the next leader

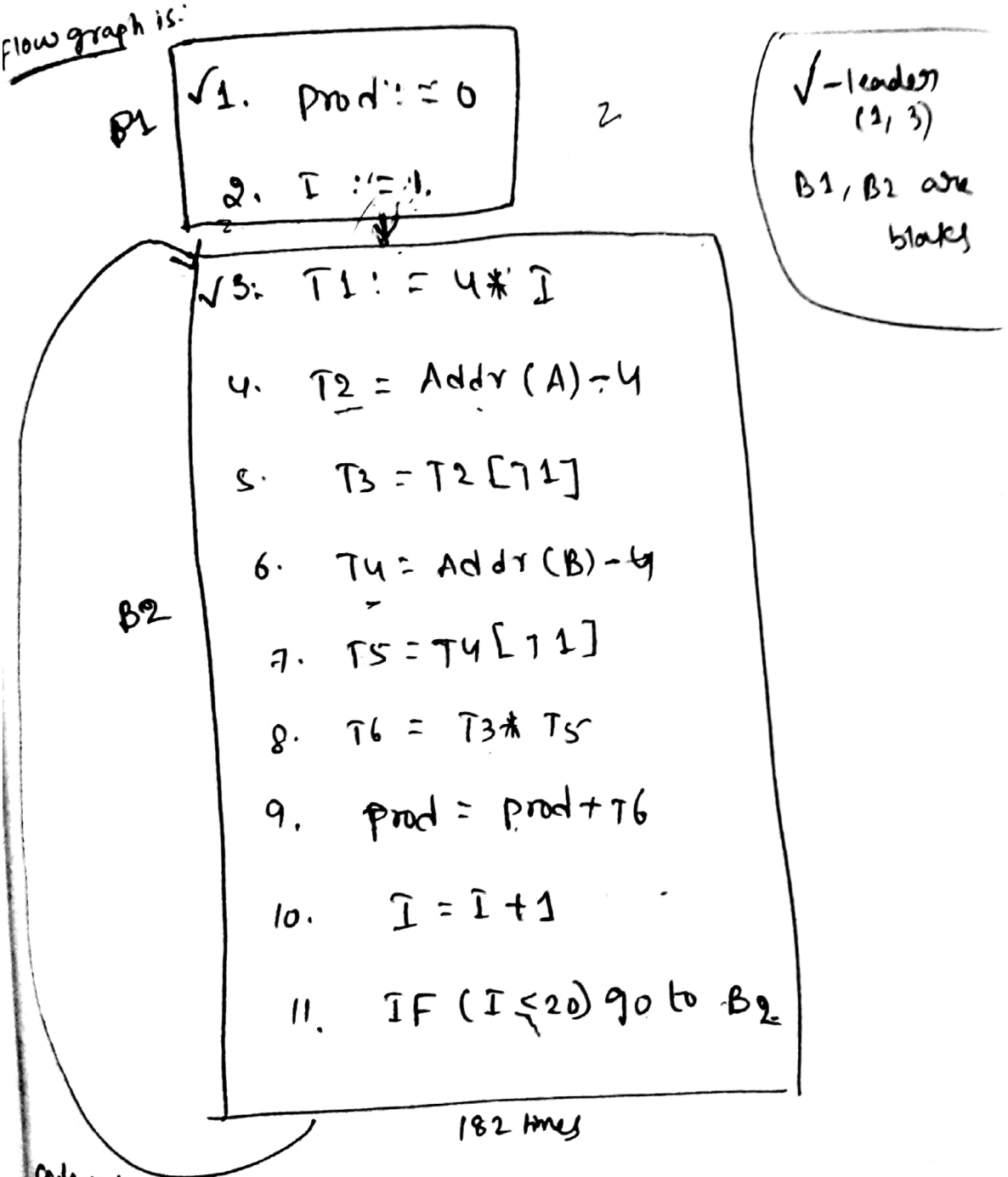
To construct the flow graph:

→ " " " " if there is a con/uncon jump from the last stmt of b_1 for the first stmt of b_2.

$b_1 \rightarrow b_2$

→ b2 imm follows b1 in the order of the pgm and b1 doesn't end in an uncon jump.

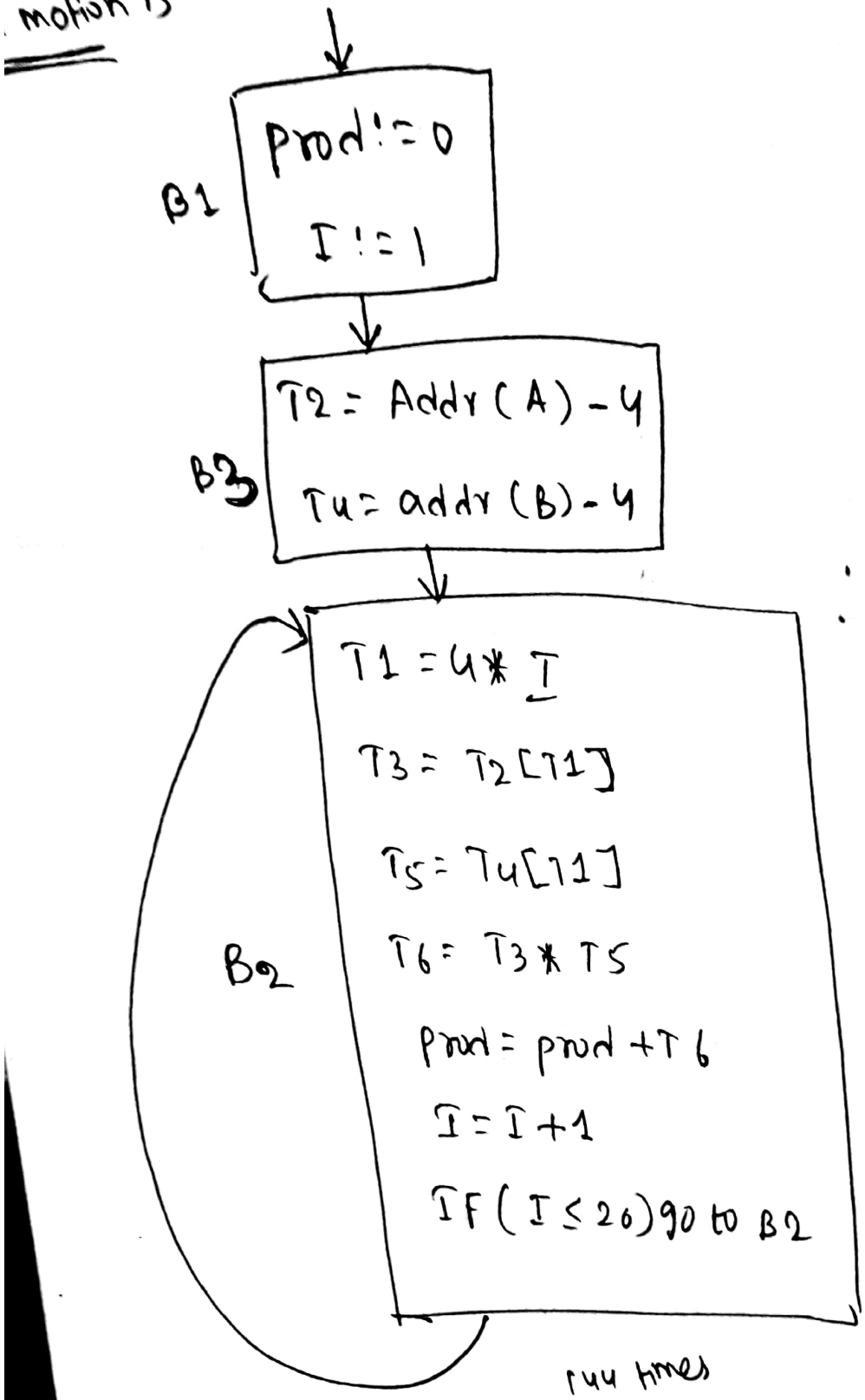
Flow graph is:



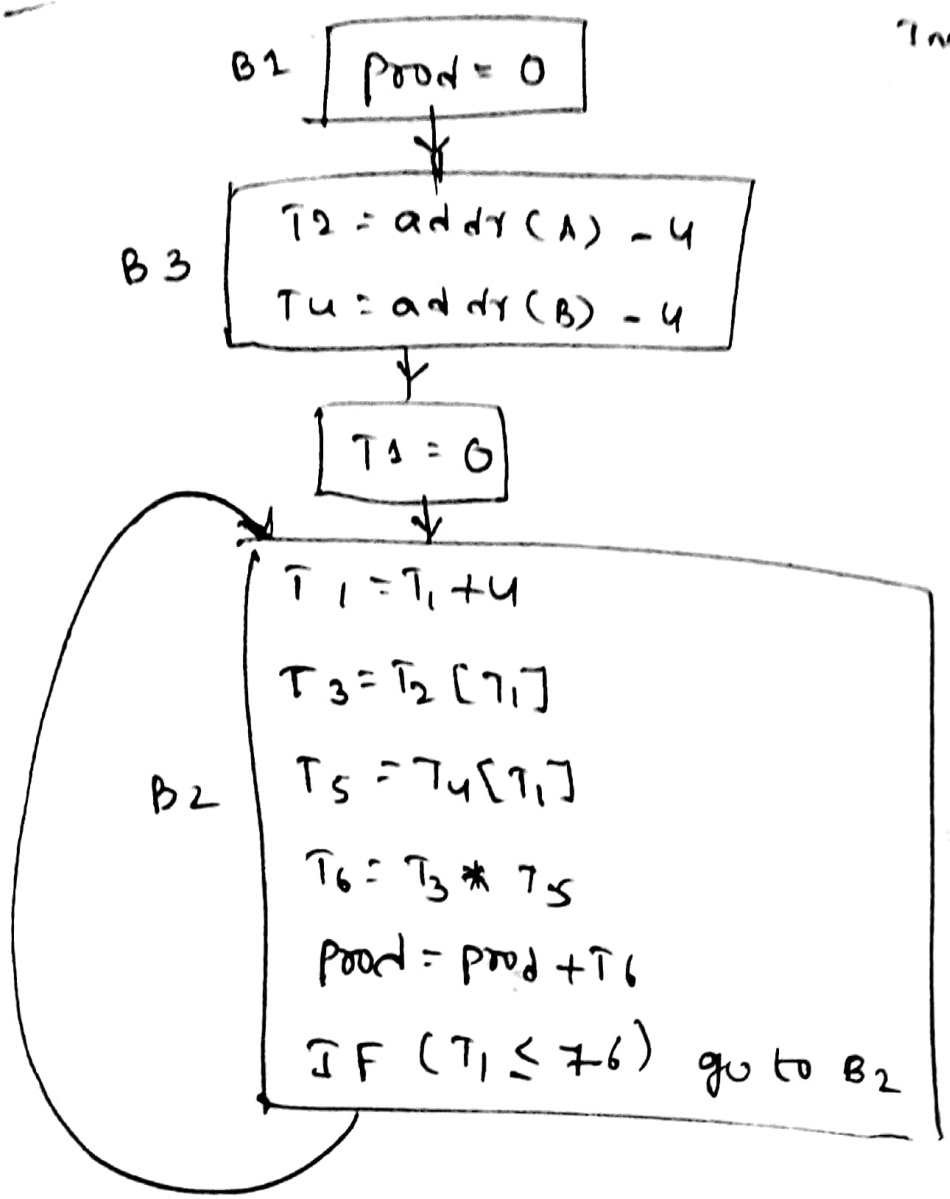
Code motion:

↳ Identify the stmts which have same values ↑ Moving the point into outside the loop is called "code motion".

motion is



Remove I and rewrite the code motion: [which is called as elimination of Induction variables]



Reduction in Strength:-

→ Replace of costlier operation by cheaper operation is called as "Reduction in strength".

DAG (Direct Acyclic Graph)

→ This is also for minimization of code.

13/3/18

DAG: (Direct Acyclic Graph)

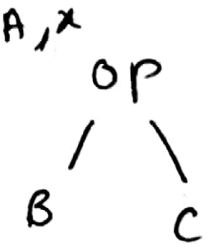
→ used for the representation of basic blocks.

→ " to identify common sub expression and (minimize the code.

→ variant of syntax tree

Eg:- A = B op c

x = B op c



↳ B op c is also existed for A. So write x on beside of A.

Q) 1: $S_1 = 4 * I$

$S_2 = addr(A) - 4$

$S_3 = S_2[S_1]$

$S_4 = 4 * I$

$S_5 = addr(B) - 4$

$S_6 = S_5(S_4)$

$S_7 = S_3 * S_6$

$S_8 = prod + S_7$

$prod = S_8$

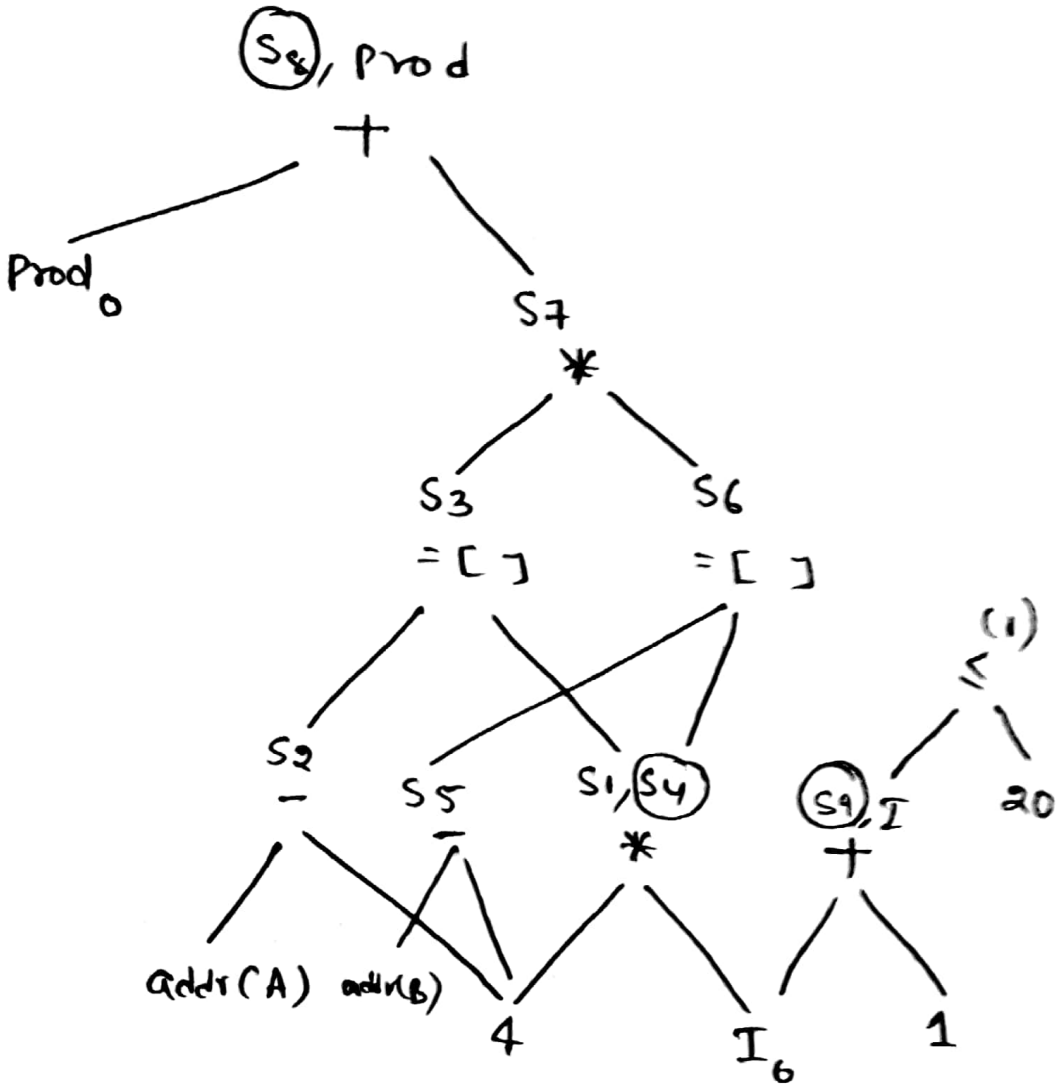
$S_9 = I + 1$

$I = S_9$

if $I \leq 20$ goto (1)

Construct DAG for this

*DAG →



2m

Advantages of DAG:

(1) Identify the common sub expressions and minimize the code.

(2) Gives the list of variables which are used in the block. [leaf nodes]

(3) List of variables which are computed in the block [Inner nodes]

→ We are eliminating S4, S8, and S9 from the code. Now rewrite the code.

1: $S_1 = 4 * I$

$S_2 = addr(A) - 4$

$S_3 = S_2[S_1]$

$S_5 = addr(B) - 4$

$S_6 = S_5[S_1]$

$S_7 = S_3 * S_6$

$prod = prod + 7$

$I = I + 1$

if $I \leq 20$ goto 1

Data Flow Analysis :- (u d chaining)

u → use, d → definition

Reaching definition:

→ If a def reaches a particular point in a pgm then we can replace that ~~occurrence~~ variable with its value of the ~~definition~~ in a pgm and removed the definition is called "reaching definition".

Eg:- ~~A := 3~~ [A := 3 is definition]

!

x = A + 1

Y = A + 5

} Here A will be replace with 3 and we remove A which is first stmt.

IN(B) - Set of all definitions reaching a point just before the first stmt of Blck B.

OUT(B) - Set of definitions reaching the point just after the last stmt of Block B.

GEN(B) - is a set of generated definitions a with in Block B that reach the end of the Block B.

KILL(B) - is a set of Definitions outside of Blk B that define identifiers which have definitions within Block B.

Calculations: [data Flow Equation] (or) [control Flow Eq's]

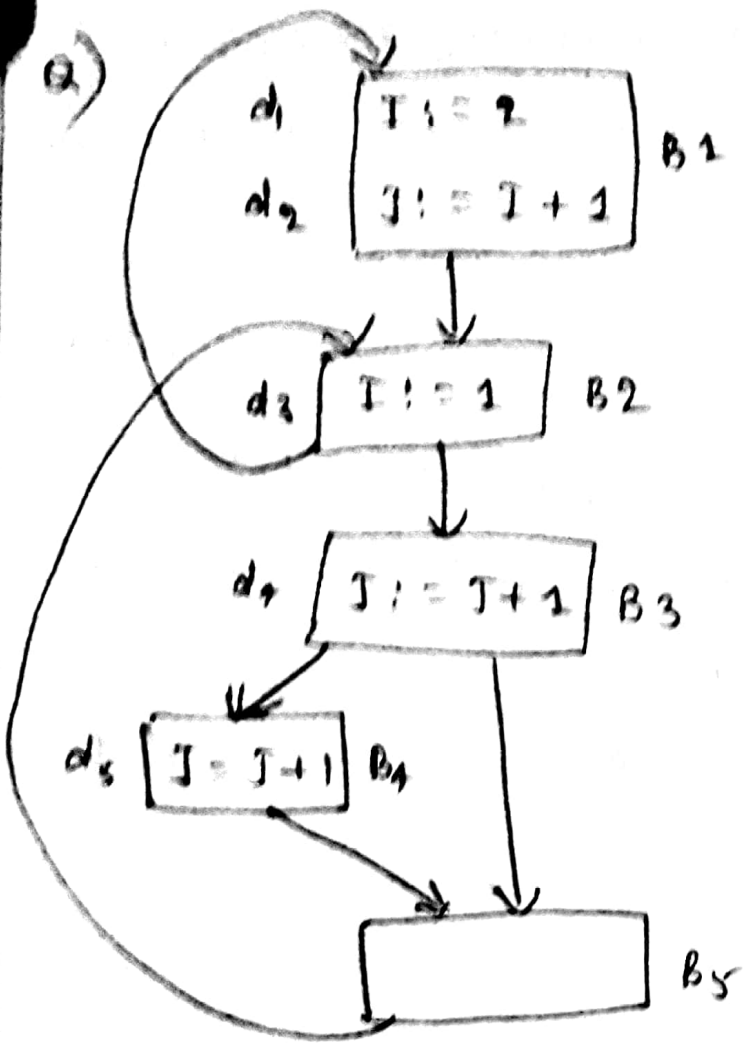
$$IN(B) = \bigcup_{\text{union}} OUT(P)$$

P = predecessors

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

→ These two equations are called as dataflow Equation.

Q)



→ d_1 → definition 1

→ Represent definitions with Bit vector:

$d_1\ d_2\ d_3\ d_4\ d_5$

Block	Gen(B)		KILL(B)	
	The definitions in that block	bit vector		bit vector
B1	{d_1, d_2}	11000	{d_3, d_4, d_5}	00111
B2	{d_3}	00100	{d_1}	10000
B3	{d_4}	00010	{d_2, d_5}	01001
B4	{d_5}	00001	{d_3, d_4}	01010
B5	{}	00000	{}	00000

make d_1 and d_2 values 1 and '0' for remaining

The definitions that are ... another block ...

19/2/18

- It is not easy to calculate the IN(B) and OUT(B) easily for blocks, so that for first pass we consider IN(B) as NULL set

OUT(B) as Gen(B)

↓ def that are generated in Pass(1) Initial

Block	IN(B)	OUT(B)	IN(B) ↑ using formula	OUT(B) ↑ using formula (P)
B1	00000	11000	00100	11000
B2	00000	00100	11000	01100
B3	00000	00010	01100	00110
B4	00000	00001	00110	00101
B5	00000	00000	00111	00111

Pass(2)

$OUT(B_2) = 00100 - (00111) \cup (11000)$

and ↑ not, minus as ↑ not, ∪ ↑ or

$= 00100 \wedge (\neg(00111)) \cup (11000)$

00100
11000
00000 } perform AND
11000 } perform OR
11000

B_2: $11000 - 10000 \cup 00100$

11000
01111
01000
00100
01100

IN(B1) = same IN(B_2) ⇓ The blocks which are coming to B_2 is taken by B_1, B_5 perform OR

OUT(B3) = 01100 − 01001 ∪ 00010

$$\begin{array}{r} 01100 \\ 10110 \\ \hline 00100 \\ 00010 \end{array}$$

$$\boxed{00110}$$

→ The pass(1) and pass(2) values for IN(B) and OUT(B) are not equal. So again do pass(3).

Pass(3)

	IN(B)	OUT(B)
B1	01100	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

IN(B2) is B1 and B5 Perform OR b/w B1 and B5

B1 = 11000 (new value @ current iteration)

B5 = 00111 (Not yet find to take value in previous pass)

OR 11111

OUT(B1) = 01100 − 00111 ∪ 11000

$$\begin{array}{r} 01100 \\ 11000 \\ \hline 01000 \end{array} \qquad \begin{array}{r} 01000 \\ 11000 \\ \hline \end{array} \rightarrow \boxed{11000}$$

Here pass (3) and pass (4) are not same. So do one more pass.

pass(4)	IN(B)	OUT(B)
B1	01111	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

pass(5)	IN(B)	OUT(B)
B1	01111	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

Pass 4 and pass 5 has same values. So, stop the table.

20/3/18

Applications of ud chaining (or Reaching def.)

1. Minimize the code by using definitions like A := 3 by replace A with 3 in all blocks

2. undefined variables.

Eg:- A = 1 (dummy def)

x = A + 1 (when A = 1 which is dummy def reaches x = A+1, then this is known as undefined variables).

(If you have some other value for A in between the terms, then the dummy def will be discarded)

Algorithm for ud chaining: (or data flow eqns.)

For each block B do

begin

IN(B) = ϕ

OUT(B) = GEN(B)

end

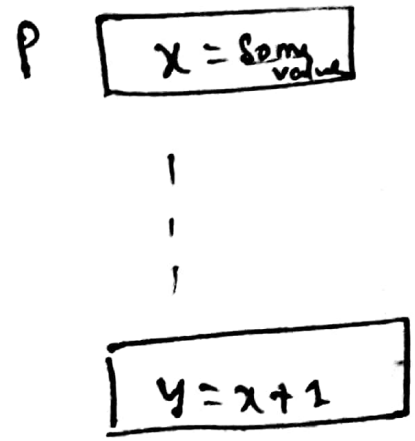
change = true

while change do

```
begin
    for each block B do
    → change = false
    begin
        NewIN = ∪ out(p)
        if NewIN ≠ IN(B) do
        begin
            change = true;
            IN(B) = NewIN( )
            OUT(B) = IN(B) − KILL(B) ∪ GEN(B)
        end
    end
end
```

Live variable Analysis:-

if x at pt p is live at pt p



If the variable at point p is used further in the example then we can say that x is live at pt p otherwise dead at pt p
(Not used further)

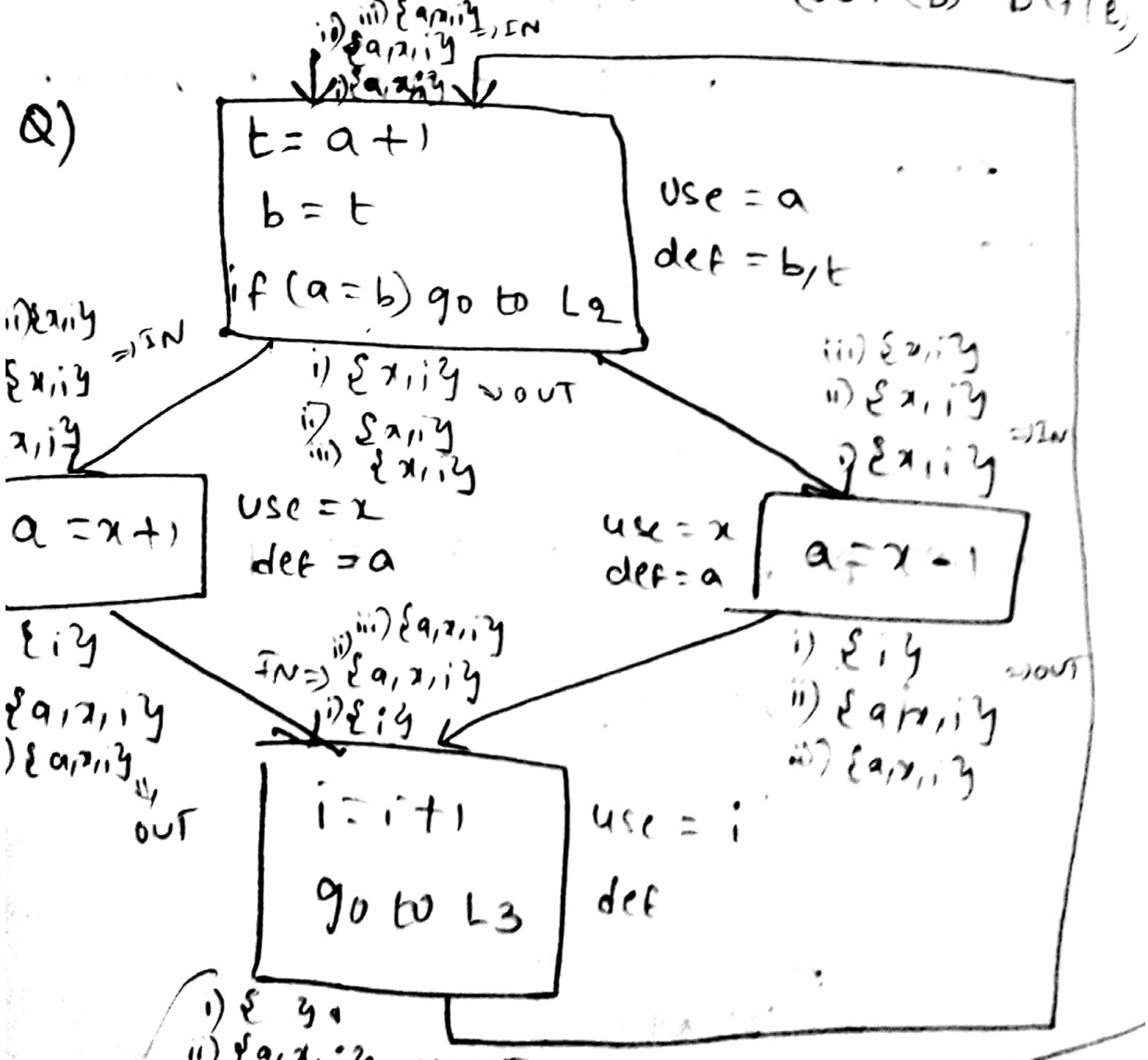
Use - Set of variables used before their definitions.

definition - Set of variables defined before their use.

$$OUT(B) = \cup IN(s)$$

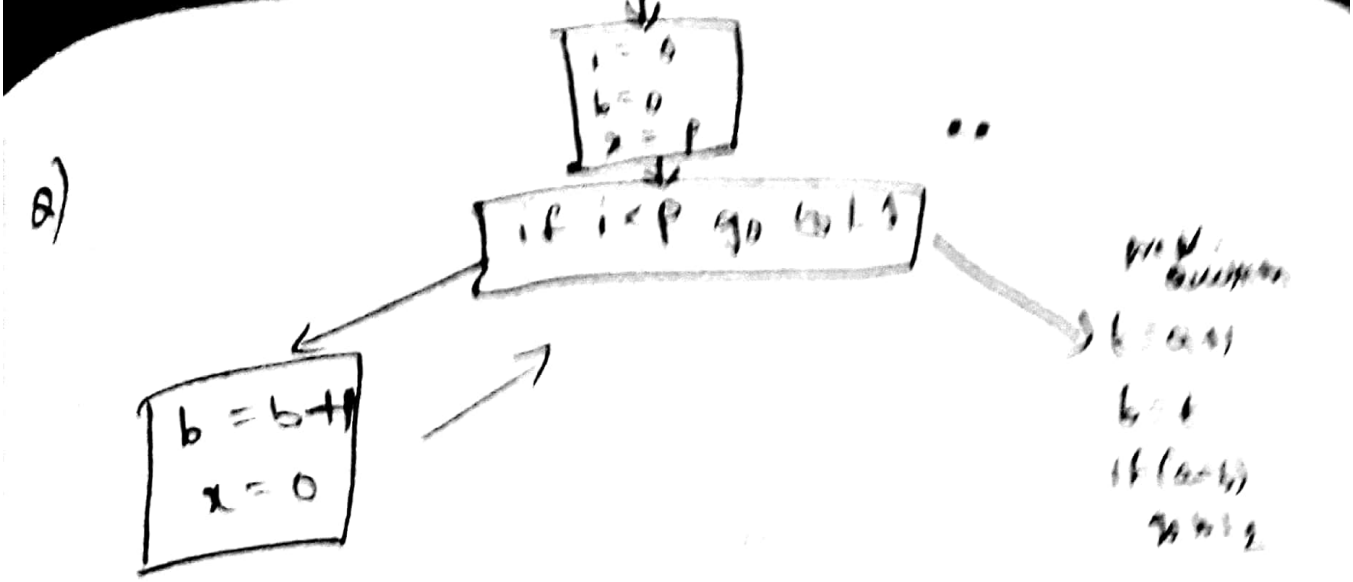
$$IN(B) = USE(B) \cup (OUT(B) - Def(B))$$

Q)

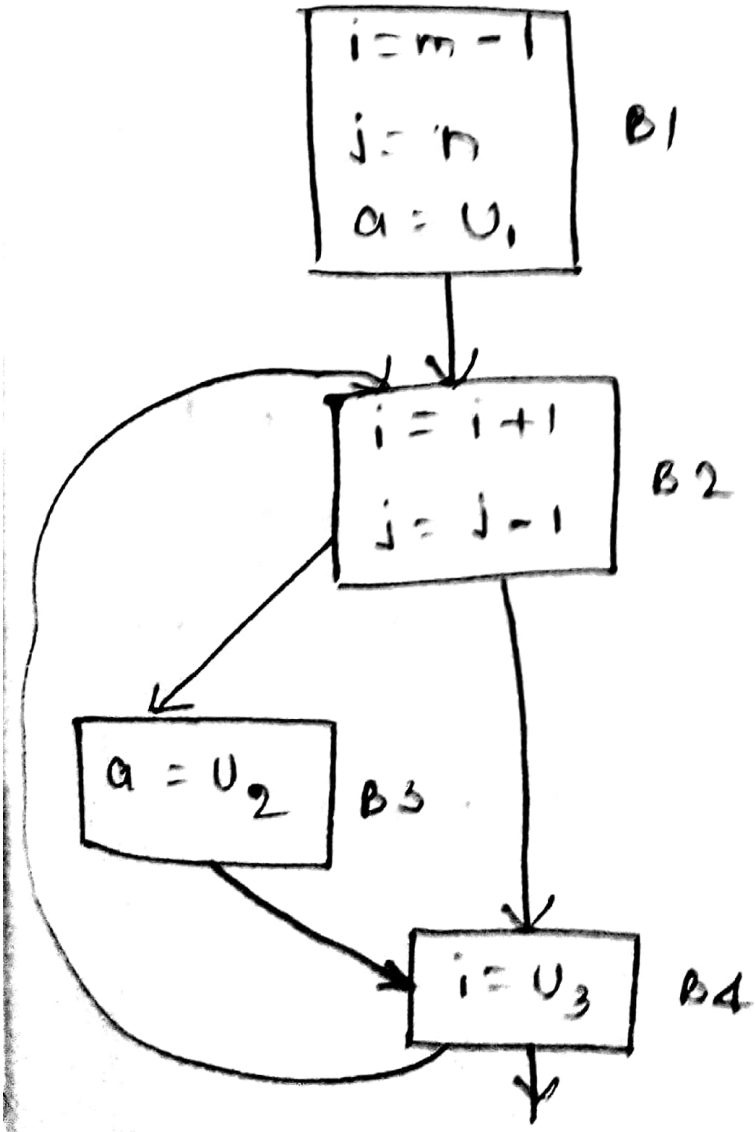


Ans:- Take OUT as null set.

→ do the process until both forms of IN & OUT are equal adjacent



8) Identify the Reaching definitions



22/3/18

** imp question

Code Generation Algorithm:-

I/p is three address code.

o/p is Assembly Language.

→ The ↑ imp for this code generation, (choosing the instruction, Registers and order of evaluation is

(1) Instruction set ⇒ LOAD B
ADD 1
STORE B
} AOS B

(2) Registers

(3) order of Evaluation.

↓↓

same order ⇒ R_1 R_3

8bit new

Reg pair / Reg

R_0, R_1, R_2, R_3 → Quotient

R_0 ↓ Remainder, R_1 ↓ Quotient, R_2 → Remainder

⇒ Even Registers [Remainder is even Registers]

⇒ Odd Registers [containing Quotient]

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

$T = A - B$

$U = A - C$

$V = T + U$

$W = V + U$

A = B op C

MOV B, R_0

op C, R_0

$\Rightarrow R_0 \in R_0 + C$

$T = A - B$	MOV	A, R_0	$R_0 \in A - B$
	SUB	B, R_0	T is in R_0
$U = A - C$	MOV	A, R_1	$R_1 \in A - C$
	SUB	C, R_1	U is in R_1
$V = T + U$	ADD	R_1, R_0	$R_0 = R_0 + R_1$
			V is in R_0
$W = V + U$	ADD	R_1, R_0	
	move	R_0, W	

↳ Here we are not moving T & U because those two variables are already exist in R_1, R_0 [which is concept of live variable].

⇓

Here we are not using T for further uses (or) next uses, so we are not moving it.

→ In this we are using two functions. They are :-

(1) Register descriptor.

(2) Address descriptor.

(1) Register descriptor:

→ A Register descriptor keeps track of what is currently in each Register.

(2) Address descriptor:-

→ Keeps track of the Location where the Current value of the name can be found.

Algorithm for identifying the location of Register (or) GetReg()

2) A := B op C.

The value of this is represented by L

GetReg()

(1) If the value of B is in a register and B holds no other names and B is not live and has no next use after the Execution of B op C then use same register to Store A. Adjust the Address descriptor indicating that B is the no longer in Particular Register R.

(2) Failing (1) Return an empty register.

(3) failing (2) Identify a suitable register move

the value of register to a memory location and return the Register.

(4) If no suitable register is found, then return A to store the result.

Code Generation Algorithm:

(1) Call the function GetReg to identify the Location L where the result A = B op C could be stored.

L is Register or memory location.

(2) Consult the address descriptor for B to determine B', prefer Reg for B'

If B' is different from L then generate MOV B', L

(3) Then Generate OP C', L where C' is the current Location of C, prefer Reg for C' if present in Reg and mov. Adjust the addr descriptor indicating that it is in L'.

(4) If B and C have no next use then adjust the Reg descriptor correspondingly

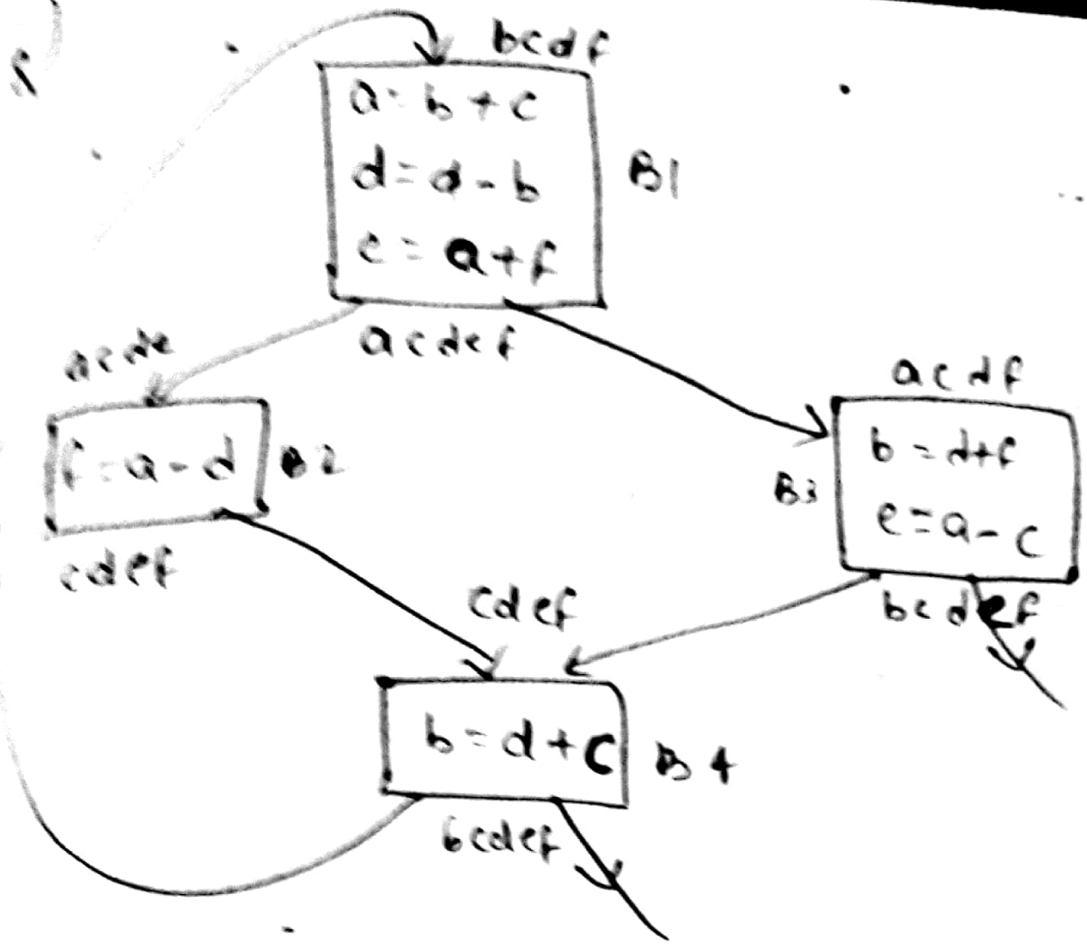
6) $W = (A - B) + (A - C) + (A - C)$

		Reg descriptor	Addr descriptor
T = A-B	MOV A, R_0	~~R_0 contains A~~	
	SUB B, R_0	R_0 Contains T	T is in R_0
U = A-C	MOV A, R_1	~~R_1 contains A~~	
	SUB C, R_1	R_1 contains U	U is in R_1
V = T+U	Add R_1, R_0	R_0 Contains V	V is in R_0
W = V+U	Add R_1, R_0	R_0 Contains W	W is in R_0
	MOV R_0, W		

24|3|18

Register Assignment Global Reg Allocation

→ Next page.



Use (a, B) - No. of times a is used in B before its def

Live (a, B) - 1 or 0

Live on exist from the block and defined in the block otherwise it is "0".

Usage count = use (a, B) + 2 * live (a, B)

	B1		B2		B3		B4	
Variable	use	Live	use	Live	use	Live	use	Live
a	0	1	1	0	1	0	0	0
b	2	0	0	0	0	1	0	1
c	1	1	0	0	1	1	1	0
d	1	1	1	0	1	0	1	0
e	0	1	0	0	0	1	0	0
f	1	0	0	1	1	0	0	0

last col.

	Usage count
a	4
b	6
c	3
d	6
e	4
f	4

R_0, R_1, R_2, R_3

→ If you have only 4 registers then store max of usage count for first three registers (R_0, R_1, R_2) and R_3 is for temporary storage location.

a b c d e f

4 6 3 6 4 4

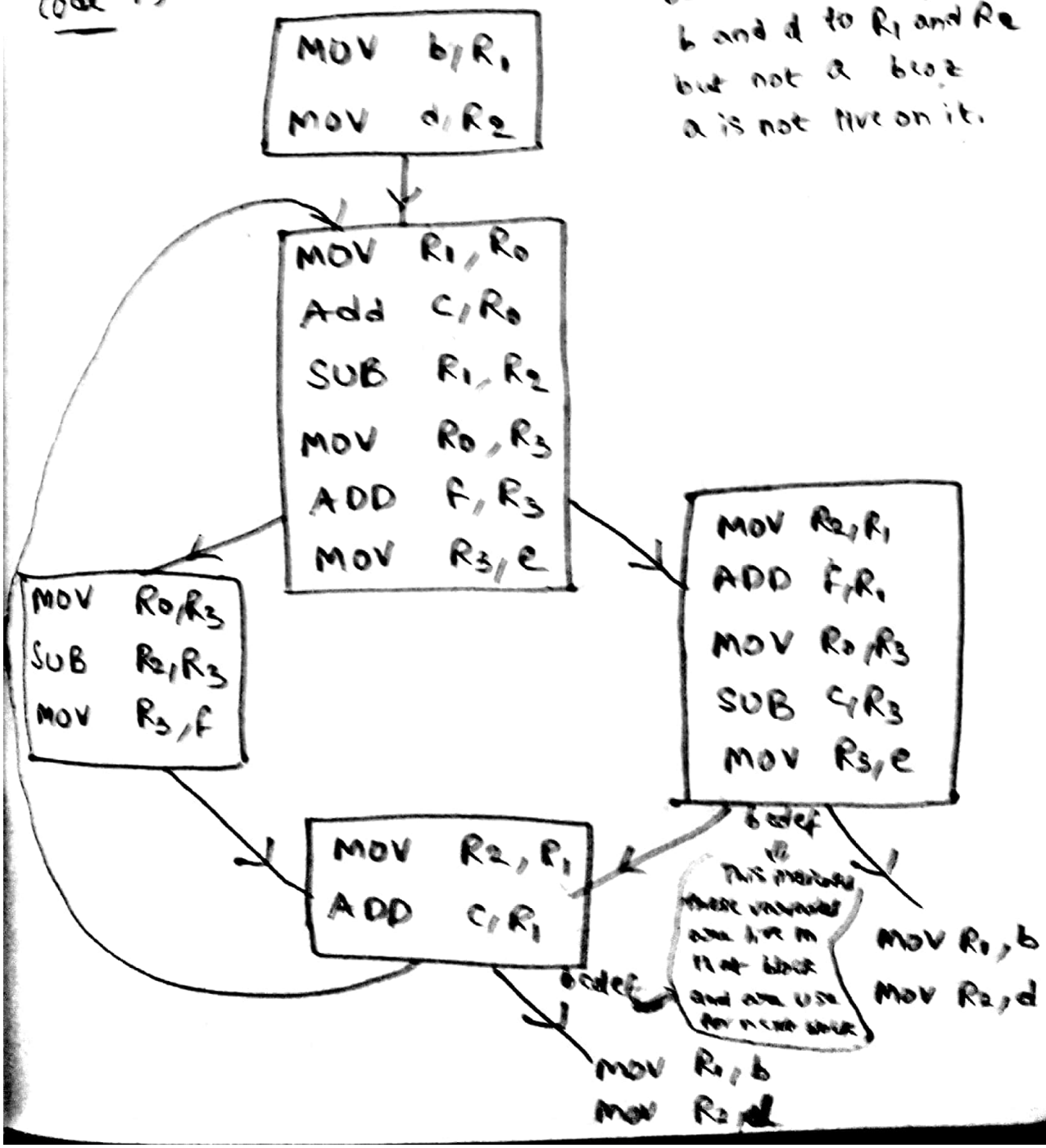
$R_0 R_1 R_2 R_3$

a = R_0

b = R_1

d = R_2

code is



We are moving only b and d to R_1 and R_2 but not a bcoz a is not live on it.

Peep hole optimization:

→ Optimization is performed by looking small piece of code.

→ There are 4 techniques. They are:

(1) Redundant instruction elimination:

mov R0, a ⇒ copy stmt which moves R0 → a

mov a, R0

In this case you eliminate mov R0, a

→ If you have Label for the instruction then don't eliminate it.

```
    mov  R0, a
L1: mov  a, R0
    :
    :
    Goto L1
```

(2) Control flow optimization:

(1) Any unlabelled stmt after an unconditional GOTO may be removed.

(2) (dead code elimination) ⇒ The stmts will never executed.

Eg: if debug == 1 go to L1
go to L2
L1: print ----
L2:

After changing:

if debug != 1 go to L2
print: - - -
L2:

Eg: go to L1
L1: Go to L2

After changing:

go to L2
L1: Go to L2

Here we are not removing L1 bcoz there may be a chance of other instruction for L1:

Eg: L1: if a<b go to L2
L3:

def: A variable is live at a point in a pgm if its value can be used subsequently in the pgm otherwise it is dead at that point.

after changing:

if a<b go to L2

Go to L3

L3: —

(3) Algebric Simplification:

$x = x + 0$

$x = x * 1$

Use of machine idioms

$x = x + 1$ } Auto increment

$x = x - 1$ } Auto decrement

(4) Common Sub expression elimination:

$X = A + B + C$

$Y = A + B + D$

$T_1 = A + B$

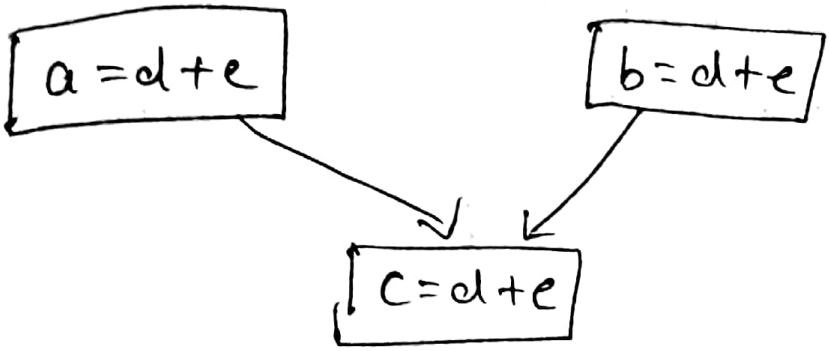
$X = T_1 + C$

$Y = T_1 + D$

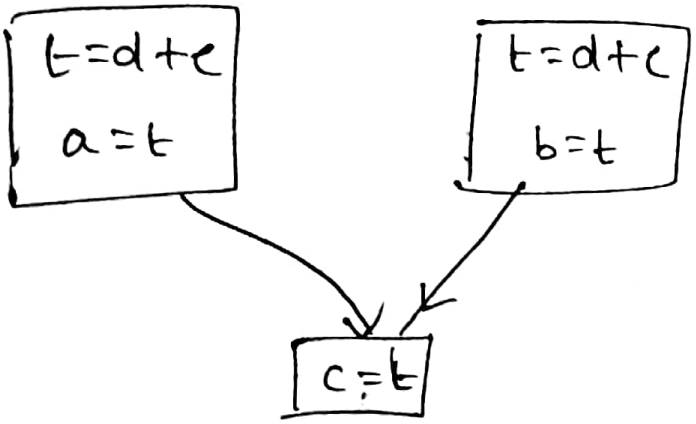
(5) Copy Propogation: [These 5 and 6 techniques are also come under peephole optimization]

→ same as subexpression little bit changes.

Eg:



Soln:



(6) Constant folding:

→ Evaluate the constant expressions at compile time and replace constant expresin by their value.

Eg: $x = 2 * 3.14$

$x = 6.28$

Available expression:

An expression x+y is available at point p if every path from entry node to p evaluates x+y and after the last evaluation prior to reaching p there are no subsequent assignment to x or y.

Eg:-

$a = b + c$ — ϕ

$b + c$

$b = a - d$

$a - d$

$c = b + c$

$a - d$

$d = a - d$

ϕ