Unit-V

Code Generation

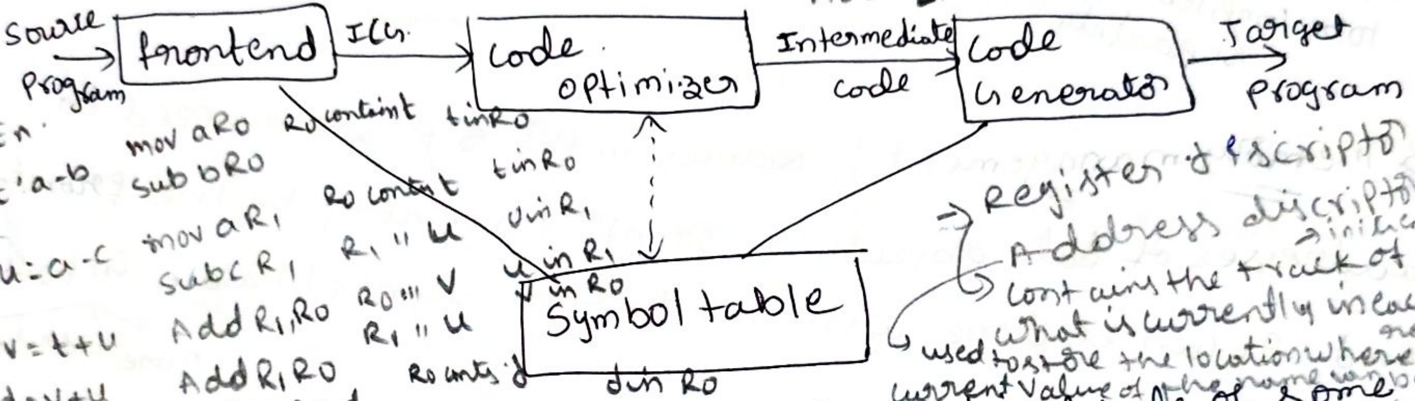
→ The final phase in our compiler model is the code generator. It takes as i/p an intermediate representation of the source program & produce as o/p an equivalent target program (Assembly)

→ The requirements traditionally imposed on a code generator are

~~sec~~ The o/p code must be correct & of high quality.

* meaning that it should make effective use of the resources of the target machine. | It should carry the exact meaning of the source code.
  ↪ It should carry the exact meaning of the SC.
  ↪ It should be efficient in terms of CPU usage & memory management.
* the code generator itself should run efficiently. (using descriptors)

Ex:- x:=y+z   mov y, R0   mov R0, x
                       add z, R0



Ex:

t:a-b	mov a R0	R0 contains t	t in R0
	sub b R0		t in R0
u:=a-c	mov a R1	R0 contains t	u in R1
	sub c R1	R1 " u	u in R1
v:=t+u	Add R1, R0	R0 " v	v in R0
		R1 " u	
d:=v+u	Add R1 R0	R0 contains d	d in R0
	mov R0, d		

→ Register descriptor → contains the track of what is currently in each reg → initially empty
→ Address descriptor ↪ used to store the location where current value of the name can be found during run time

→ the code generator by the compiler is an object code of some lower-level programming language, for Ex: Assembly language.

Issues in the design of code generator:-

→ The important criterion is that it produce correct code. The design goals are. * correctness * easily implemented * tested & maintained

* In the code generation phase, various issues can arises.

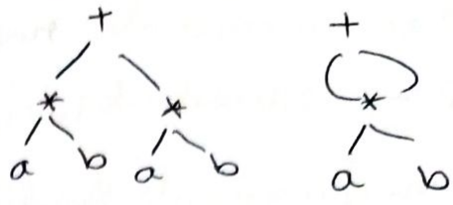
1. Input to the code Generator:-

It consists of the intermediate representation of the source program produced by front end, together with info in the S.T to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

→ Intermediate representation can be

1. Postfix ab+
2. 3-address code { triple, quadruple, Indirect triple
3. syntax tree / dags.

a*b+a*b



* frontend produces low-level intermediate representation directly. manipulated by the machine instructions.

* The code generator phase needs complete error free Intermediate code as an i/p.

2. Target Program :- The o/p of the CG is the target program

The o/p may be
directly placed in a fixed memory ← * Absolute machine ~~code~~ Language } .exe
It allows subprograms to be compiled separately ← * Relocatable machine Language , .obj's (Linker)
* Assembly Language, .asm (loader)
Nemonics

3. memory management :- Names in the S.P are mapped to addresses of data objects in runtime memory by the front-end. It can be done with the help of symbol table info's
name address type

**
4. Instruction selection :

→ The instruction set of the target machine should be complete & uniform.

→ when you consider the efficiency of target machine then the instruction speed & machine idioms are important factors

→ The quality of the generated code can be determined by its speed & size.

Ex1:- a:= b+c

```
MOV b,R0 .   R0→b
ADD c,R0     R0+C0
MOV R0,a     a→R0
```

Ex2: x:=x+1

```
        MOV x,R0
INC x { ADD #1,R0
        MOV R0,x.
```

Ex3:- x=y+z, w=x+s

```
MOV y,R0
ADD z,R0
MOV R0,x
MOV x,R0   This can be eliminated
ADD s,R0
MOV R0,w
```

5. Register allocation :- Instructions involving register operands are shorter & faster than those involving operands in memory.

⇒ The following subproblems arise when we use registers.

* Register allocation : set of var's that reside in registers/ (allocation of the memory to the variables)

* Register Assignment :- The specific register that a variable will reside is selected | we have to select the specific register for a particular variable (Assigns the values to the variable)

Ex :- $t = x + y$		MOV x, R0	R0 → x	Here we didn't allocate
$t = t * z$	⇒	ADD y, R0	R0 + y	⇒ Reg. for temp. variable
$t = t / w$		MUL z, R0	R0 * z	&
		DIV w, R0	R0 / w	R0 is a single reg is
		mov R0, t	t → R0	used to perform all operations.

6. Evaluation order :- The code generator decides the order in which the instruction will be executed. The order in which computations are performed can effect the efficiency of the target code.

Ex :- $a + b - \overset{t_2 *}{(c + d)} * e$

Three-address code	Code	Reordered Three-address code	Code	
$t_1 := a + b$	1. MOV a, R0	$t_2 := c + d$	1. MOV c, R0	The reorded
$t_2 := c + d$	2. ADD b, R0	$t_3 := t_2 * e$	2. ADD d, R0	Inst's reduced
$t_3 := t_2 * e$	3. MOV R0, t1 R0 free	$t_1 := a + b$	3. MOV e, R1	The no. of
$t_4 := t_1 - t_3$	4. MOV c, R1	$t_4 := t_1 - t_3$	4. MUL R0, R1	Final code by
	5. ADD d, R1		5. MOV a, R0	2. Thus
	6. MOV e, R0		6. ADD b, R0	saved in
	7. MUL R1, R0 mul: R0		7. SUB R1, R0	cost.
	8. MOV t1 [illegible] R[illegible]		8. MOV R0, t4	
	9. SUB R0, R1			
	10. MOV R4, t4			

Generic code Generation Algorithm :-

It generates target code for a sequence of instructions.

→ code generation algorithm considers each three-address instruction in turn and decides what loads are necessary to get the needed operands into registers.

→ After generating the loads, it generates the operation itself. then, if there is a need to store the result into a memory location it also generates that store.

→ The code generator has to track both the registers & addresses while generating the code. for both availability & location of values the following two descriptors are used.

1. Register descriptor:- for each available register, a register descriptor keeps track of the variable names whose current value is in that register. initially all registers are empty. These registers are used for local use within a basic block.

2. Address descriptor:- for each program variable, an address descriptor track of the location where the current value of the name can be found. It may be a register or stack location memory address.

Ex:- $d = (a-b) + (a-c) + (a-c)$

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

$t_4 = 4 * i$

$t_4 = 4 * i$ $t_4 = t_4 + 4$

$4i$

$j = 1$
$j = j - 1$
$j = 1$
$j = 0$
$t_4 = 0$

$x = 2 * a$

$x = a + a$

Statements	Code Generator	Reg. Descriptor	Address descrip
$t_1 = a - b$	MOV a, R0 Sub b, R0	Reg's are empty R0 contains t_1	t_1 in R0
$t_2 = a - c$	MOV a, R1 Sub c, R1	R0 contains t_1 R1 contains t_2	t_1 in R0 t_2 in R1
$t_3 = t_1 + t_2$	ADD R1, R0	R0 contains t_3 R1 contains t_2	t_2 in R1 t_3 in R0
$d = t_3 + t_2$	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 & memory

~~Generating code from DAG:~~

Code Generation using DAG:-

→ Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too.

* leaf nodes represent identifiers, names or constants.
* Interior nodes represent operators.
* Interior nodes also represent the results of Expressions

→ Generating code from DAG is much simpler than the linear seq of three addr code

→ methods generating code from DAG's are :

→ using DAG we can rearrange some sequence of instructions & generate an efficient

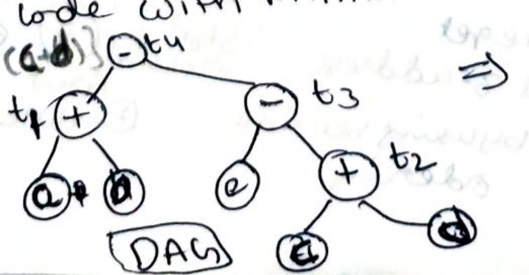
1. Rearranging order. 2. Heuristic ordering 3. Labeling algorithm

1. Rearranging order: The order of 3-address code effects the cost of the object code being generated.

→ By changing the order in which computations are done we can obtain the object code with minimum cost.

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

$(a+b) - [e - (c+d)]$



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

10	8
MOV a, R0	MOV c, R0
ADD b, R0	ADD d, R0
MOV c, R1	MOV e, R1
ADD d, R1 ⟹	SUB R0, R1
MOV R0, t1	MOV a, R0
MOV e, R0	ADD b, R0
SUB R1, R0	SUB R1, R0
MOV t1, R1	MOV R0, t4
SUB R0, R1	Ass. code
MOV R1, t4	

Simple . . .

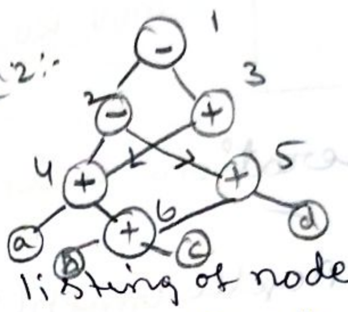
+euristic ordering :-

obtain all the interior nodes. consider these interior nodes as unlisted nodes.

Algorithm:-

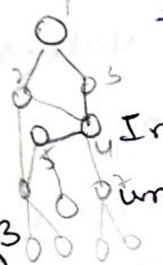
```
Algm H-DAG()
while(unlisted interior nodes remain) do
{
    Pickup an unlisted node n, whose parents have been listed or root.
    list n;
    while ( the leftmost child m of 'n' has no unlisted parent
                                          AND is not leaf)
    {
        list m;
        n:= m;
    }
}
order = reverse of the order of listing of nodes;
}
```

Ex 2:-



$t_6 = b+c$
$t_5 = t_6+b$
$t_4 = a+t_6$
$t_3 = t_4+t_5$
$t_2 = t_4-t_5$
$t_1 = t_2-t_3$

→ Given the DAG first numbered from top to bottom & from left to right then consider the unlisted interior nodes.



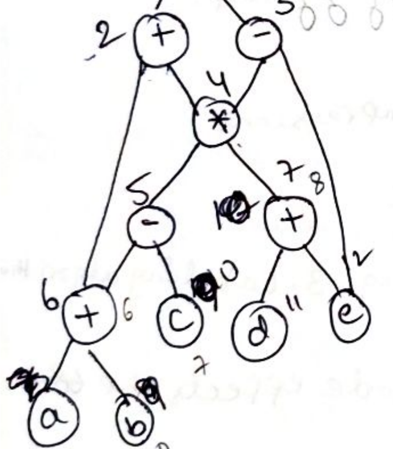
Interior nodes = 1, 2, 3, 4, 5, 6, 7

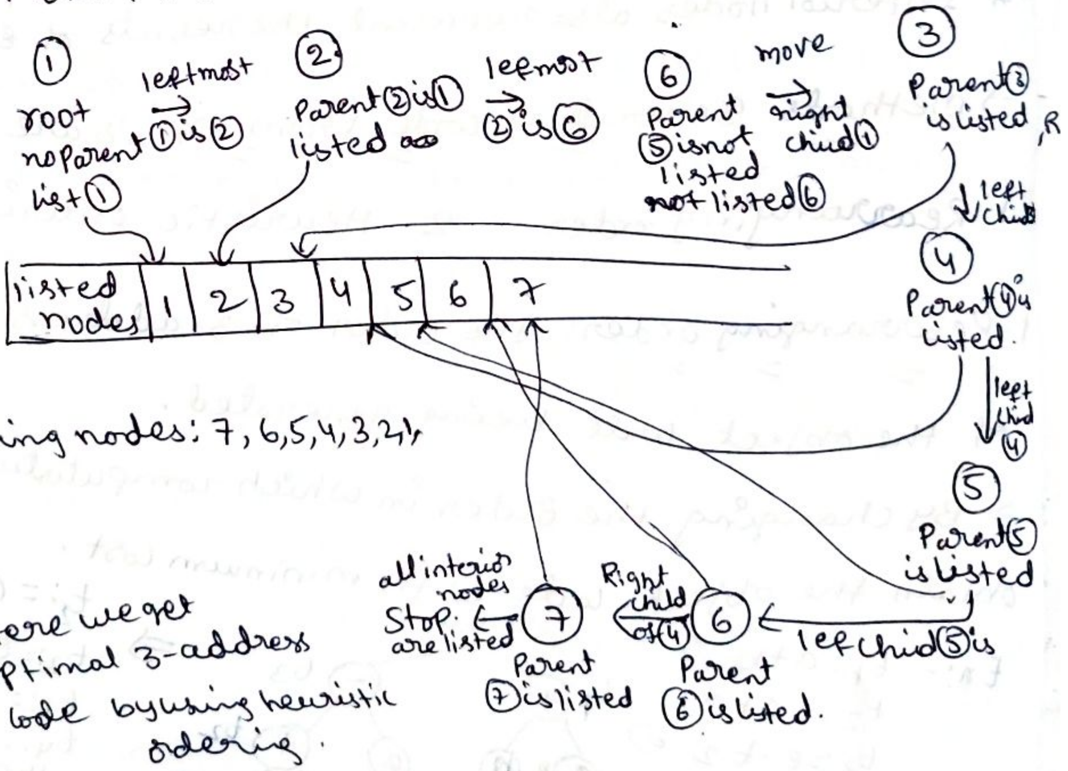
unlisted nodes = 1, 2, 3, 4, 5, 6, 7.

⇓

Pickup an unlisted node, whose parents have been listed

Ex:-





Reverse order of listing nodes: 7, 6, 5, 4, 3, 2, 1

$t_7 = d+e$
$t_6 = a+b$
$t_5 = t_6 - c$
$t_4 = t_5 * t_7$
$t_3 = t_4 - e$
$t_2 = t_6 + t_4$
$t_1 = t_2 * t_3$

Here we get optimal 3-address code by using heuristic ordering.

3. Labeling Algorithm:- The labeling algorithm generates the optimal code for given Expression in which minimum registers are required.

→ In this labelling algm is used to find the min. no. of reg used for Executing the code

→ using labeling algorithm the labeling can be done to tree by visiting nodes in bottom up order. (leafnode → interior → rootnode)

→ for computing the label at node 'n' with the label L_1 to left child then the value is '1'. and label L_2 to right child then the value is '0'.

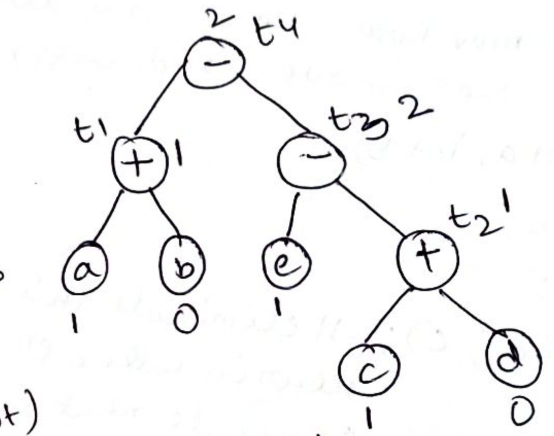
$L_1 \neq L_2$

Label (n) = max (L_1, L_2) if L_1 not equal to L_2

Label (n) = $L_1 + 1$ if $L_1 = L_2$

3-addresscode

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



⇒

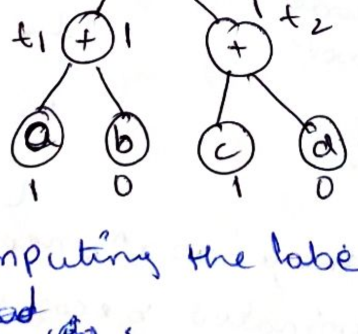
```
MOV a, R0
ADD b, R0
MOV R0, t1
MOV c, R1
ADD d, R1
MOV R1, t2
MOV e, R0
SUB R0, R1
ADD R1, R0
SUB R0, R1
MOV R1, t4.
```

two registers R0, R1

(L-R-Root)
Postordertraversal : a b t_1 e c d t_2 t_3 t_4

Eg2:- $t_1 = a+b$
$t_2 = c+d$
$t_3 = t_1 + t_2$

=



⇒

```
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
ADD R0, R1
MOV R1, t3
```

Algorithm:- for computing the label at node n

1. If n is a leaf ~~then~~ and
2. If n is the leftmost child of its Parent then label(n) := 1
3. else lable (n) := 0. else begin /* n is an interior node */

So Label (n_1) ≥ Label (n_2) ≥ . . . ≥ label (n_k);

4. label(n) := max (label(n_i) + i - 1) where 1 < i < K.

end

Peephole optimization technique:-

→ This technique works locally on source code to transform it into an optimized code.

→ It is a short seq of target instruction that can be replaced by shorter or faster seq. instructions in a window (Peephole).

→ This tech is applied to improve the performance of prgm by Examining a short seq of inst in a peep hole.

Characteristics of Peephole optimization :-

1. Redundant instruction elimination.
2. Removal of unreachable code
3. flow of control optimization
4. Algebraic simplifications
5. machine idioms. (Ex : a=a+1 → INC a, a=a-1 → DEC a)

→ Ex1:- mov R0, a } mov R0,a Here a is already in Register R0
mov a, R0 so remove 2nd instruction & consider one.

→ Eg2:-

```
int add (int a, int b)
{ int c=a+b;
  return c;
  Print f("%d", c);  // eliminate this inst. becoz once return
}
```

occur in called fn control moves to calling fn & Execute next stmt in calling fn. so it is not a optimized code.

→ Ex3:- Here using peephole optimization unnessary Jumps can be eliminated

```
goto L1                                            goto L3
L1: goto L2      multiple Jumps can                L1 : goto L3
                 ⟹ make the code
L2: goto L3      inefficient above                 L2 : goto L3
                 code can be
L3: mov a,R0     replaced by                       L3 : mov a, R0
```

Ex4:- $x = x+0$ (or) $x = x*1$ ~~[illegible]~~, $a = x^2$ replaced with $a = x*x$

the above stmts can be eliminated becoz by Executing those stmts the result 'x' won't changes.

Ex5:- It is the Process of using powerful features of CPU inst

Ex:- a=a+1 → Inc a
a=a-1 → Dec a

Simple code Generator (or) Generic code generation algorithm:-

It generates target code for a sequence of instructions.

→ It uses a function getReg() to assign registers to variables.

→ For each operator in a three address code there is a correspond target language operator (+, -, * etc)

→ It uses 2 data structures. 1. Register descriptor
2. Address descriptor

Register descriptor (RD):- used to keep track of which variable is stored in a register. Initially all registers are empty. for Example if we are using R_0 & R_1 for storing the values, then register descriptor tells what is currently in R_0, R_1.

→ RD maintain the value stored in register & inform the availability of registers to code generator.

Address descriptor:- used to keep track of location where the variable is stored. location may be register, memory address, sta etc.

→ It maintain the memory location which are used in program

Algorithm:- [used to generate code for the single basic block]

For Each three address code of the form $x = y \text{ OP } z$, & Assumes that 'L' is the location where the o/p of y OP z is to be stored.

1. call function getReg() to gets the location of L.
2. Determine the present location of 'y' by consulting address descriptor of y. If 'y' is not present in location 'L' then generate the instruction mov y, L to copy value of y to L.
3. The present location of z is determined using step 2 and the instr is generated as OP z, L. → update address descriptor of 'x' to indicate that it is stored in L; if so,
4. Now 'L' contains the value of y OP z i.e assigned to 'x'. if y & z have no next uses and not live on Exit, update the descriptors to remove y & z.

→ Ex:- d = (a-b) + (a-c) + (a-c) → simple code Generation of code generation

three address code : $t_1 = a-b$
$t_2 = a-c$
$t_3 = t_1 + t_2$
$d = t_3 + t_2$

Statement	Code Generation	Register descriptor	Address descriptor
$t_1 = a-b$	mov a, R0 sub b, R0	R0 contains t_1	t_1 in R0
$t_2 = a-c$	mov a, R1 sub c, R1	R0 contains t_1 R1 contains t_2	t_1 in R0 t_2 in R1
$t_3 = t_1 + t_2$	Add R1, R0	R0 contains t_3 R1 contains t_2	t_3 in R0 t_2 in R1
$d = t_3 + t_2$	add R1, R0 ~~mov R0, d~~	R0 contains d R1 contains t_2	d in R0 ↳ final result

In this ~~getreg~~ getReg is used to select register for each memory for each memory location associated with the three address stmts. This function decides the status of register & location of value name.

→ Basic block is a collection of 3 address stmts, keeps track of register allocation & assignment.

* Register allocation :- is a process of deciding what value a register must hold.
* Register Assignment is a process of picking up a register to store a value of variable.

Operations of getReg():- There are 3 types of operations

1. Load operations:- LD r, x loads value in location x in register r.
2. Store operations :- ST x, r stores value in register 'r' in location x.
3. computation Operations:- Add, Sub, mul, div, Inc etc.

Global register allocation:-

1. The code generation algorithm used registers to hold values for the duration of a singleblock. however all the live variables were stored at the end of each block.

2. To save some of these stores and corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers ~~registers~~ consistent across block boundries (globally).

3. Since programs spend most of their time inner loops, a natural approach to global register assignment is try to keep a frequently used value in a fixed register throught a loop.

4. one strategy for global register allocation is to assign some fixed no. of registers to hold the most active values in each inner loop.

usage count:- count a savings of one for each use of x in loop L.

→ if x is allocated a register, then count a savings of two for each block L.

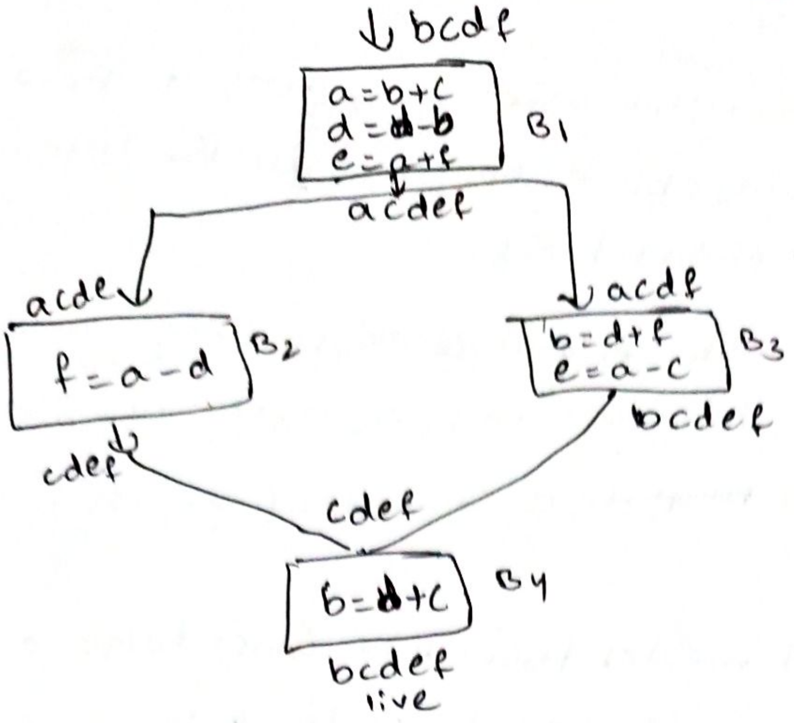
→ formula for the benefit of 'x' allocated to a reg. to be realised from allocating a register to 'x' within loop is:

$$\sum_{\text{blocks } B \text{ in } L} [use(x,B) + 2 * live(x,B)]$$

where $use(x,B)$ is the no. of times x is used in B prior to any definition of x. and is not preceded by an assignment to x in the same block.

$live(x,B) = 1$, if x is live on exit from B and assigned a value in B. $live(x,B) = 0$, otherwise.

En:-



B1 B2 B3 B4 save units of cost

$a = (0 + 2*1 = 2) + (1 + 2*0 \underset{=1}{)} + (1 + 2*0 \underset{=1}{)} + (0 + 0) = 4$

$b = (2 + 2*0 = 2) + (0 + 2*0 = 0) + (0 + 2*1 = 2) + (0 + 2*1 = 2) = 6$

$c = (1 + 2*0 = 1) + (0 + 2*0 = 0) + (1 + 2*0 = 1) + (1 + 2*0 = 1) = 3$

$d = (0 + 2*1 = 2) + (1 + 2*0 = 1) + (1 + 2*0 \underset{=1}{)} + (1 + 2*0 = 1) = 5$

$e = (0 + 2*1 = 2) + (0 + 2*0 = 0) + (0 + 2*1 = 2) + (0 + 2*0 = 0) = 4$

$f = (1 + 2*0 = 1) + (0 + 2*1 = 2) + (1 + 2*0 = 1) + (0 + 2*0 = 0) = 4$

$$\begin{array}{lll} a\ R_0 & \overset{d}{R_1} & \overset{e}{R_2} \\ \uparrow & \uparrow & \uparrow \\ a & b & d \\ e\ or\ f & & \end{array}$$

a, b, d higher units to save so here fix $R_0, R_1, \& R_2$ Registers

→ c, e, f are used to use any registers R_0, R_1, R_2