① Explain the design issues of code generation.

The issues of code generation are as follows :-

(i) Input to code generator :-

∘ The input to the code generator is an intermediate representation of source program produced by the front end and the information stored in the symbol table.

∘ The information is used to determine run-time addresses of data objects denoted by names in IR.

There are different kinds of IR :-

a) Three address representations :- Quadraples, Triples, Indirect triples

b) Virtual machine representations :- bytecode, stack machine code

c) linear representation :- Postfix notation

d) Graphical representation :- Syntax Tree, DAG's.

∘ The code generator assumes that the input is free from all kinds of errors.

(ii) Target Program:-

- The output of code generator is target program which has a significant impact.
- The target machine also has a significant impact on the difficulty of constructing a good code generator that produces high quality machine code.
- The common target machine architecture are RISC, CISC and stack based where each of them pose their own difficulty.
- There are variety of output possible but Absolute machine language has an advantage that it can be placed in a fixed location in memory and immediately executed.
- Relocatable machine language program allows sub programs to be compiled seperately.
- Producing Assembly language program makes job of code generator easier.

(iii) Instruction selection :-

- The code generator must map the IR program into a code sequence that can be executed by the target machine.
The complexity of this is determined by :-
  - Level of IR
  - The nature of instruction-set architecture
  - Desired quality of generated code.

- If IR is high level :- poor code, low effecient code
- Nature of Instruction set :- Set of target machine has a strong effort on difficulty of instruction selection
- Uniformity and completeness of instruction set are important factor.

(iv) Register Allocation :-

- Instruction involving register operands are usually shorter and faster than those involving memory.
- Efficient utilization of limited set of registers is important to generate good code.

:) The use of register is subdivided into two problems :-

- Register allocation, during which we select the set of variables that will reside in registers at each point of program
- Register assigment, during which we pick specific register that variable will reside in.

(V) Evaluation Order :-

:) The order in which computations are performed can affect the effeciency of target code.

:) Picking a best order in general case is a difficult NP- Complete Problem.

:) When instructions are independent, their evaluation order can be changed to utilize registers and save on instruction cost.

② Write short notes on the following with respect to optimization of basic block:-

(i) Local Common Subexpression

(ii) Dead code Elimination

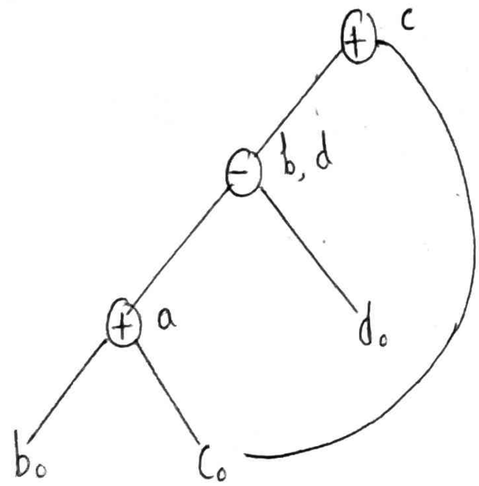
(i) Local Common Subexpression :-

⇒ Local common subexpressions are the instructions that compute a value that has already been computed.

$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = a - d$$

⇒ In the above given 3 address code we can see that the instruction $d = a - d$ is computing the same value that the instruction $b = a - d$ has already computed since value of 'a' and 'd' does not change.

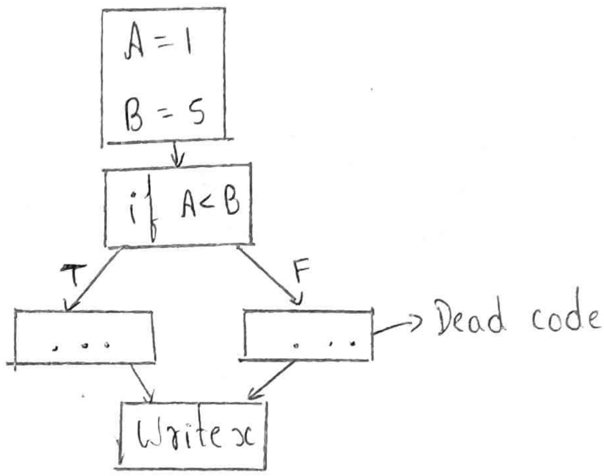
•) The common subexpression can be detected by noticing, as a new node M is about to be added, whether there is an existing node N with the same value of children in the same order with the same operator. If so, N computes the same value as M and may be used in its place.



•) In the above ~~sub~~-DAG, we can see that a single node is marked with two variables b, d depicting that they are the local common subexpression.

(ii) Dead code Elimination :-

•) Dead code refers to the sections of code within the program that is never executed during runtime and has no impact on program's output (or) behaviour.



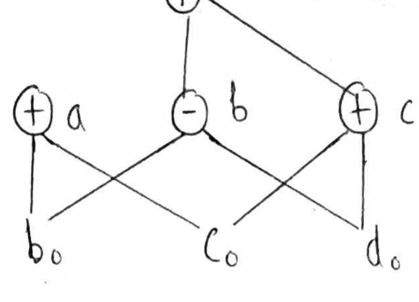
•) To remove dead code from the program we delete any root (node with no ancestors) from DAG that has no live variables attached i.e it is unused (or) unreachable

$a = b + c$

$b = b - d$

$c = c + d$

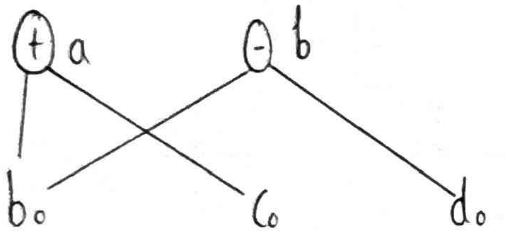
$e = b + c$



- In the given DAG, a & b are live but c and e are not, we can immediately remove the root labelled e.
- Then, node labelled c becomes a root and can be removed.

The roots labelled a & b remain, since they each have live variables attached.

After dead code removal:-



③ Explain code generation algorithm and generate code for the following expression $X = (a-b) + (a+c)$

- The essential part of the algorithm is a function getReg(I) which selects registers for each memory location associated with the three-address code instruction I.

→ **Machine Instructions for Operations:-**

For a three-address instruction such as $x = y + z$, do the following:

(i) Use getReg $(x = y + z)$ to select registers for x, y and z. Call these R_x, R_y and R_z

(ii) If y is not in R_y (according to register descriptor for R_y), then issue an instruction LD R_y, y', where y' is one of the memory locations of y (according to address descriptor of y).

(iii) Similarly, if z is not in R_z, issue an instruction LD R_z, z' where z' is location for z.

(iv) Issue the instruction ADD R_x, R_y, R_z.

→ **Ending of Basic Block:-**

For each variable x, we must generate the instruction ST x, R where R is the register in which x's value exists at the end of block.

Given expression :- $X = (a-b) + (a+c)$

The three address code is :-

$$t = a - b$$
$$u = a + c$$
$$v = t + u$$
$$X = v$$

where, t, u, v are temporaries.

$t = a - b$: LD R1, a
LD R2, b
SUB R2, R1, R2

$u = a + c$: LD R3, C
ADD R1, R1, R3

$v = t + u$: ADD R3, R1, R2

$X = v$: ST X, R3

(4) Write the three address code, basic block, flow graph for the following program segment:-

```
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        a[i, j] = 0.0;
    for (i=0; i<10; i++)
        a[i, i] = 1.0;
```

3 - Address code :-

```
i = 0
c1 : if i < 10 goto L1
     else goto L2
L1 : j = 0
     c2 : if i < 10 goto L3
          else goto L4

L4 : t5 = i+1
     i = t5
     goto c1
```

```
L3 : t1 = i * 40
     t2 = j * 4
     t3 = t1 + t2
     a[t3] = 0.0
     t4 = j+1
     j = t4
     goto c2
```

L_2 : $i = 0$

C_3 : if $i < 10$ goto L_5

else goto L_6

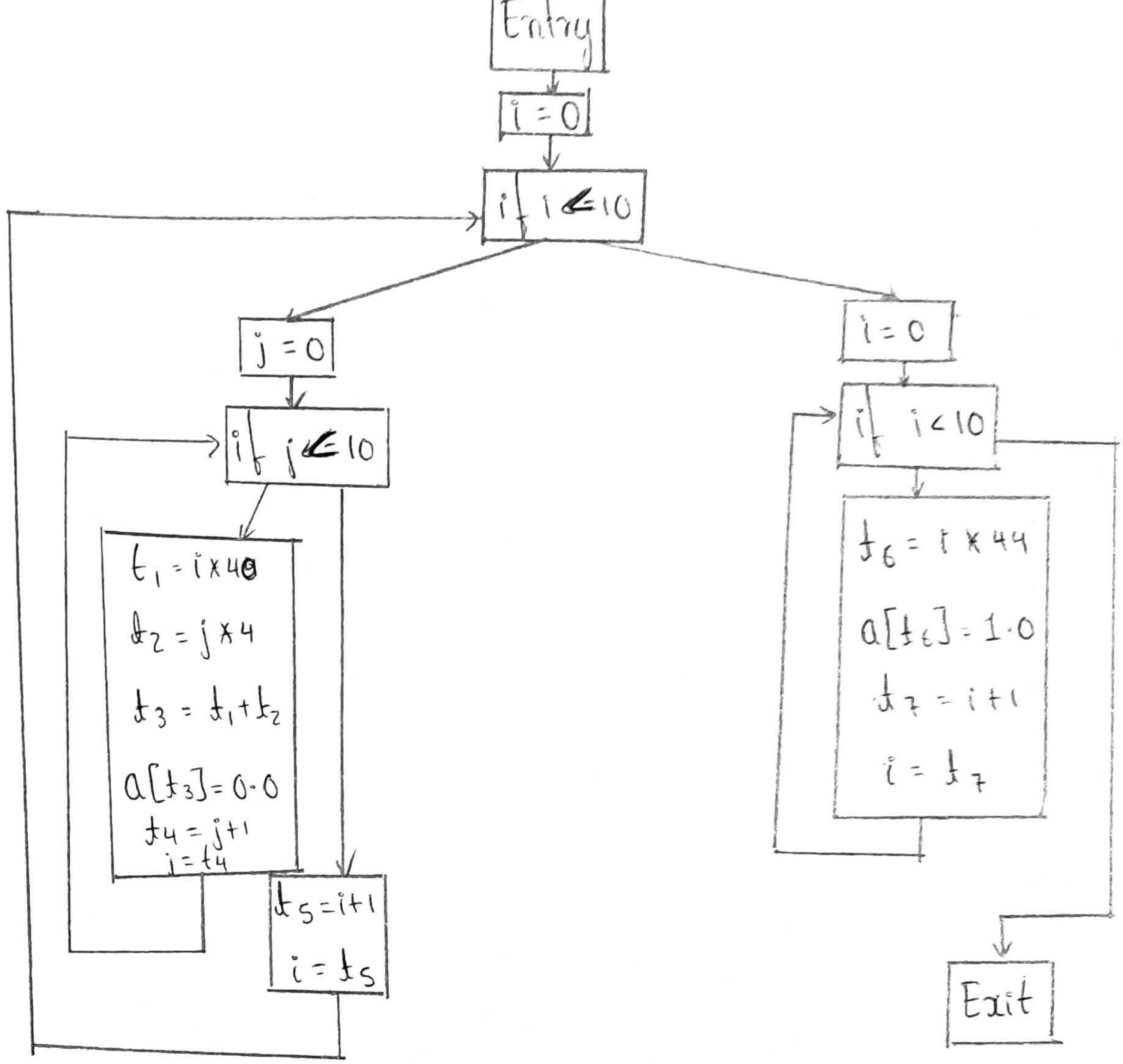
L_5 : $t_6 = i * 44$

$a[t_6] = 1.0$

$t_7 = i + 1$

$i = t_7$

goto C_3


Entry
i = 0
if i < 10
j = 0
if j < 10
t1 = i x 40
t2 = j x 4
t3 = t1 + t2
a[t3] = 0.0
t4 = j+1
j = t4
t5 = i+1
i = t5
i = 0
if i < 10
t6 = i x 44
a[t6] = 1.0
t7 = i+1
i = t7
Exit