Q. 4(b)(OR) Explain Peephole Optimization method. **(7 Marks)**

Ans. : Peephole Optimization method :

For each 3-address statement corresponding target code is generated. This statement-by-statement code generation strategy often produces target codes that have redundant instructions and sub optimal constructs. Quality of such target code can be improved by applying some optimizing transformations. Here optimizing transformation may or may not produce relative optimized code. Peephole optimization is the simple and effective technique used to improve target code locally. It is process of examining short sequence of target instruction known as peepholes by shorter and faster sequence whenever possible.

Local optimization that allows optimization of shorts sequence of target instructions of same basic block. It attempts to improve the performance of short sequence of target instructions by replacing this sequence by still shorter and faster sequence of target instructions. This technique can be directly applied to intermediate code also to improve intermediate representation of program. Peephole can be considered as short sequence of target instruction being examined for optimization that can be replaced by shorter and faster sequence. Peephole is assumed as small moving window on target program. Some peephole optimization techniques are :

(a) Elimination of Redundant Instructing
(b) Elimination of Unreachable Code
(c) Flow of Control Optimization
(d) Algebraic Simplification
(e) Reduction in Strength
(f) Use of Machine Idioms

(a) Elimination of redundant instructions :

Redundant instructions are instructions which do not achieve anything when executed sequentially.

Example

```
Load Ro, A      //load contents of register Ro into memory location A.
Store A, Ro     //Store content of memory location A into register Ro.
```

Such instructions are redundant. Therefore unnecessary store instruction in this case can be deleted.

(b) Elimination of unreachable code :

Unreachable code is code that cannot be reached by following any of execution paths of the program. That means unreachable code is kind of code that never executed.

Such code can be removed safely.

Example

An unlabeled instruction that is written immediately after an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions

Consider following example.

If DEBUG = 1 go to L_1

goto L_2

L_1: go to Print_Debug

L_2:

Here **Number of jumps = 2.** The first jump at go to label L1 and second jump at go to Print_Debug label.

This code can be replaced as

if DEBUG ≠ 1 go to Print_Debug.

L_2:

Here **Number of jumps = 1.** Only one jump at go to Print_Debug label.

Thus removing unnecessary jump optimises the code.

(c) Flow of control optimization :

Intermediate code often produces unnecessary jumps to jumps, or jumps to unconditional jump or conditional jumps to unconditional jumps. Unnecessary jumps can be eliminated in either intermediate code or target code by using peephole optimization.

Example

```
Go to L1
.
.
.
L1: go to L2
```

The above code can be replaced as :

```
Go to L2
Go to L2
```

Now if there is no other jump at L1 then other statement L1 can be safely eliminated as if it is preceded by unconditional jump.

```
If a < b go to L1
    :
L1 : go to L2
    ↓
jump to L1 is go to L1 can be eliminated as
    ↓
if a < b go to L2
    :
L1 : go to L2
    ↓
```

If there is only one jump to L_1 and L_1 is preceded by an unconditional jump then

go to L_1

:

L_1 : if a < b go to L_2

L_3 :

↓ can be replaced by if a < b go to L_2 go to L_3.

L_3 :

(d) Algebraic simplification :

In the program some algebraic identities occur frequently which can be eliminated.

Example

$$x := x + 0$$

or

$$x := x * 1$$

Such algebraic identities can be found and eliminated safely by peephole optimization.

(e) Reduction in strength :

This technique of peephole optimization replaces expensive operations by cheaper and equivalent operations.

Example

Implementation of x^2 can be replaced by cheaper but equivalent operation x * x. Implementation of x*2 can be replaced by cheaper and equivalent operation x + x. Fixed point multiplication division by power of 2 is cheaper to implement as shift operation.

(f) Use of machine idioms :

Target machine may have some hardware instructions to implement specific operation efficiently. Thus use of machines hardware instruction greatly reduces execution time.

Example

If machine supports auto increment or auto decrement of any variable value by 1 then it is better to replace instructions INC i as i = i + 1 and DNC i as i = i – 1.

Q. 5(a)

to repl...

Q. 5(a) Discuss generic issues in the design of code generator. (7 Marks)

Ans. : Generic Issues in the design of code generator :

Following generic issues are concerned while designing code generator :

(1) Input to code generator
(2) Target program
(3) Memory management
(4) Instruction selection
(5) Register allocation
(6) Choice of evaluation order
(7) Approaches to code generation

1. Input to Code Generation Phase (or Input to Code Generator) :

The input to code generator is the output of Intermediate Code Generator phase. The input to code generation phase is the complete error free intermediate code. So semantic errors should be detected before submitting input to the code generator and the output of intermediate code generator phase is :

(a) Intermediate representation (Intermediate code) :

This internedíate code is generated by front end of compiler.

Intermediate code produced by intermediate phase can be any of the following form :

(1) Quadruples (2) Triples

(b) Information in symbol table :

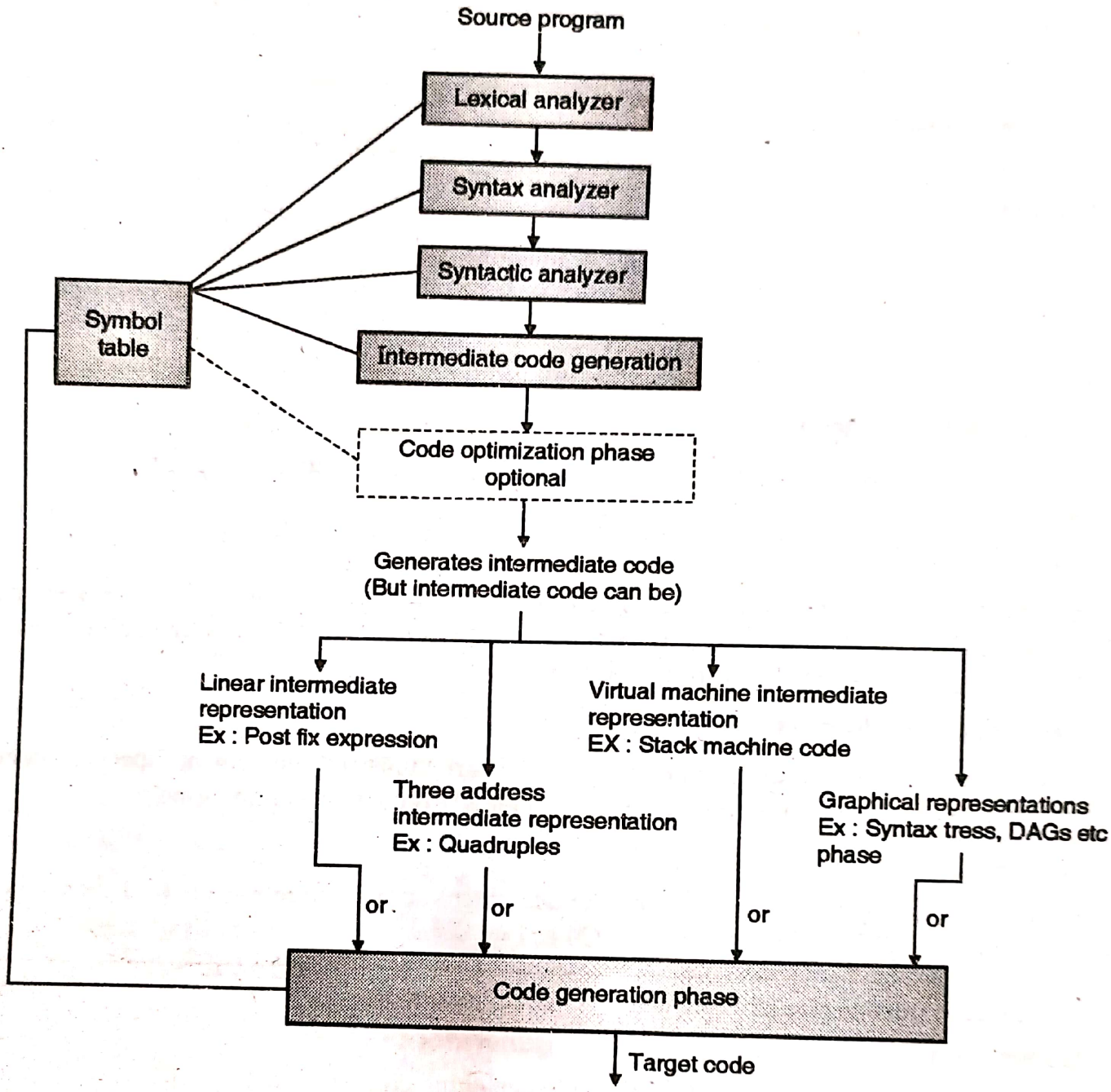

Fig. 1-Q. 5(a) : Input to code generator

This information is used to determine runtime address of all data objects which are specified by heir names in the input of code generation phase i.e. intermediate code as one of possible intermediate epresentations. Thus, before code generation phase, source program is processed by Lexical analyzer, Syntax analyzer, Semantic analyzers, Intermediate code generator and may or may not be Code ptimizer. Because code optimization can be implemented before and/or after and/or during code eneration phase of the compiler.

Thus, while designing code generator it is assumed that :

(i) Source program is already analysed, parsed and some equivalent intermediate code is generated.

(ii) Type checking is already performed.

(iii) Intermediate code has some additional type conversion functions.

(iv) All syntactic and semantic errors are detected.

(v) Intermediate code given as input to code generation phase is error free.

As shown in Fig. 1-Q. 5(a). intermediate code generated by intermediate code generation phase can be either of a possible types of intermediate representations as

(a) Linear representation as postfix expressions.

(b) Three address intermediate representation as quadruples.

(c) Virtual machine representation as stack machine code.

(d) Graphical intermediate representation as parse tree or DAGS.

Now code generator algorithms are different for different intermediate representation of source program i.e. code generator algorithms has to be designed according to intermediate representation generated by intermediate code generator. Algorithm discussed in this chapter can be used for three address code, parse or syntax trees. Other algorithms can be used for other intermediate representations.

2. Target Programs :

Code generation phase accepts intermediate code as input and produces target code as output. Target code produced can be of any one of the forms :

(a) Absolute machine language code :

Code generation phase can produce absolute machine language code as target code. This type of code can be placed to fixed location in main memory and can be immediately executed. This job is done by special system program known as Loader. Small programs can be compiled and executed quickly. WATFIV and PL/C and examples of compilers that produces absolute machine language code.

(b) Relocatable machine language code :

Code generation phase can produce relocatable machine language code (Object Modules) as target code. This target code has an advantage that they allow subprograms to be compiled separately. For execution steps to be followed are :

(i) Compilation of all subprograms separately.

(ii) Linking of all relocatable object modules.

(iii) Loading of all relocatable object modules.

(iv) Finally execute.

Advantages :

(i) All subprograms can be compiled separately.

(ii) Previously compiled program can be called by any other object program.

Disadvantages :

(i) Additional expenses of linking and loading.

(ii) Target machine code must be relocated and then loaded. Therefore compiler has to provide explicit relocation information if target machine do not support relocation.

(c) Assembly language code :

Code generation phase can produce assembly language code as target code.

Advantages :

1. Makes process of code generation easier by using symbolic names and macro facilities.
2. Reasonable alternative for machine with small memory where compiler uses many passes.
3. Good for explaining Code Generation Phase as it is more readable so throughout discussion we are using assembly language code as target code.

Disadvantages :

1. Bears price paid for additional step of assembly that is process of conversion of assembly language code to machine language code.
2. Thus, target code issue of code generation phase can be summarized as

Table 1-Q. 5(a) : Target code

Absolute machine code	Relocated machine code	Assembly language code
Can be directly placed into main memory and executed Smaller programs can be complied and executed directly	Allows separate compilation of subprograms. Any objet module can call previously compiled program. Object modules have to be linked, loaded and then can be executed. Bears price of linking and loading. If relocation is not supported by target m/c then compiler has to provide explicit relocation information.	Makes easier process of code generation using symbolic names and macro facility. Has better readability. Bears price for additional steps of assembly.

3. Memory Management :

Memory management is concerned with mapping of names in the source program to addresses of data objects in run time memory. This mapping is done collectively by Front end and code generation phase.

Responsibility of front end :

To create symbol table and store information in symbol table. Symbol table have information as name of symbol, amount of storage required storing that symbol and relative address of symbol. During processing of declarations, front end decides symbol name i.e. symbol being declared and amount of storage required for it by knowing type of symbol being declared and also relative address of that symbol in subprogram. Code generation phase uses information stored in symbol table and some mechanism to determine address of symbol.

If the three address code contains labels, then those labels can be converted in to equivalent memory addresses. For example, if reference to 'goto l' is encountered in three address code then appropriate jump instruction is generated by computing the memory address for label 'l'. Load and store does operations between main memory and external memory which result in slow execution of code. For local optimization it is necessary to improve code at level of basic block. Redundant load and store elimination result in local optimization at basic block level which results optimize target code.

4. Instruction Selection :

For target code set of instructions must be selected by code generation phase.

Important features of instruction set are :

(1) Uniformity (2) Completeness (3) Speed (4) Size

If target code i.e. set of target instruction do not support each data type informally then each exception for general rules needs special handling. If we are not concerned with efficiency of target code then instruction set can be selected straightforwardly but this often produces poor and redundant code. For each type of three address codes the code skeleton can be prepared which ultimately gives the target code for the corresponding construct.

5. Register Allocation and Assignments :

Instructions involving register operands are smaller and faster than those involving operand in memory. But for target machine registers are limited. Therefore efficient utilization of register is very important issue for good code generation.

Utilization of register has two subjects :

(i) Register allocation (ii) Register assignment

(I) Register allocation :

It is the process of just selecting set of variables that will reside in registers at any point in program.

(ii) Register assignment :

After register allocation explicitly specific registers are picked in which any variable can reside. Optimal register assignment to variables is difficult. Mathematically it is NP-Complete problem. Some machines require the registers which may be one or more than one depends upon the operand and results. For example in IBM system integer multiplication and integer division requires two registers.

Consider following three address code :

```
t = a / b
t = t + c
t = t * d
```

The efficient target machine code sequence will be

```
MOV a, R0
DIV b, R0
ADD c, R0
MUL d, R0
MOV R0, t
```

6. Choice of Evaluation Order :

The evaluation order factor is very important factor in generating an efficient target code as it affect efficiency of target code. Because different order of evaluation or computation needs different number of registers to hold intermediate results. Selecting the best order is one of the difficulties in code generation and we can avoid this problem by referring the order in which the three address code is generated by semantic actions. Again selecting optimal order of evaluation is NP-complete problem.

7. Approaches to Code Generation :

Irrespective of the approach used for code generation; the target code should be correct. Because the most important factor for code generation is that it should be correct. Using different approaches target code can be generated

(1) A simple code generation i.e. Straight forward code generation :

Uses subsequent use of operands. Considers each statement in turn. Keeps operands in registers as long as possible. Using peephole optimizations output of this algorithm can be optimized.

(2) Approach to code generation by considering flow of control in intermediate code :

Registers are allocated to heavily used operands in inner loops. As accessing and using contents of register is faster and easier.

(3) Tree-directed code generation techniques :

Allows construction retargetable code generators. Till this point we are now familiar with general issues or concepts of code generator. Before going into further detail to code generation phase let us take a view of target machine for which we have to generate target code.

❑❑❑

Q. 4(a) Explain Activation record and Activation tree in brief. **(7 Marks)**

Ans. :

Activation record

Activation record or frame is a contiguous block of storage to manage the information needed by a single execution of a procedure. Pascal and C uses stack area while FORTRAN uses static area to store the activation record. Activation record consist of collection of fields as shown in Fig. 1-Q.4(a).

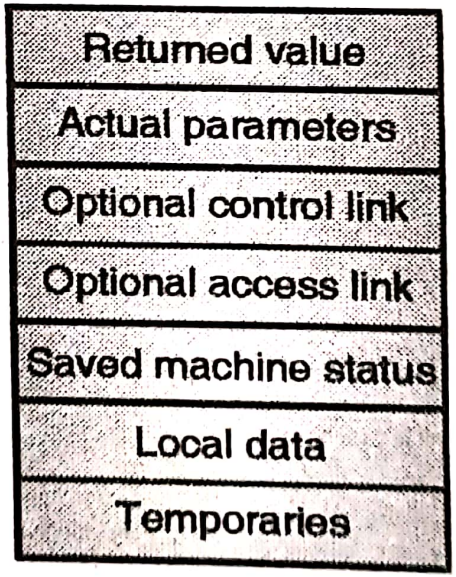

Fig. 1-Q.4(a) : A general activation record

The purpose of the various fields of an activation record is as follows :

(1) **Temporaries**

Temporary values such as those arising in the evaluation of expressions are stored in the field for temporaries.

(2) **Local data**

The field for local data holds data that is local to an execution of a procedure.

(3) **Saved machine status**

The field for saved machine status holds information about the state of the machine just before the procedure is called. This information includes the values of the program counter and machine registers that have to be restored when control returns from the procedure.

(4) **Optional access link**

The optional access link refers to nonlocal data held in other activation records.

For a language like Fortran access links are not needed because nonlocal data is kept in a fixed place. Access links or other related display mechanism, are needed for Pascal.

(5) **Optional control link**

The optional control link points to the activation record of the caller.

(6) **Actual parameters**

The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.

(7) **Returned value**

The field for the returned value is used by the called procedure to return a value to the calling procedure.

All the languages or all the compilers do not use all of these fields most of the time registers are used to take place of one or more of them. In languages like Pascal and C, the activation record of a procedure is pushed on the run-time stack when the procedure is called and it is popped from the stack when control returns to the caller or calling procedure. The size of each of these fields is determined at the time a procedure is called. In fact, the sizes of almost all fields can be determined at compiler time. An exception occurs if a procedure may have a local array whose size is determined by the value of an actual parameter, available only when the procedure is called at run time.

Activation Trees

Regarding the flow of control among procedures during the execution of a program the following thins are assumed.

(1) The execution of a program consists of a sequence of steps and the control flows sequentially i.e. one step to another.

(2) Each execution of a procedure starts at the beginning of the procedure body and eventually returns control to the point immediately following the place where the procedure was called. i.e. the flow of control between procedures can be depicted using trees.

Each execution of a procedure body is referred to as an activation of the procedure. The **lifetime** of an activation of a procedure P is the sequence of steps between the first and last steps in the execution of the procedure body. This lifetime includes the time spent executing procedures called by P, the procedures called by them and so on. So in general, the term **lifetime** refers to a consecutive sequence of

steps during the execution of a program. We can use a tree called an **activation tree** to depict the way how control enters and leaves activations.

In activation tree

(1) Each node represents an activation of a procedure.

(2) The root represents the activation of the main program.

(3) The node for P is the parent of the node for Q if and only if control flows from activation P to Q.

(4) The node for P is to the left of the node for Q if and only if the lifetime of P occurs before the lifetime of Q.

Q. 5(a)(OR) Explain Dynamic storage allocation technique.

Ans. : Dynamic storage allocation :

The internal data is de-allocated explicitly by the compiler and user allocated data is de-allocated either by user or the garbage collector. The techniques need to implement dynamic storage allocation depends on how storage is de-allocated. If de-allocation is implicit, then the run-time support package is responsible for determining when a storage block is no longer needed. If de-allocation is done explicitly by the programmer, then compiler has to perform very less task.

1. Explicit De-allocation :

(1) Explicit Allocation of Fixed Sized Blocks

(2) Explicit Allocation of Variable-Sized Blocks.

Explicit Allocation of Fixed Sized Blocks :

It is the simplest form of dynamic allocation. Allocation and de-allocation can be done by linking the blocks with some or no storage overhead as shown in following Fig. 1-Q. 5(a)(OR).

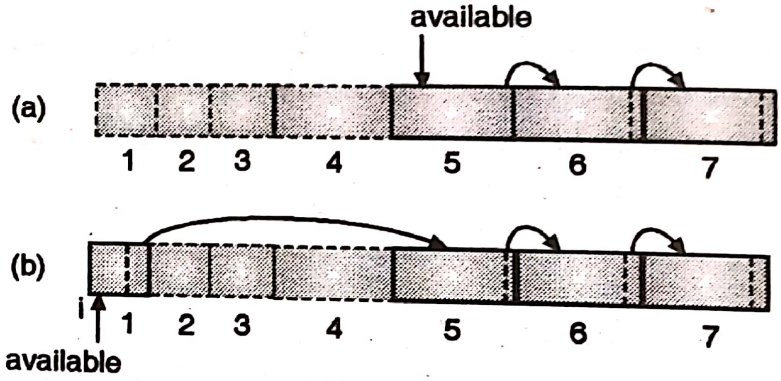

Fig. 1-Q. 5(a)(OR) : A de-allocation block is added to the list of available blocks

In Fig. 1-Q. 5(a)(OR) the pointer available points to the first block. Initialization is done by using a portion of each block for a link to the next block. Allocation means block is removed from the list and deallocation means block is added to the list of free block. And block is treated as a variant record. As the entire block is used there is no space overhead. When the block is de-allocated then the compiler routine uses some of the space from the block itself to link it into the list of available blocks as shown in above Fig. 1-Q. 5(a)(OR).

Explicit Allocation of Variable – Sized Blocks :

Storage is fragmented when blocks are allocated and de-allocated means heap may consists of alternate blocks that are free and in use as shown in Fig. 2-Q. 5(a)(OR).

Fig. 2-Q. 5(a)(OR) : Free and used blocks in a heap

Fragmentation occurs if block are variable size otherwise it will not occur. One method for allocating variable sized blocks is **first fit** method. In this method if a block of size S is to be allocated then we search for the first free block that is of size $f \geq S$ (where f – size of free block). Then this block is subdivided into a used block of size S and a free block of size (f – s). But in this method the time overhead will occur as time is required to search for a free block that is large enough. When a block is de-allocated, we check to see if it is next to a free block. If possible the de-allocated block is combined

with a free block next to it to create a larger free block. Combining a adjacent free blocks into a larger free block prevent further fragmentation from occurring.

2. Implicit De-allocation :

Cooperation between the user program and the run-time package is must in implicit de-allocation because run time package needs to know when a storage block is no longer in use. This cooperation is implemented by fixing the format of storage blocks. Consider the format of a storage block as shown in Fig. 3-Q. 5(a)(OR). The first problem is that of recognizing block boundaries. If the size of blocks 75 fixed, then position information can be used. For example if each block occupies 20 words then a new block begins every 20 words.

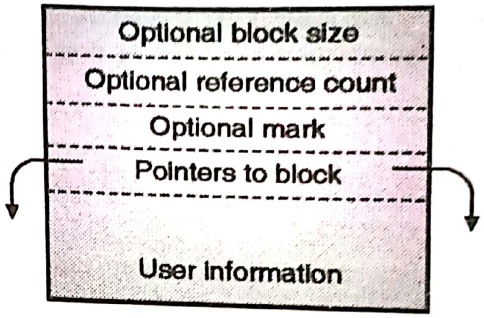

Fig. 3-Q. 5(a)(OR) : The format of a block

Otherwise in the inaccessible storage attached to a block we keep the size of a block. So we can determine where the next block begins. The second problem is that of recognizing if a block is in use we assume that a block is in use if it is possible for the user program to refer to the information in the block. The reference may occur through a pointer or after following a sequence of pointers, so the compiler needs to know the position in storage of all pointers.

Two approaches can be used for implicit de-allocation.

(1) Reference counts :

Reference count is used to keep the track of number of blocks pointing to the present block if this count drops to 0 then the block is de-allocated as it is not referred afterwards. But maintaining count is costly.

(2) Marking Techniques :

In this frozen pointers are used to keep the track of block in use. These pointers are placed in the heap itself.

These pointers are used to pour the paint into the heap if any block is reached by the paint then that block is in use and if not reached these are unused hence de-allocated.

Because of variable sized blocks, the used blocks are moved at one end and unused at another end; the process is called as compaction. Compaction also requires information about the pointers in blocks because when a used block is moved all pointer to it have to be adjusted to reflect the move. Its advantage is that afterward fragmentation of available storage is eliminated. Comparison in between static storage management and dynamic storage management.

Q. 5(b)(OR) Discuss any three methods for code optimization. **(7 Marks)**

Ans. : Methods for code optimization :

(1) Common sub expression elimination.
(2) Loop invariant code motion.
(3) Removal of induction variable.
(4) Reduction in strength.

1. Elimination of Common Sub expression :

The common sub expression is the expression that is used repeatedly in the program and computed only once. So any expression 'E' is called common sub expression if value of expression 'E' is computed previously and is not changed since previous computation. Ex. Consider following simple code.

```
p = a + b – c;        q = p / p
r = q + p             s = a + b – c;
```

In these sequence of instructions value of a + b – c is computed at first instruction. This first instruction is followed by two instructions which calculates value of q and r without changing the values of variables a, b and c. It means that in these sequence of instructions value of a + b – c is computed at first instruction. Last instruction is again calculating the value of a + b – c. So the expression a + b – c is considered as common sub expression. And such common sub expressions can be eliminated by using optimization techniques. Common sub expression may be local or global. The common sub expression appearing in only one basic block is called as local common sub expression.

So in above sequence of instructions common sub expression a + b – c can be eliminated as

```
temp = a + b – c;     p = temp;
q = p / p             r = q + p
s = temp;
```

A value of common sub expression a + b – c is calculated in the temporary variable 'temp' at the first instruction and same is used at the last instruction instead of recalculating. Consider following basic block consist of sequence of instructions

Table 1-Q. 5(b)(OR) : Basic block B_1

```
t1 := 8 + i
t2 := a [8+i]
t3 := 7 * j
t4 := 7 * j
t5 := t3 + t2
t6 := a [t4] + t5
goto B2
```

In the above basic block sequence of 3-address instructions are executed sequentially. Sequence is ended with unconditional jump to basic block B_2. In the basic block B_1 same expressions i.e. 8 + i and 7 * j are used to compute value of temporary variables t_1, t_2 and t_3, t_4 respectively. In the sequence

values of variables i and j are not changed in between the calculations. Therefore, these expressions are called as common sub expressions. And as these common sub expressions are appearing in the same basic block these expressions are called as local common sub expressions. So to improve the performance of the execution we need to perform code optimization by avoiding the recalculation of such local common sub expressions. Therefore common sub expressions can be removed to form another equivalent basic block B_1 as shown.

Table 2-Q. 5(b)(OR)

$t_1 := 8 + i$
$t_2 := a\ [t_1]$
$t_3 := 7 * j$
$t_5 := t_3 + t_2$
$t_6 := a\ [t_3] + t_5$
goto B_2

So common sub expressions are eliminated by avoiding recalculations, for that the instruction $t_4 = 7*j$ is removed and t_4 is replaced by t_3 and in the second instruction $8 + i$ is replaced by t_1. After elimination of common sub expression the basic block should be equivalent means the value calculated by them should be same.

2. Loop Invariant Code Motion :

This is also known as code movement and also known as removal of loop invariant code. Loop invariant code is the code which computes same value in every iteration of loop. Code motion decreases amount of code in loop by moving invariant code outside of the loop. Such loop invariant code is detected and placed before loop by the process of code motion. In this technique code is moved from inside the loop to outside the loop i.e. before the entry point of loop.

```
while (i < Min-2)
{
    // some code that does not change the value of Min
}
```

Now in the above code value of Min is never changed so it is called as code invariant and Min-2 can be shifted outside the loop as

```
A = Min-2
while (i < A)
{
    // some code that does not change the value of Min
}
```

Example

```
for (i=0; i<n; i++)
{
    x = y + 4;
    // some code
```

```
        z = y + 4 * 60;
}
```

In the above code the statement y + 4 is loop invariant as it does not change For every iteration so that can be moved outside the loop as :

```
p= y + 4
for( i=0; i<n; i++)
{
        x = p;
        // some code
        z = p * 60;
}
```

3. Removal of Induction Variable :

Induction variables are variables which either decreases or increases by same amount for each iteration of loop. If there are more than one loop variable then we can get rid of all except one induction variable i.e. for loop there must be at least one induction variable to reach loop terminating conditions. In inner loop of block B_2 there are two induction variables n and t_3 whose values are decreased by same amount for every iteration of loop.

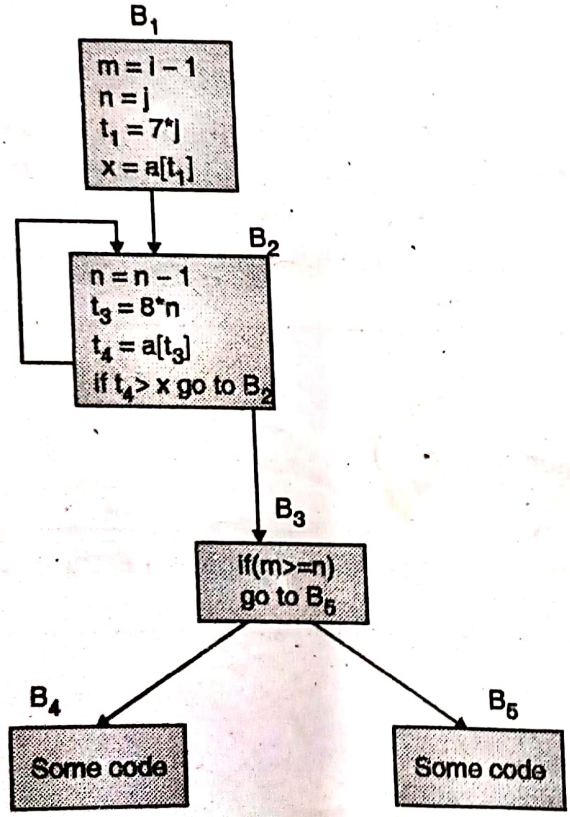

Fig. 1-Q. 5(b)(OR) : Before strength reduction

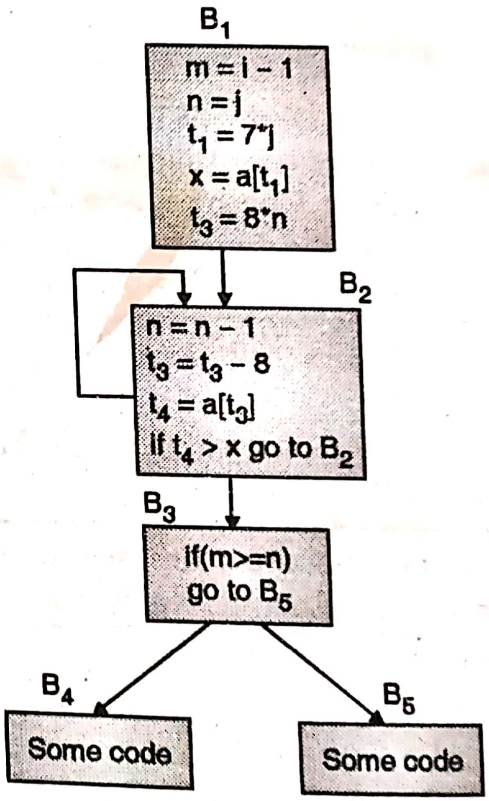

Fig. 2-Q. 5(b)(OR) : After strength reduction

But we cannot get rid of either n or t_3 because n is used in B_3 and t_3 is used in block B_2 itself. Now value of n is decreased by 1 for every iteration & hence value of t_3 is decreased by 8 because $t_3 = 8 * n$. Now, expensive multiplication operation in the loop can be replaced by cheaper subtraction operation if in inner loop we have statement as $t_3 = t_3 - 8$. But to have this statement we have to initialize value of t_3 also. So t_3 can be initialized with n as $t_3 = 8 * n$. This statement can be inserted in the block in which value of 'n' is initialized i.e. block B_1 as shown.