AKTU

B.Tech 6th Sem

GATEWAY CLASSES

CS IT & Allied

Compiler Design

Unit-5 in ONE SHOT

- AKTU PYQs
- Imp.Questions

Helpline No- 7819 0058 53

AKTU QUESTIONS

Q.1.	Define loop jamming and induction variable	AKTU 2022-23
Q.2.	Explain the following – copy propagation, dead code elimination, code motion, Reduce in strength	AKTU 2022-23
Q.3.	Explain in DAG representation of the basic block with example	AKTU 2022-23
Q.4.	Define the following terms- Basic block ,Next use information, Flow graph	AKTU 2021-22
Q.5.	Discuss the various issues in design of code generator and code loop optimization	AKTU 2021-22
Q.6.	Discuss about non linear type intermediate code and constant folding	AKTU 2020-21
Q.7.	Explain in detail about loop optimization	AKTU 2020-21
Q.10.	Define DAG construct DAG for the expression p+p*(q-r)+(q-r)*s	AKTU 2020-21
Q.11.	EXPLAIN the following in the organization of the code optimizer- code flow analysis, data flow analysis transformation	AKTU2009-10
Q.12.	Explain the optimization of basic block and explain the DAG representation of basic block	AKTU 2009-10
Q.13.	Define between LR and LL parse	AKTU 2022-23

AKTU QUESTIONS

Q.14	What is meant by viable prefix and peephole optimization	AKTU 2018-19
Q.15.	What is global flow analysis? How it is used in code optimization	AKTU 2018-19
Q.16.	Write a short note with example loo unrolling, loop jamming, dominators ,vaible prefix	AKTU 2018-19
Q.17.	Explain what constitute a loop in flow graph and how will you do loop optimization in code optimization of a compiler	AKTU 2018-19/2015-16
Q.18.	Write a short note on loop optimizations	AKTU 2018-19
Q.19.	Construct the flow graph for the following prod = 0 ; i = 1 ; do { prod = prod + a[i] x b[i] ; i = i + 1 ; } while (i <= 10) ;	AKTU 2018-19
Q.20.	Explain the following :- loop unrolling, loop jamming , Global dataflow analysis ,induction variable elimination, local and loop optimization	AKTU 2015-16
Q.21.	Explain local and global elimination of common sub-expression	AKTU 2015-16
Q.22.	Explain copy propagation and dead code elimination ,optimization of basic block	AKTU 2015-16
Q.23.	How to perform register assignment for outerloop	AKTU 2016-17

AKTU QUESTIONS

Q.24.	Why are quadruples preferred over triples ina optimizing compiler?	AKTU 2016-17
Q.25.	Discuss the detail of process of optimization of basic block with an example	AKTU 2017-18
Q.26.	Write an algorithm to partition of sequence of 3 address statement into basic block	AKTU 2016-17
Q.27.	What is DAG? How DAG Is different from syntax tree	AKTU 2018-19

Code generation in compiler design

- Code Generation is the final phase of a compiler. It takes the intermediate representation (IR) of a program (produced after parsing, semantic analysis, and optimization) and converts it into target code, usually assembly or machine code.

Purpose:

- To generate low-level code that can be executed by a real or virtual machine (e.g., CPU)

- **Input:**

Intermediate Code (e.g., Three-address code, syntax tree)

- **Output:**

Target Code (e.g., Assembly, Machine code, Bytecode)

Source code a = b + c * d;

Intermediate Code (3-address):

t1 = c * d

t2 = b + t1 , a = t2

target code:

MOV R1, c

MUL R1, d

MOV R2, b

ADD R2, R1

MOV a, R2

Tasks in Code Generation:

- **Instruction selection** – Choose proper machine instructions.

- **Register allocation** – Decide which variables go into which registers.
- **Address calculation** – Handle memory locations and variable addresses.
- **Instruction ordering** – Arrange instructions for efficiency.

LL Parser	LR Parser
First L of LL is for left to right and second L is for leftmost derivation.	L of LR is for left to right and R is for rightmost derivation.
It follows the left most derivation.	It follows reverse of right most derivation.
Using LL parser parser tree is constructed in top down manner.	Parser tree is constructed in bottom up manner.
non-terminals are expanded.	terminals are compressed.
Starts with the start symbol(S).	Ends with start symbol(S).
Ends when stack used becomes empty.	Starts with an empty stack.
Terminal is read after popping out of stack.	Terminal is read before pushing into the stack.

Issues in the design of a code generator

Input to Code Generator

- The input to the code generator comes from the intermediate code generated by the compiler's front-end.
- This intermediate code is usually a higher-level representation of the program, like triples, quadruples, or abstract syntax trees.
- Along with this intermediate code, the code generator also uses information from the symbol table, which holds the addresses of variables and other data objects.
- One key challenge here is that the input must be free from syntactic and semantic errors, as the code generator assumes that proper type-checking and other error checks have already been handled by the front-end.
- Handling the input correctly is crucial for generating the correct target code.

2. Target Program

The target program is the final output of the code generator, which can be in the form of absolute machine language, relocatable machine language, or assembly language. Each type of output has its own set of challenges:

- **Absolute Machine Language** is easy to execute but lacks flexibility because it is bound to specific memory locations.
- **Relocatable Machine Language** allows parts of the program to be moved around in memory, making it suitable for linking multiple modules, but it requires a linking loader and has some overhead.
- **Assembly Language** is symbolic and needs an additional step (an assembler) to convert it into machine code, but it makes the code generation process easier.

3.Memory Management

- Memory management in the code generation phase involves mapping variable names to their corresponding memory locations.
- The code generator works closely with the front-end to access the symbol table, where memory addresses for variables are stored.
- A major challenge is ensuring that the code generator uses memory efficiently, avoids memory conflicts, and correctly handles dynamic memory allocation.

- This requires careful handling of variable storage, particularly for dynamically allocated objects or large data structures, such as arrays or objects in object-oriented languages.

Instruction Selection

- Instruction selection is the process of choosing the most suitable machine instructions to translate intermediate code into executable code. The goal is to optimize the generated code by selecting instructions that are efficient and appropriate for the target machine.
- If the right instructions are not selected, the resulting code can be inefficient and slow.
- A code generator might need to decide between different ways of implementing the same operation, such as using different addressing modes or optimizing for processor-specific features

Three Address Code:

P := Q + R

S := P + T

Assembly Code (Inefficient):

MOV R0, Q (Load the value of Q into register R0)

ADD R0, R (Add the value of R to the value in R0)

MOV P, R0 (Store the value of R0 into the variable P)

MOV R0, P (Load the value of P back into R0)

ADD R0, T (Add the value of T to R0)

MOV S, R0 (Store the value of R0 into the variable S)

Assembly Code (Efficient):

MOV R0, Q (Load Q into R0)

ADD R0, R (Add R to R0)

ADD R0, T (Add T to R0)

MOV S, R0 (Store the final result in S)

Register Allocation Issues

Efficient use of registers is important because registers are faster than memory, and utilizing them effectively can significantly improve program performance. The challenge lies in selecting the right variables to store in registers at different points in the program.

Register allocation involves two stages:

Register Allocation: It is selecting which variables will reside in the registers at each point in the program

Register Assignment: Assigning specific registers to those variables selected in Register Allocation. The difficulty arises in managing which variables are allocated to registers, especially when the number of available registers is limited. Poor register allocation can lead to spills, where data is temporarily stored in memory, causing slower performance.

Three address code:

t:= a + b

t:= t*c

t:= t/d

Their efficient machine code sequence is as follows:

(inefficient way)

```
Mov R0, a
ADD R0, b
Mov t, R0
Mov R1, t
Mul R1, C
Mov t, R1
```

```
Mov, R0, a
Mov R1, b
ADD R0, R1
.
.
```

(efficient)

```
Mov R0, a      R0 ← M[a]
Add R0, b      R0 ← R0 + M[b]
Mul R0, C      R0 ← R0 * M[C]
Div R0, d      R0 ← R0 * M[d]
Mov t, R0      M[t] ← R0
```

Evaluation Order –

- Evaluation order is the sequence in which parts of an expression are calculated during code generation.
- A good order uses fewer registers and less memory
- A bad order can make the program slower and less efficient

Disadvantages of code generator

- **Limited Flexibility**

  Code generators often create code for one platform or purpose, so they can't handle many types of inputs or multiple targets easily.

- **Maintenance Overhead**

  You must update the generator whenever the generated code changes — this adds extra work and risk of errors.

- **Hard to Debug**

  Generated code can be complex or unreadable, making it harder to find and fix bugs compared to hand-written code.

Performance Issues

Generated code is not always optimized, so it might run slower than well-written manual code a problem in performance-critical apps.

- **Steep Learning Curve**

  Developers need to learn the generator system, which can take time and make it harder for new team members to start.

- **Over-Reliance**

  Relying too much on code generators can make developers forget how to write manual code, reducing flexibility and creativity.

Role of Addresses in Target Code

- Addresses in target code help convert intermediate representation (IR) names (like variables, labels, temporaries) into actual memory locations or offsets.
- They are essential for generating correct machine-level instructions during procedure calls, variable access, and returns.

When a compiled program runs, its logical address space is typically divided into four main areas:

a. Code Area

- Holds the executable target code.
- Size is known at compile time.
- Contents include machine instructions generated from source code.

b. Static Data Area

- Stores global constants, global/static variables, and compiler-generated data.
- Size is also known at compile time.
- Values here persist for the program's entire execution.

Heap Area

- Used for dynamic memory allocation (e.g., malloc() in C, new in C++).
- Managed at runtime, grows upward in memory.
- Size is not known at compile time.

Stack Area

- Stores activation records (stack frames) for procedure/function calls.
- Includes local variables, return addresses, and parameters.

downward in memory.

- Size is not known at compile time.

Code Generation for Procedure Calls and Returns

Procedure call handling uses stack allocation:

- Parameters and return addresses are pushed onto the stack.
- Local variables use relative addresses (e.g., offset from the stack pointer).
- When a procedure returns, its activation record is popped from the stack.

- Addresses in the target code serve to:
- Map IR names to physical/logical locations.
- Support memory management across statically and dynamically allocated regions.
- Enable structured handling of procedure calls using the stack.

Basic Blocks

- Basic block is a set of statements that always executes in a sequence one after the other.

The characteristics of basic blocks are-

- They do not contain any kind of jump statements in them.
- There is no possibility of branching or getting halt in the middle.
- All the statements execute in the same order they appear.
- They do not lose the flow control of the program.

Example Of Basic Block

Three Address Code for the expression a = b + c + d is-

(1) T1 = b + c

(2) T2 = T1 + d

(3) a = T2

All the statements execute in a sequence one after the other. Thus, they form a basic block.

Example Of Not A Basic Block-

(1) If A<B goto (4)

(2) T1 = 0

(3) goto (5)

(4) T1 = 1

(5)

Partitioning Intermediate Code Into Basic Blocks-

- Any given code can be partitioned into basic blocks using the following rules-

Rule-01: Determining Leaders-

Following statements of the code are called as **Leaders**–

- First statement of the code.
- Statement that is a target of the conditional or unconditional goto statement.
- Statement that appears immediately after a goto statement.

```
1. a = 5
2. if a < 10 goto 4
3. b = 20
4. c = a + b
5. goto 2
6. d = c * 2
```

First statement is a leader.

Line 1 is a leader.

Any statement that is the target of a conditional/unconditional goto is a leader.

goto 4 → Line 4 is a leader.

goto 2 → Line 2 is a leader.

Any statement that immediately follows a conditional goto is a leader.

Line 2 has if a < 10 goto 4, so Line 3,6 is a leader.

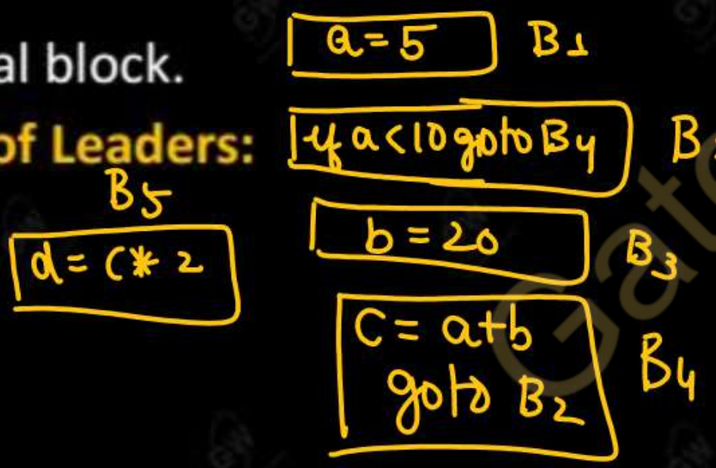
Rule-02: Determining Basic Blocks-

- All the statements that follow the leader (including the leader) till the next leader appears form one basic block.
- The first statement of the code is called as the first leader.
- The block containing the first leader is called as Initial block.

Final List of Leaders:

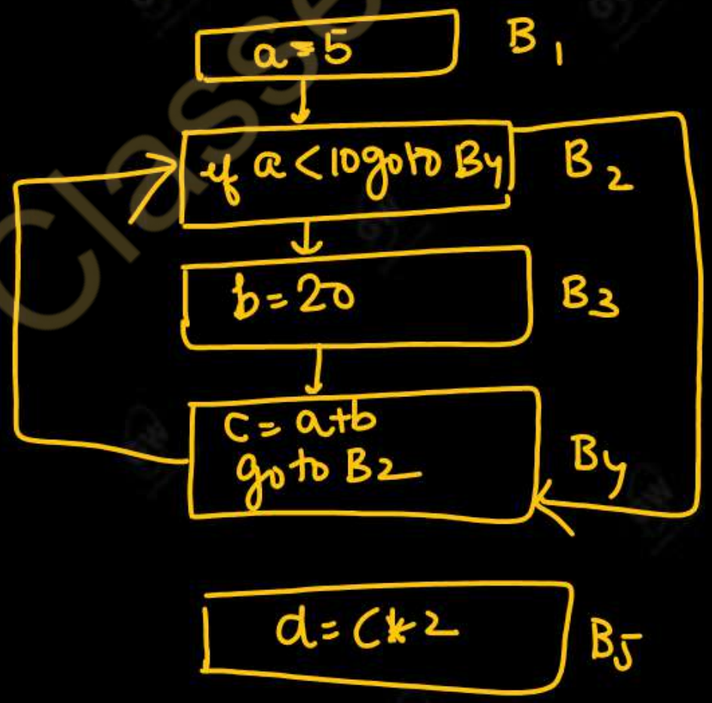
Line 1
Line 2
Line 3
Line 4

Lihe 6



Flow Graphs-

A flow graph is a directed graph with flow control information added to the basic blocks.



Problem-01:

Compute the basic blocks for the given three address statements

(1) PROD = 0

(2) I = 1

(3) T2 = addr(A) – 4

(4) T4 = addr(B) – 4

(5) T1 = 4 x I

(6) T3 = T2[T1]

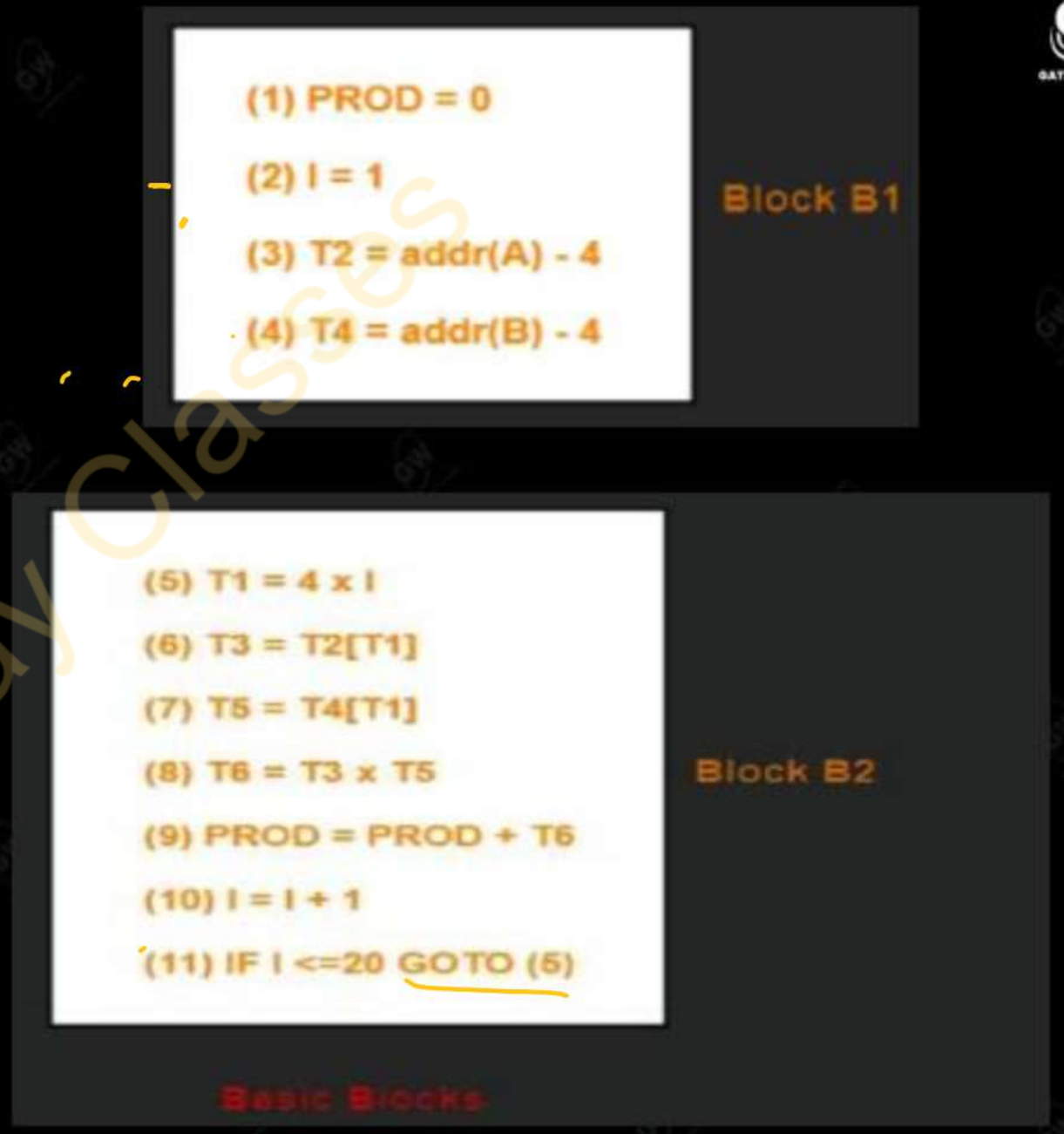
(7) T5 = T4[T1]

(8) T6 = T3 x T5

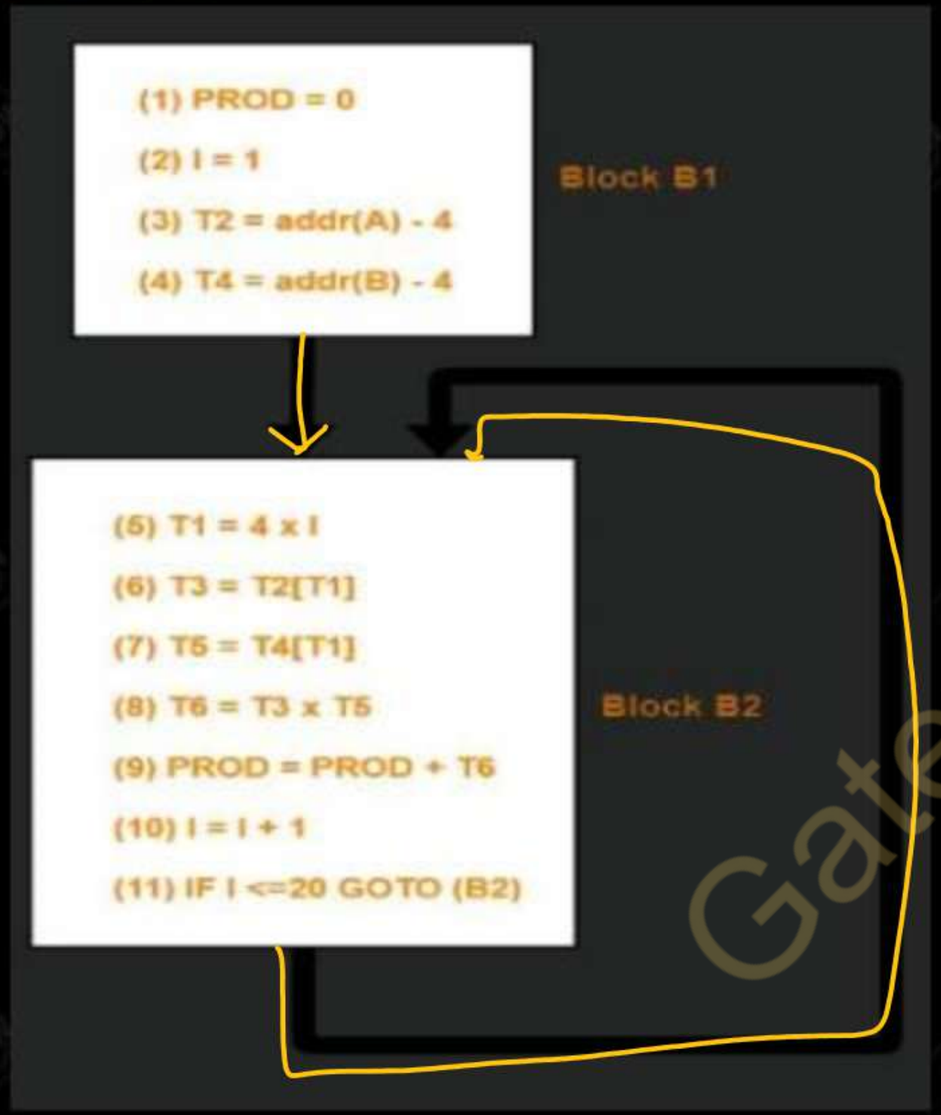
(9) PROD = PROD + T6

(10) I = I + 1

(11) IF I <=20 GOTO (5)



flow graph for the three address statements



three address code

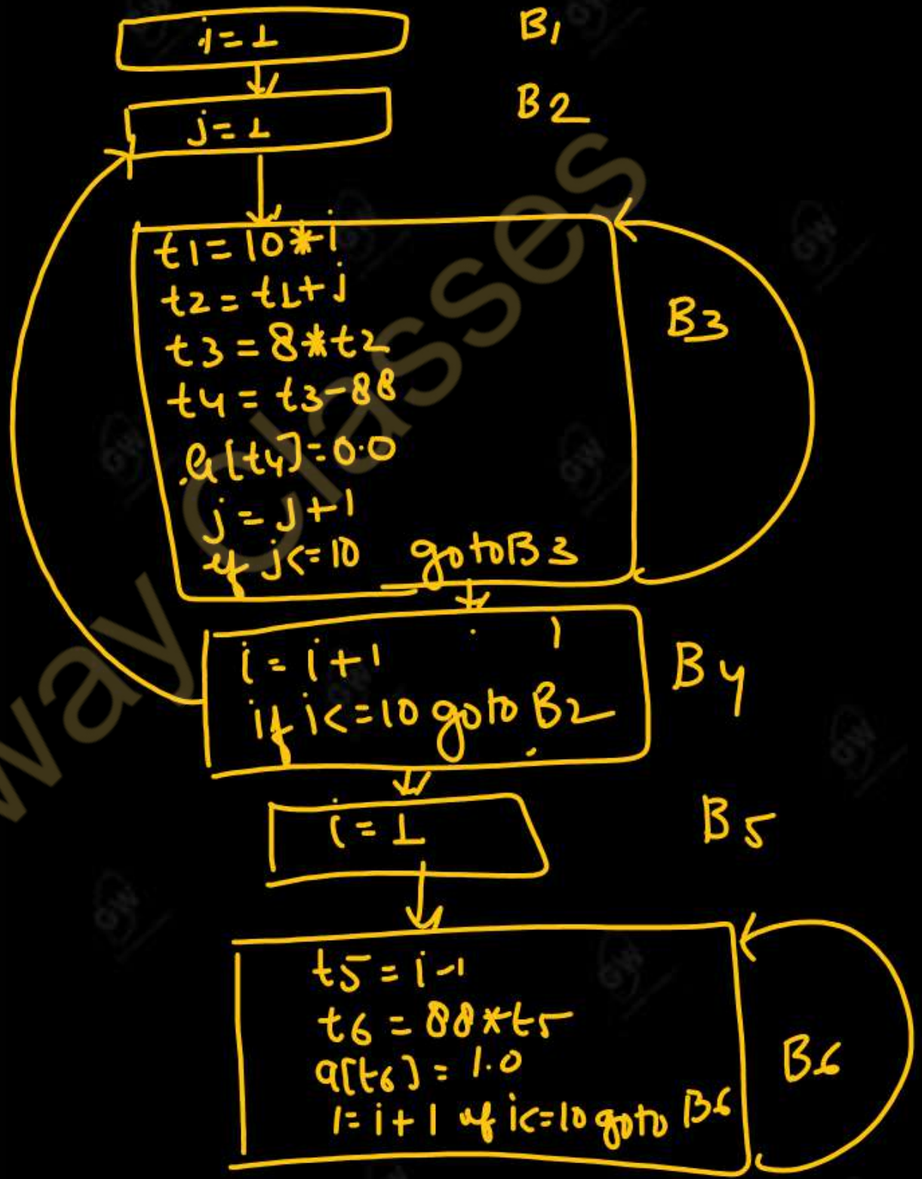
The following quads do the same

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if J <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

The leaders are then 1, 2, 3, 10, 12, and 13.

The basic blocks are {1}, {2}, {3,4,5,6,7,8,9}, {10,11}, {12}, and {13,14,15,16,17}.

Flow Graph



Write an algorithm to partition a sequence of three address statements into basic blocks.

The algorithm for construction of basic block is as follows :

Input : A sequence of three address statements.

Output : A list of basic blocks with each three address statements in exactly one block.

Method :

1. We first determine the set of leaders, the first statement of basic block.

The rules we use are given as :

a. The first statement is a leader.

b. Any statement which is the target of a conditional or unconditional goto is a leader.

c. Any statement which immediately follows a conditional goto is a leader.

For each leader construct its basic block, which consist of leader and all statements up to the end of program but not including the next leader. Any statement not placed in the block can never be executed and may now be removed, if desired.

Consider the following code-

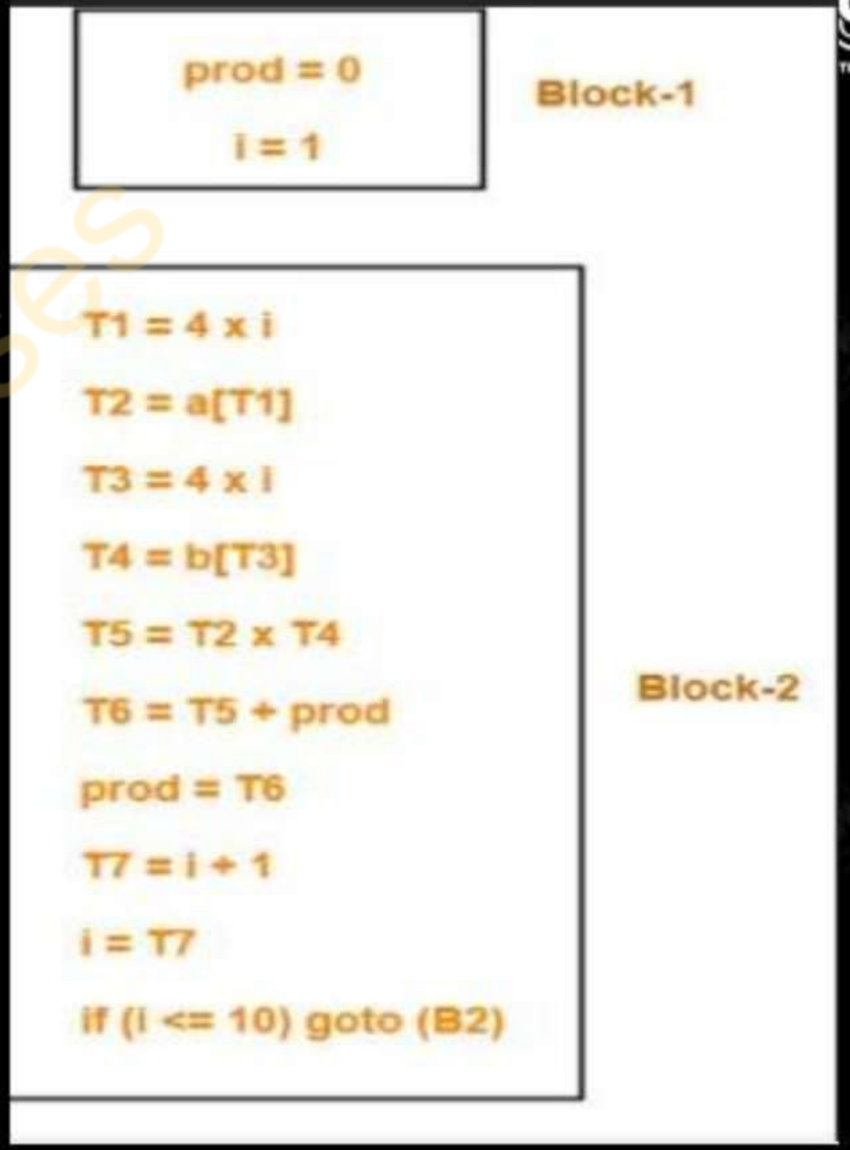
```
prod = 0 ;
i = 1 ;
do
{
prod = prod + a[ i ] x b[ i ] ;
i = i + 1 ;
} while (i <= 10) ;
```

Compute the three address code.

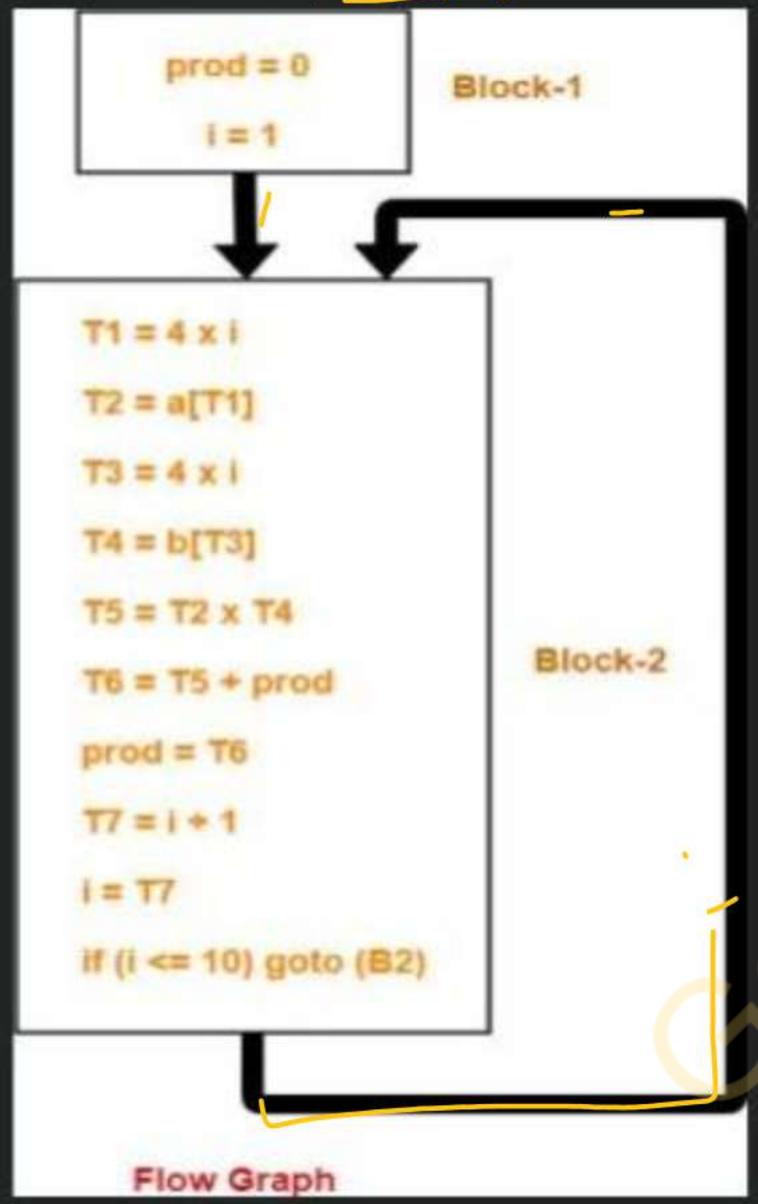
Compute the basic blocks and draw the flow graph.

Three address code for the given code is-

```
1  prod = 0
2  i = 1
3  T1 = 4 x i
4  T2 = a[T1]
5  T3 = 4 x i
6  T4 = b[T3]
7  T5 = T2 x T4
8  T6 = T5 + prod
9  prod = T6
10 T7 = i + 1
11 i = T7
12 if (i <= 10) goto (3)
```

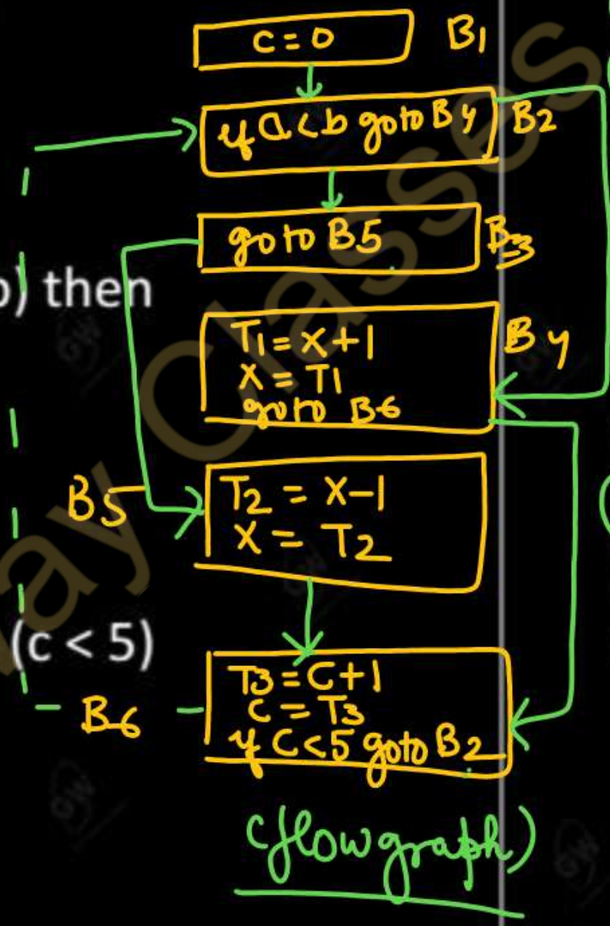


Flow graph



Write the three address code and find the basic block and and draw flow graph

```
c = 0
do
{
if (a < b) then
x++;
else
x–;
c++;
} while (c < 5)
```



(flow graph)

Three address code for the given code is-

```
1.c = 0                          — B1
2.if (a < b) goto (4)            — B2
3.goto (7)                       — B3
4.T1 = x + 1
5.x = T1                         — B4
6.goto (9)
7.T2 = x – 1
8.x = T2                         — B5
9.T3 = c + 1
10.c = T3                        — B6
11.if (c < 5) goto (2)
```

What is loop ?

Loop is a collection of nodes in the flow graph such that :

1. All such nodes are strongly connected i.e., there is always a path from any node to any other node within that loop.

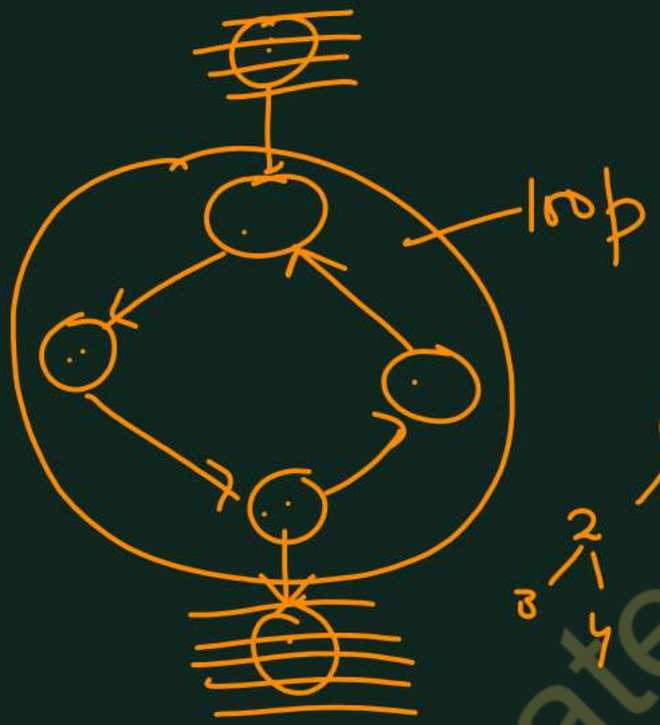
2. The collection of nodes has unique entry. That means there is only one path from a node outside the loop to the node inside the loop.

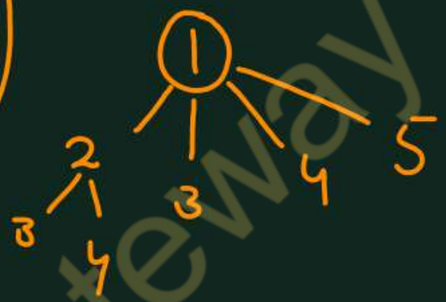
Following term constitute a loop in flow graph:

1. Dominators :

- A node is said to dominate node n in a flow graph if every path to node n from initial node goes through d only.
- Every initial node dominate all the remaining node in a flow graph.
- Every node also dominate itself.
- The immediate dominator (or idom) of a node n is the unique node that strictly dominates n but does not strictly dominate any other node that strictly dominates n. Every node, except the entry node, has an immediate dominator.

$A \rightarrow B \rightarrow C$ Example





Start node

	Dominating
1	1, 2, 3, 4, 5
2	2, 3, 4, 5
3	3,
4	4
5	5

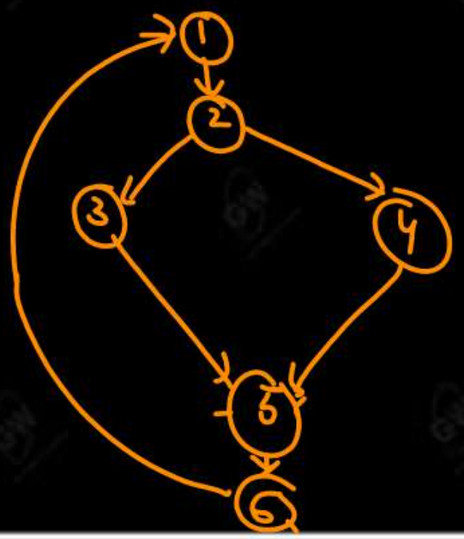
∴

- A dominator tree is a tree where each node's children are those
- nodes it immediately dominates. Because the immediate dominator is unique, it is a tree. The start node is the root of the tree.

Natural loop

6 → 1
n → d

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



2.Natural loops:

- The natural loop can be defined by a back edge n -> d such that there exists a collection of all the nodes that can reach to n without going through d and at the same time d can also be added to this collection
- Loop in a flow graph can be denoted by n –> d such that d dom n.
- These edges are called back edges and for a loop there can be more than one back edge.
- If there is p-> q then q is a head and p is a tail and head dominates tail.

3. Inner loop:

- A loop that contain no other loop is called inner loop.

4. Pre-header :

- The pre-header is a new block created such that successor of this block is the header block.
- All the computations that can be made before the header block can be made before the pre-header block.

5. Reducible flow graph

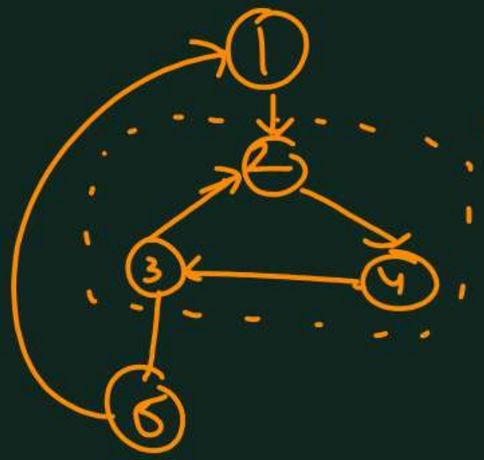
- A flow graph G is reducible graph if an only if we can partition
- the edges into two disjointed groups i.e., forward edges and backward edges.
- These edges have following properties :

i. The forward edge forms an acyclic graph.

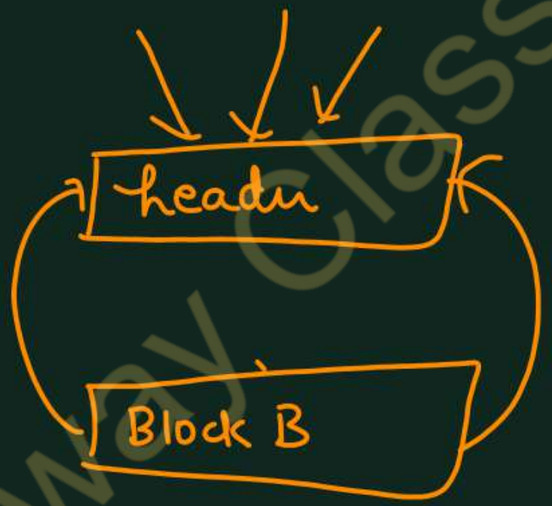
ii. The backward edges are such edges whose heads dominate their tails.

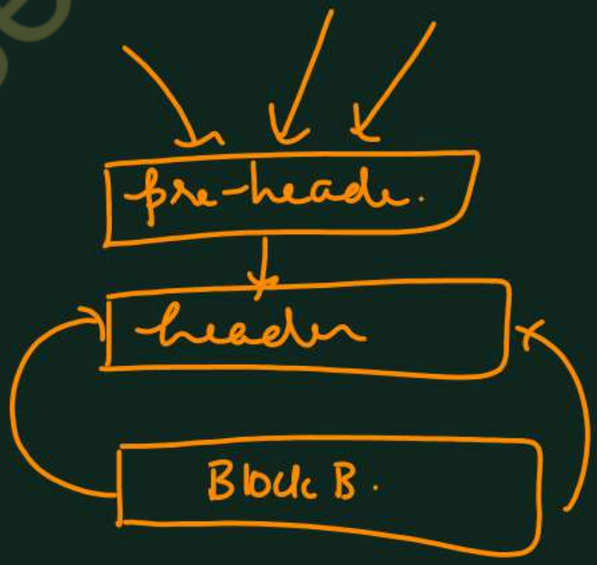
- The program structure in which there is exclusive use of if-then, while-do or goto statements generates a flow graph which is always reducible.

Innerloop



2,4,3

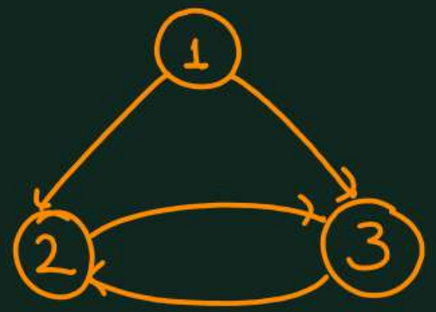





```
for(i=0; i<10, i++)
  {  y= a+z
     a[i] = a[i] +5
  }
```

```
y= a+z
for(i=0, i<10, i++)
  {
     a[i] = a[i] +5
  }
```

Non-Reducible flow graph

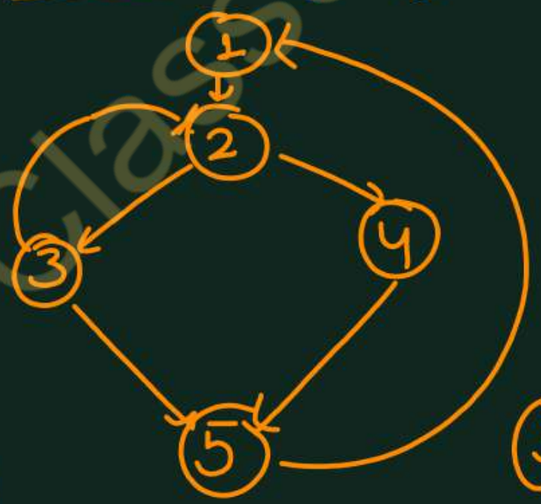


$n \to d$

$3 \to 2$ — a not dominating this is not backedge

2 ← 3 Not Backedge

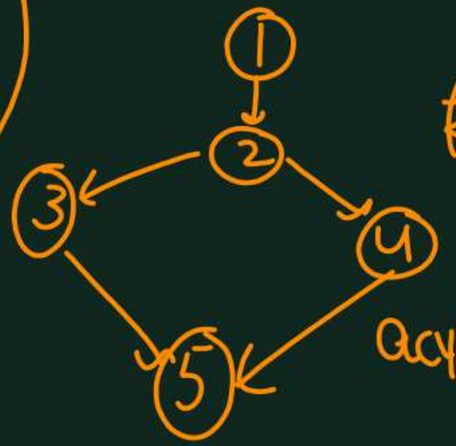
Reducible flow graph



$5 \to 1$ — dominating edge

$n \to d$

$3 \to 2$ — dominating edge



forward edge

Acylic graph

Detail the process of optimization of basic blocks.

Or

Basic block optimization

Or

transformation of basic blocks.

Or

different issues in code optimization

- Optimization is applied to the basic blocks after the intermediate code generation phase of the compiler.
- Optimization is the process of transforming a program that improves the code by consuming fewer resources and delivering high speed. In optimization, high-level codes are replaced by their equivalent efficient low-level codes.
- Optimization of basic blocks can be machine-dependent or machine-independent.
- These transformations are useful for improving the quality of code that will be ultimately generated from basic block.

There are two types of basic block optimizations:

- Structure preserving transformations
- Algebraic transformations

Structure-Preserving Transformations:

The structure-preserving transformation on basic blocks includes:

- Dead Code Elimination
- Common Subexpression Elimination
- Renaming of Temporary variables
- Interchange of two independent adjacent statements

1. Common sub-expression elimination :

- A common sub-expression is nothing but the expression which is already computed and the same expression is used again and again in the program.
- If the result of the expression not changed then we eliminate computation of same expression again and again.

a=b+c	a=b+c
b=a-d	b=a-d
C=b+c	c=b+c
d=a-d	d=b

Dead Code Elimination:

- Dead code is defined as that part of the code that never executes during the program execution.
- So, for optimization, such code or dead code is eliminated.
- The code which is never executed during the program (Dead code) takes time so, for optimization and speed, it is eliminated from the code.
- Eliminating the dead code increases the speed of the program as the compiler does not have to translate the dead code.

X=5

Y=z+3

Z=y-z since x is dead , so remove dead code x=5

3.Renaming of Temporary variables

- Statements containing instances of a temporary variable can be changed to instances of a new temporary variable without changing the basic block value.
- **Example:** Statement t = a + b can be changed to x = a + b where t is a temporary variable and x is a new temporary variable without changing the value of the basic block.

4.Interchange of Two Independent Adjacent Statements:

- If a block has two adjacent statements which are independent can be interchanged without affecting the basic block value.

Example:

t1 = a + b **t2=a+b**

t2 = c + d **t1=c+d**

(a+b)*(c+d)

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

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

$t_3 = t_1 * t_2$

- These two independent statements of a block can be interchanged without affecting the value of the block.

Algebraic Transformation:

Countless algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set. Some of the algebraic transformation on basic blocks includes:

- Constant Folding
- Copy Propagation
- Strength Reduction
- Algebraic identities

1. Constant Folding:

Solve the constant terms which are continuous so that compiler does not need to solve this expression.

Example:

x = 2 * 3 + y ⇒ x = 6 + y (Optimized code)

2. Copy Propagation:

It is of two types, Variable Propagation, and Constant Propagation.

1.Variable Propagation:

x = y , z = x + 2

⇒ z = y + 2

(Optimized code)

2.Constant Propagation:

x = 3 **⇒ z = 3 + a** Optimized code

z = x + a

3. Strength Reduction:

Replace expensive statement/ instruction with cheaper ones.

x = 2 * y (costly) ⇒ x = y + y (cheaper)

x = 2 * y (costly) ⇒ x = y << 1 (cheaper)

Algebraic identities

$x+0=0+x=x$, $x-0=x$

$x*1=1*x=x$, $x/1=x$

Function preserving transformation in basic block

- Common sub-expression elimination
- Copy propagation
- Dead code elimination
- Constant folding

Code Optimization

- Code optimization is a crucial phase in compiler design aimed at enhancing the performance and efficiency of the executable code.
- By improving the quality of the generated machine code optimizations can reduce execution time, minimize resource usage, and improve overall system performance.
- This process involves the various techniques and strategies applied during compilation to produce more efficient code without altering the program's functionality.
- The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result

The compiler optimizing process should meet the following objectives:

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

When to Optimize?

Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase performance.

Why Optimize?

- Optimizing an algorithm is beyond the scope of the code optimization phase. So the program is optimized. And it may involve reducing the size of the code. So, optimization helps to:
- Reduce the space consumed and increases the speed of compilation.
- Manually analyzing datasets involves a lot of time. Hence, we make use of software like Tableau for data analysis.

Similarly, manually performing the optimization is also tedious and is better done using a code optimizer.

- An optimized code often promotes re-usability.

Types of Code Optimization

The optimization process can be broadly classified into two types:

Machine Independent Optimization: This code optimization phase attempts to improve the intermediate code to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.

Machine Dependent Optimization: Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

Machine independent optimization

1. Loop optimization
   - Code motion or frequency reduction(loop invariant computation)
   - Loop unrolling
   - Loop jamming or fusion
   - Reduction in strength
   - Induction variable elimination
2. Folding
3. Redundancy elimination
4. Agebraic identities

2 Marks

Machine independent optimization

1. Peephole optimization
   - Redundant load/store
   - Flow of control optimization
   - Use of machine idioms

Nonlinear type of intermediate code

A non-linear type of intermediate code is an intermediate representation where the flow of control is not strictly sequential — it includes branches, loops, or jumps, making the execution path non-linear.

```
if a < b goto L2
    t1 = a + b
    goto L3
L2: t1 = a - b
L3: c = t1
```

Why quadruples preferred over triples in a optimizing compiler?

Quadruples are preferred over triples in an optimizing compiler mainly because they simplify the handling of temporary results during code optimization and transformation.

Key reasons:

1.Stability of references:

- In triples, results are referred to by their position (e.g., (op, arg1, arg2) where arg1 or arg2 might be a previous triple index).
- In quadruples, results are stored in temporary variables (e.g., t1 = op arg1, arg2), so reordering instructions is easier without breaking references.

2.Ease of code movement:

- Optimizations like common subexpression elimination or code motion require moving code around.
- With quadruples, since operands refer to named temporaries (t1, t2, etc.), these moves don't affect operand referencing, unlike in triples where moving a statement changes its index and breaks references.

List out the criteria for code improving transformations.

- A transformation must preserve meaning of a program.
- A transformation must improve program by a measurable amount
- on average.
- A transformation must worth the effort.

What is the use of algebraic identities in optimization of basic blocks

Algebraic identities are used in basic block optimization to simplify expressions and reduce the number of operations, improving efficiency.

Uses:

Eliminate redundant operations:

Example: x * 1 → x, x + 0 → x

Constant folding:

Example: 3 * 4 → 12 done at compile time

Strength reduction:

Replace expensive operations with cheaper ones

Example: x * 2 → x + x

Loop optimization:

It is a technique in which optimization is performed on the loops.

Code motion

- Code motion is a technique which moves the code outside the loop.
- If some expression in the loop whose result remains unchanged even after executing the loop for several times, then such an expression should be placed just before the loop (i.e., outside the loop)..

- Code motion is done to reduce the execution time of the program.

Before code motion

```
for (int i = 0; i < n; i++) {
   y = a * b;    // loop-invariant (same in every iteration)
   x[i] = y + i;
}
```

After code motion

```
y = a * b;      // moved outside the loop
for (int i = 0; i < n; i++) {
   x[i] = y + i;
}
```

1)T1 = 4 x I

T2 = addr(A) – 4

T3 = T2[T1]

T4 = addr(B) – 4

T5 = T4[T1]

T6 = T3 x T5

t7= PROD + T6

Prod=t7

t8 = I + 1

I=t8

IF I <=20 GOTO (1)

(three address code

- T2 = addr(A) – 4
- T4 = addr(B) – 4

1)T1 = 4 x I

T3 = T2[T1]

T5 = T4[T1]

T6 = T3 x T5

t7= PROD + T6

Prod=t7

t8 = I + 1

I=t8

IF I <=20 GOTO (1)

2. Induction variables :

- A variable x is called an induction variable of loop L if the value of variable gets changed every time.
- It is either decremented or incremented by some constant.

Numerical Remove the induction variable

```
for (i = 0; i < n; i++) {
    j = 4 * i;
    a[i] = b[j];
}
```

i = 0 → j = 4 * 0 = 0
i = 1 → j = 4 * 1 = 4
i = 2 → j = 4 * 2 = 8

- i is a primary induction variable.
- j is also an induction variable because it's linearly dependent on i:

  j = 4 * i
- Every time i increases by 1, j increases by 4.
- But in this version, j = 4 * i is computed again in every iteration — which involves multiplication (more costly)
- Goal: Replace repeated computation (4 * i) with a simple update (j += 4).

```
for (i = 0, j = 0; i < n; i++, j += 4) {
    a[i] = b[j];
}
```

First: i = 0, j = 0
Then: i = 1, j = 0 + 4 = 4
Then: i = 2, j = 4 + 4 = 8

Removes repeated multiplication

- Uses only addition (j += 4), which is faster
- Improves loop performance

Three address code:

1: i = 0

2: L1: t1 = 4 * i // recomputed every time

3: t2 = b[t1]

4: a[i] = t2

5: i = i + 1

6: if i < n goto L1

Rem induction variab
Apply Induction process

TAC After Induction Variable Elimination:

1: i = 0

2: j = 0 // j tracks 4 * i directly

3: L1: t1 = b[j]

4: a[i] = t1

5: i = i + 1

6: j = j + 4 // no multiplication, just addition

7: if i < n goto L1

3. loop unrolling

Loop Unrolling is an optimization where the loop body is repeated multiple times within a single iteration to reduce loop overhead and increase performance.

Original Loop

```
for (int i = 0; i < 4; i++) {
    a[i] = a[i] * 2;
}
```

Unrolled version

```
a[0] = a[0] * 2;
a[1] = a[1] * 2;
a[2] = a[2] * 2;
a[3] = a[3] * 2;
```

No loop.
Faster for small arrays.
Removes loop control overhead (i++, i < 4, etc.

Benefits of Loop Unrolling:

- Reduces loop overhead (fewer jumps, comparisons)
- Can improve performance in tight loops
- Better CPU pipelining and instruction-level parallelism

4. loop jamming (Loop fusion)

- Loop Jamming (also called Loop Fusion) is a compiler optimization technique where two or more loops that iterate over the same range are combined into one loop.
- Reduces loop overhead (fewer loop control instructions)
- Improves cache locality
- Makes code more efficient

Before Loop Jamming (Two separate loops):

```
for (int i = 0; i < n; i++) {
  a[i] = b[i] + c[i];
}
for (int i = 0; i < n; i++) {
  d[i] = a[i] * 2;
}
```

After Loop Jamming (Fused into one loop):

```
for (int i = 0; i < n; i++)
 {
a[i] = b[i] + c[i];
 d[i] = a[i] * 2;
}
```

5. Reduction in strength (strength reduction)

Strength Reduction is a compiler optimization that replaces expensive operations (like multiplication or division) with cheaper ones (like addition or bit shifts).

Why?

Operations like * and / are slower than + or << (bitwise shift)

Before strength reduction

```
for (int i = 0; i < n; i++) {
    a[i] = 5 * i;
}
```

After strength reduction

```
int t = 0;
for (int i = 0; i < n; i++) {
    a[i] = t;
    t = t + 5;
}
```

i=0 a[0]=0
i=1 a[1]=5
i=2 a[2]=10

i=0 a[0]=0
i=1 a[1]=5
i=2 a[2]=10

DAG

- The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, visualize the flow of values between basic blocks, and provide optimization techniques in basic blocks.
- .Directed acyclic graphs are a type of data structure and they are used to apply transformations to basic blocks.

- The Directed Acyclic Graph (DAG) facilitates the transformation of basic blocks.
- DAG is an efficient method for identifying common sub-expressions.
- It demonstrates how the statement's computed value is used in subsequent statements.
- A DAG gives picture of how the value computed by each statement in
- the basic block is used in the subsequent statement of the block.
- Constructing a DAG from three address statement is a good way of determining common sub-expressions within a block.

A DAG for a basic block has following properties :

1. Leaves are labeled by unique identifier, either a variable name or constants.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels.

- Since, DAG is used in code optimization and output of code optimization is machine code and machine code uses register to store variable used in the source program.

Advantage of DAG :

1. We automatically detect common sub-expressions with the help of DAG algorithm.

2. We can determine which identifiers have their values used in the block.

3. We can determine which statements compute values which could be used outside the block.

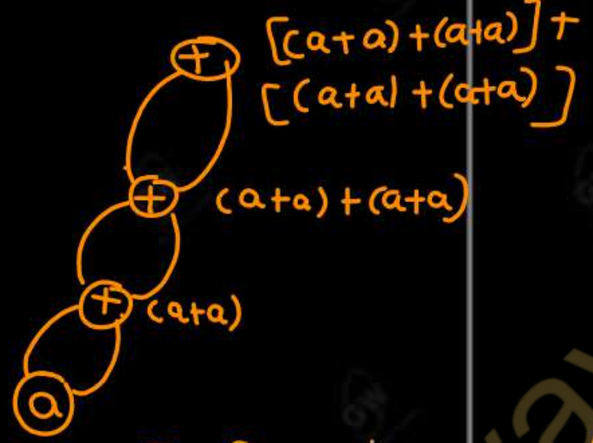
Draw the DAG for the following

Expression

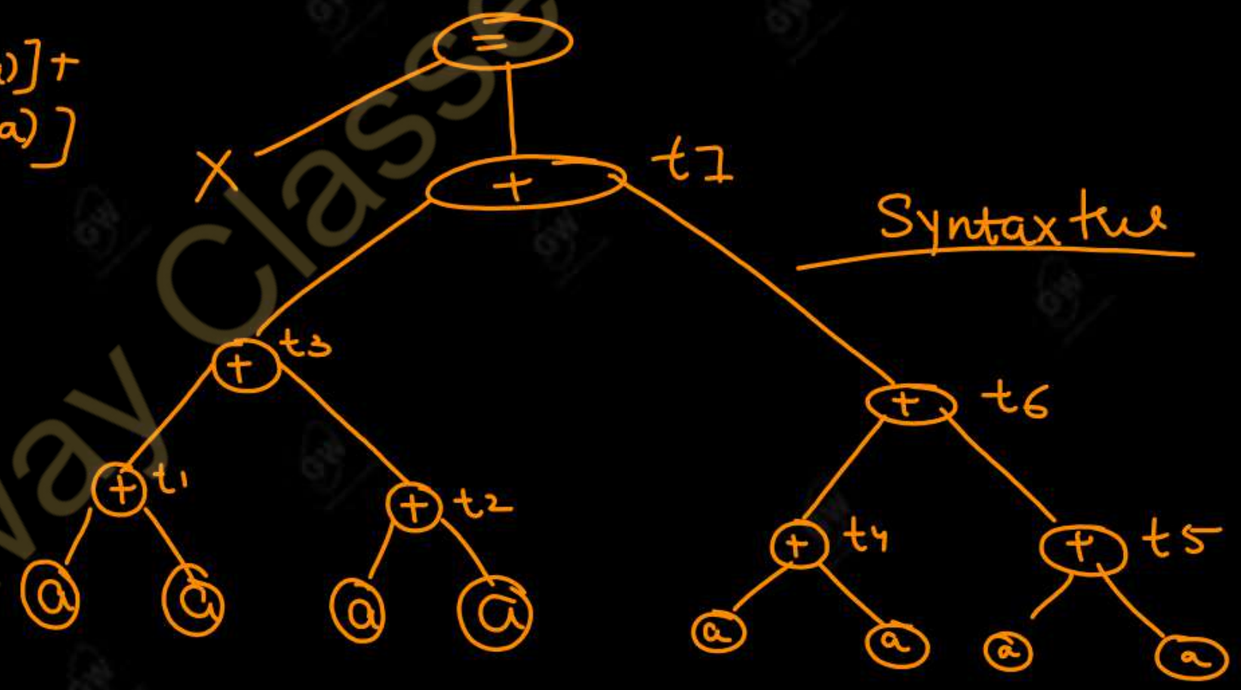
$X=(((a+a)+(a+a))+((a+a)+(a+a)))$

$t_1 = a+a$
$t_2 = a+a$
$t_3 = t_1+t_2$
$t_4 = a+a$
$t_5 = a+a$
$t_6 = t_4+t_5$
$t_7 = t_3+t_6$
$X = t_7$

three address Code



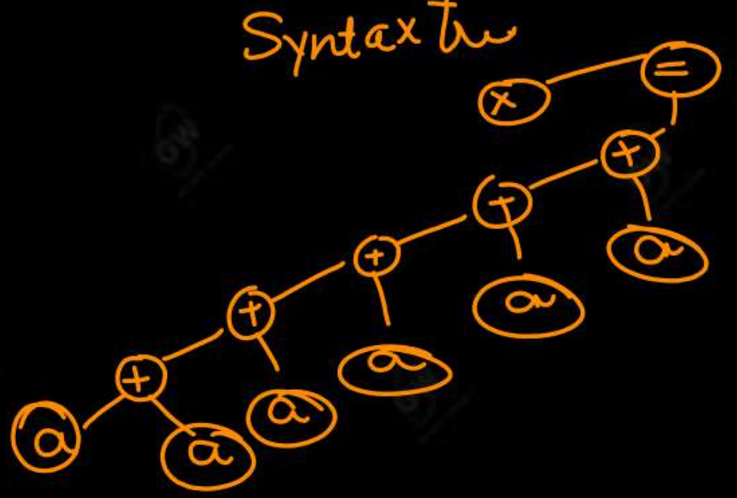
DAG

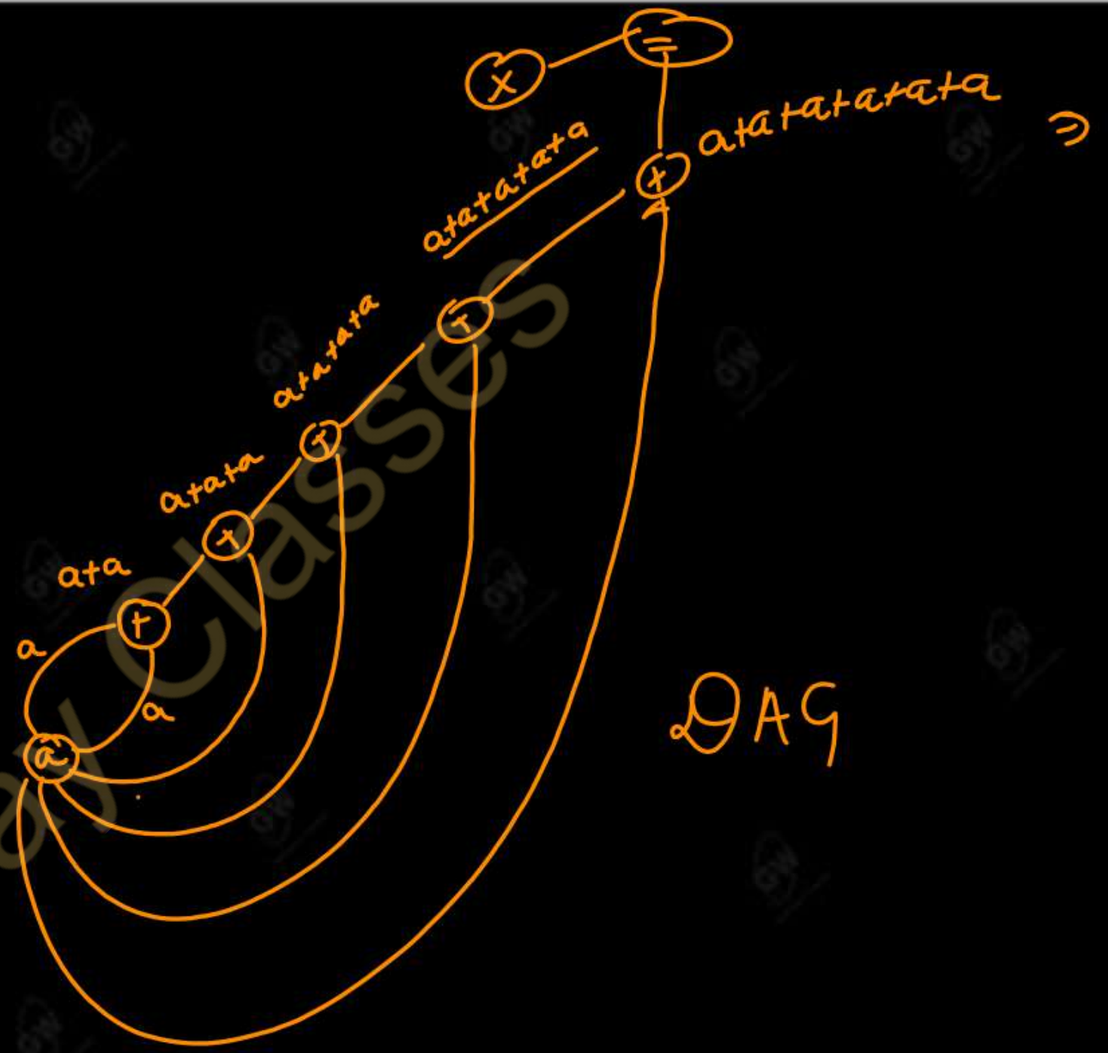


Draw the DAG for the following

Expression

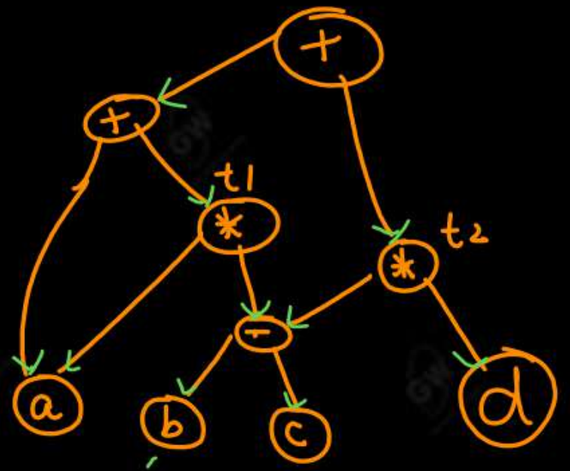
X=a+a+a+a+a+a





Draw the DAG for the following Expression

a+a*(b-c)+(b-c)*d (AKTU 2016-17)



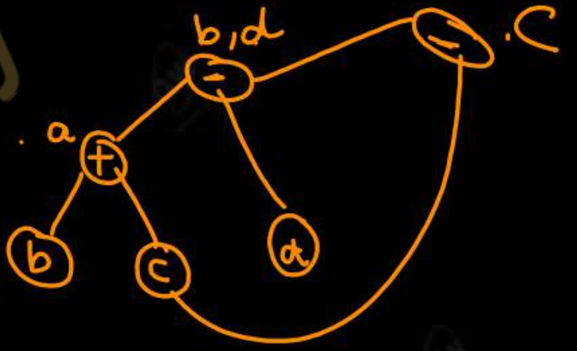
Draw the DAG for the following Expression

a=b+c

b=a-d

C=b+c

d=a-d (AKTU 2015-16/2018-19)



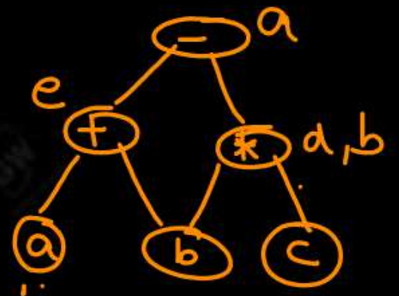
DAG

d=b*c

e=a+b

b=b*c

a=e-d



DAG

1 peephole optimization

- Peephole optimization is a type of compiler optimization technique where a small set of instructions (a "peephole") in the generated code is examined and replaced with more efficient instructions without changing the output.
- Redundant load/store elimination
- a=b+c ,d=a+e

```
LOAD R0,b      R0 ← M[b]
ADD R0,c       R0 ← R0+M[c]
STORE a,R0     a ← R0
```

```
LOAD R0,a      R0 ← M[a]
ADD R0,e       R0 ← R0+M[e]
STORE d,R0     M[d] ← R0
```

After removing redundant instructions

```
LOAD R0,b      R0 ← M[b]
ADD R0,c       R0 ← R0 + M[c]
ADD R0,e       R0 ← R0 + M[e]
STORE d,R0     M[d] ← R0
```

2 Flow of control optimization

- Flow of control optimization improves the structure of jump and branch instructions in code to make execution faster and more efficient.
- Reduce unnecessary jumps, simplify branches, and make execution more direct.

2.1. Avoid jumps on jumps

Before optimization

```
JMP L1      ; Jump to label L1
...
L1: JMP L2  ; L1 just jumps to L2
```

What's wrong?

Jumping to L1, which only jumps to L2, is inefficient

```
JMP L2      ; Jump directly to L2
```

2.2.Elimination of dead code:

```
Int i=0;
If (i==1)
{
Printf("hello");
}
```

dead code

3. use of machine idioms

i++ i=i+1


```
LOAD R0,I       R0 ← M[I]
ADD R0,#1       R0 ← R0 + 1
STORE I,R0      M[I] ← R0
```

(Assembly Lang)

fro this machine idioms are used

INC i

scope of optimization

1. Local optimization

Local optimization is a type of compiler optimization that is applied within a basic block — a straight-line sequence of code with no jumps or branches (except at the entry and exit).

Goal:

To improve performance and reduce code size by optimizing instructions locally (block-by-block), without needing to analyze the whole function or program.

Why Local?

- Easy to analyze: no need to track across branches or loops.
- Faster to apply during compilation.
- Still gives good performance improvements.

Common Local Optimizations:

- Constant folding ,Constant propagation ,Common subexpression elimination (CSE) ,Dead code elimination – Strength reduction

GLOBAL OPTIMIZATION

- Global optimization refers to compiler optimizations applied across multiple basic blocks, typically at the function or program level, unlike local optimization which is confined to one block.

Goal:

To improve performance and efficiency by analyzing control flow and data flow across the entire function or program.

Why Global?

- Some inefficiencies can't be seen within one block.
- Tracks variable values, expression usage, and control flow through loops, branches, and calls.

Common Global Optimizations:

- Global Common Subexpression Elimination (GCSE)
- Loop Invariant Code Motion (LICM)
- Global Constant Propagation
- Dead Code Elimination (across blocks)

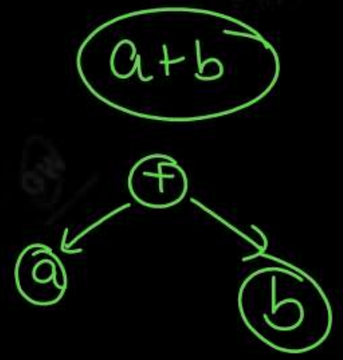
Write an algorithm to create a DAG

Input : A basic block.

Output : A DAG with label for each node (identifier).

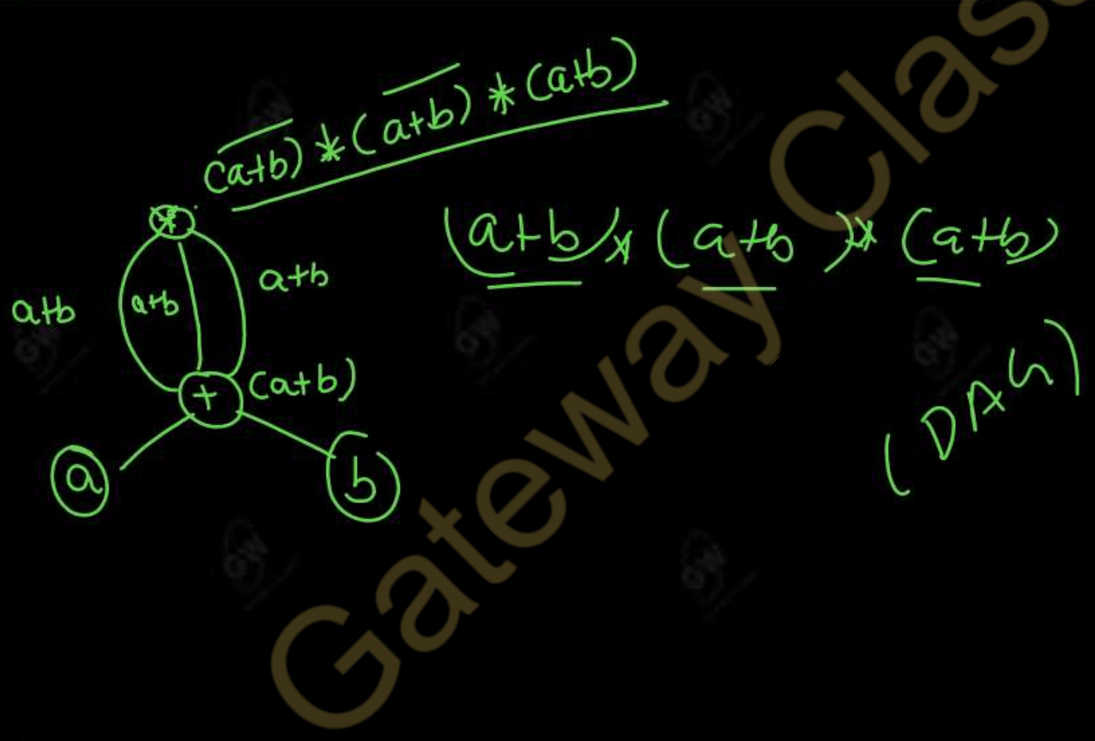
Method :

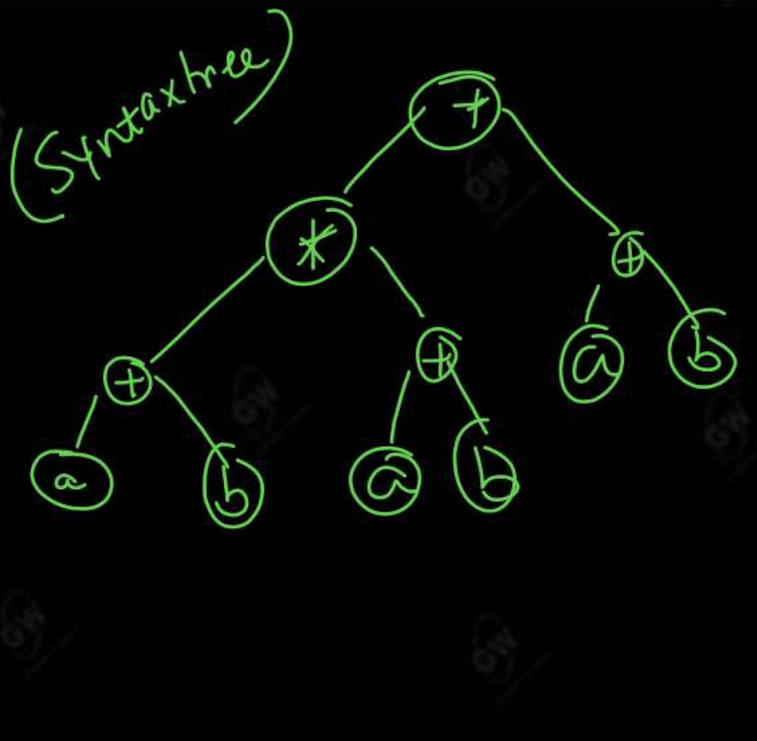
1. Create nodes with one or two left and right children.
2. Create linked list of attached identifiers for each node.
3. Maintain all identifiers for which a node is associated.
4. Node (identifier) represents value that identifier has the current point in DAG construction process. Symbol table store the value of node (identifier).
5. If there is an expression of the form x = y op z then DAG contain "op" as a parent node and node(y) as a left child and node(z) as a right child.



Feature	Syntax Tree	DAG (Directed Acyclic Graph)
Structure	Tree (hierarchical, no shared nodes)	Graph (nodes can be shared, no cycles)
Redundancy	No sharing of sub-expressions (each node is unique)	Shared sub-expressions (reused nodes for common sub-expressions)
Node Representation	Represents operators and operands	Represents sub-expressions or computations
Use Case	Used for **parsing** and analyzing syntax of expressions	Used for **optimization**, code generation, and expression simplification
Memory Efficiency	Less efficient (duplicates common sub-expressions)	More efficient (reuses common sub-expressions)
Edges	One edge from parent to child	Multiple edges may lead to the same node
Type of Structure	Tree (only one path from root to any node)	Acyclic graph (multiple paths, no cycles)

Feature	Syntax Tree	DAG (Directed Acyclic Graph)
Computation Efficiency	Not focused on computation reduction	Helps in reducing redundant computations
Typical Example	Expression parsing in compilers	Optimized code representation in compiler backends





Give the algorithm for the elimination of local and global common sub-expressions algorithm with the help of example.

Algorithm for elimination of local common sub-expression : DAG algorithm is used to eliminate local common sub-expression.

Algorithm for elimination of global common sub-expressions

GCSE eliminates repeated computations of the same expression across basic blocks in a control flow graph (CFG), improving efficiency by computing the expression once and reusing the result.

Following expressions are used :

a. avail[B] = set of expressions available on entry to block B

b. exit[B] = set of expressions available on exit from B

c. killed[B] = set of expressions killed in B/Expressions whose operands are reassigned in block B

d. defined[B] = set of expressions defined in B

e. exit[B] = avail[B] – killed[B] + defined[B]

Algorithm Steps

Compute defined[B] and killed[B] for each basic block.

Iteratively compute avail[B] and exit[B] until reaching a fixed point.

For each statement s: a = b op c in block B:

- If b op c is available at entry of B and b or c are not redefined before s:
  - Find the original definition of b op c.
  - Assign its result to a new temp variable, say t = b op c.
  - Replace a = b op c with a = t in B.

Input Program

B1:

```
x = a + b;
```

B2:

```
c = a + b;
```

B3:

```
d = a + b;
```

Step 1: Identify Expressions in Each Block

Block	defined[B]	killed[B]
B1	{a+b}	if a or b is reassigned here
B2	{a+b}	same logic
B3	{a+b}	same logic

STEP 2:Compute avail/exit Sets

Block	avail[B]	exit[B]
B1	Ø	{a+b}
B2	{a+b}	{a+b}
B3	{a+b}	{a+b}

Step 3: Apply GCSE

In B2 and B3, a + b is available at the block entry and not killed

a+ b is computed only once and reused.

B1:

```
t = a + b;
x = t;
```

B2:

```
c = t;
```

B3:

```
d = t;
```

Benefits of This Algorithm

- Reduces repeated expression evaluations across blocks.
- Saves CPU cycles and register/memory operations.
- Helps generate more optimized intermediate code

Another Example:

B1:

a = b + c;

B2:

x = b + c;

y = a * d;

B3:

z = b + c;

b, c, and d are not modified after B1.

The control flow is: B1 → B2 → B3.

Block	defined[B]	killed[B]
B1	{b + c}	Ø (no operand modified)
B2	{b + c, a * d}	Ø (no operand modified)
B3	{b + c}	Ø

Compute avail[B] and exit[B]

We compute avail[B] and exit[B] iteratively until a fixed point is reached.

Block B1

avail[B1] = Ø (entry block)

exit[B1] = avail[B1] - killed[B1] ∪ defined[B1] = Ø ∪ {b + c}

= {b + c}

Ø - Ø ∪ {b+c}

Block B2

avail[B2] = exit[B1] = {b + c}

exit[B2] = {b + c, a * d}

Block B3

avail[B3] = exit[B2] = {b + c, a * d}

exit[B3] = {b + c} (since only b + c is defined in B3)

Step 3: Identify Common Subexpressions

We now look at each block for expressions that are:

Available at the block's entry, and

Whose operands are not redefined before their use.

Block B2

x = b + c;

b+ c is available at entry.

b, c are not redefined in B2.

Can eliminate this subexpression.

y = a * d;

a * d is not available at entry (a was just defined in B1).

Cannot eliminate

Block B3

z = b + c;

b + c is available.

→ Can eliminate this subexpression.

B1:

t = b + c;

a = t;

B2:

x = t;

y = a * d;

B3:

z = t;

Global data flow analysis

- Global data flow analysis collects the information about the entire program and distributed it to each block in the flow graph.
- The gathered information helps to achieve number of optimizations.
- Global data flow analysis used to solve a specific problem user definition chaining
- "given that the identifier A is used at a point p, at what point could the value of A used at p haS been defined

reaching definition:

The reaching definition implies the determination of definition that apply at a point p in a flow graph

It follows the following steps:

1.Assign a distinct number to each definition such that

d1,d2,d3,d4,d5....dn

2. For each variable i , make a list of all definition in entire program where it is used

3. Each basic block B, compute the following:

GEN(B) the set GEN(B)consists of all the definition generated in a block B.

KILL(B): the set of all definition outside block B that defines the same variable having definition in block B also.

4.For all the basic block B compute the following:

IN(B): the set of all the definition reaching the point just before the first statement of block B.

OUT(B): The set of all definition reaching the point just after the last statement of basic block B.

Data flow equations:

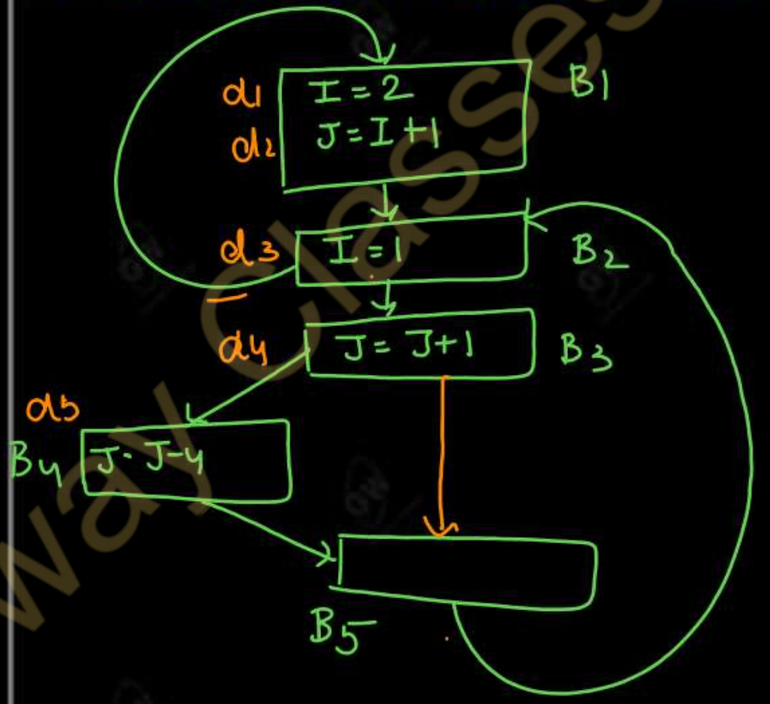
there are two set of equation that called data flow equations:

1 out[B]=IN[B]-KILL[B]UGEN[B]
(IN[B]AND(~KILL[B])U GEN[B]

2 IN[B]= U OUT[B]
P IS PREDECESSOR OF B

CONSIDER THE FLOW GRAPH OF THE GIVEN FIGURE:

- FIND GEN AND KILL FOR EACH BLOCK
- IN AND OUT FOR RACHING DEFINITION



initially IN[B]=Φ OUT[B]=GEN[B]

FOR PASS- 1

IN[B1]=OUT[B2]=00100

OUT[B1]= IN[B1] ∩(~KILL[B1])UGEN[B1]

(00100) ∩(~00111)U11000

(00100) ∩(11000)U11000

00000U11000

11000

IN[B2]=OUT[B1]UOUT[B5]

11000U00000

11000

OUT[B2]= IN[B2] ∩(~KILL[B2])UGEN[B2]

11000∩(~10000)U00100

11000∩01111U00100

01000U00100=01100

IN[B3]=OUT[B2]=01100

OUT[B3]= IN[B3] ∩(~KILL[B3])UGEN[B3]

01100∩(~**01001**)U00010

01100∩10110U00010

00100U00010= 00110

IN[B4]=OUT[B3]=00110

OUT[B4]= IN[B4] ∩(~KILL[B4])UGEN[B4]

00110∩(~**01010**)**U00001**

00110∩10101U00001

00100U00001=00101

IN[B5]=OUT[B3]UOUT[B4] =00110 U 00101 =00111

OUT[B5]=**IN[B5]** ∩(~KILL[B5])UGEN[B5]

00111∩(~**00000**)U(00000)=00111

BLOCK B	GEN[B]	BIT VECTOR d1,d2,d3,d4,d5	Kil[B]	BIT VECTOR d1,d2,d3,d4,d5
B1	[d1,d2]	11000	[d3,d4,d5]	00111
B2	[d3]	00100	[d1]	10000
B3	[d4]	00010	[d2,d5]	01001
B4	[d5]	00001	[d2,d4]	01010
B5	Φ	00000	Φ	00000

BLOCK	IN[B]	OUT[B]
B1	00000	11000
B2	00000	00100
B3	00000	00010
B4	00000	00001
B5	00000	00000

BLOCK	PASS1		PASS2		PASS3		PASS4	
	IN[B]	OUT[B]	IN[B]	OUT[B]	IN[B]	OUT[B]	IN[B]	OUT[B]
B1	00100	11000	01100	11000	01111	11000	01111	11000
B2	11000	01100	11111	01111	11111	01111	11111	01111
B3	01100	00110	01111	00110	01111	00110	01111	00110
B4	00110	00101	00110	00101	00110	00101	00110	00101
B5	00111	00111	00111	00111	00111	00111	00111	00111

SIMILALRY CALULATION FOR PASS2 PASS 3 AND PASS 4 we get same result in pass 3 and pass 4 SO WE WILL STOP AFTER PASS 4

Pass2

$In[B1] = Out[B2] = 01100$

$Out[B] = IN[B1] \cap (\sim KILL(B1) \cup Gen[B1])$

$01100 \cap 11000 \cup 11000$

$01000 \cup 11000$

11110 G

$In[B2] = Out[B1] \cup Out[B5]$

$11100 \cup 00011$

11111

$Out[B2] = IN[B2] \cap (\sim KILL B2) \cup Gen(B2)$

$11111 \cap 01111 \cup 00100$

$01111 \cup 00100$

01111

Local and Global Elimination of Common Subexpressions

Common Subexpression Elimination (CSE) is a compiler optimization where redundant expressions are identified and reused instead of recomputed.

1. Local Common Subexpression Elimination

Scope:

Works within a basic block (a straight-line code sequence with no jumps).

Idea:

If an expression like a + b is computed more than once in the same basic block, and its operands are not modified, compute it once and reuse the result

```
// Basic block (local)
x = a + b;
y = a + b;   // redundant
```

After Local CSE:

```
t = a + b;
x = t;
y = t;
```

Global Common Subexpression Elimination

Scope:

- Works across basic blocks, considering control flow.

Idea:

- If an expression like a + b is computed in one block and again in another block (with no changes to a or b in between), eliminate the second computation and reuse the earlier result.

```
B1:
t1 = a + b;

B2:
...       // no change to a or b
t2 = a + b;  // redundant across blocks
```

After global CSE

```
B1:
t1 = a + b;

B2:
...
t2 = t1;
```

Feature	Local CSE	Global CSE
Scope	Within a basic block	Across multiple basic blocks
Based on	Simple linear scan	Data flow analysis
Control flow considered	❌ No	✅ Yes
Complexity	Low	Higher (needs CFG and data flow info)
Example use	Optimizing repeated code in a block	Optimizing loops or repeated logic in functions

control flow graph.

How global flow analysis used in code optimization?

Global flow analysis is used in code optimization to gather information about how data moves across all basic blocks of a program. It helps identify redundant computations, unused variables, and opportunities to reuse results

what is next use information?

- Next-use information tells the compiler where a variable will be used next in the program. It is used in code generation, especially for register allocation and instruction selection.

To help the compiler decide:

- Which variables to keep in registers
- Which variables can be safely overwritten
- Which values should be stored back to memory

Used In

- Register allocation: If a variable has no next use, its register can be freed.
- Code generation: Helps generate efficient machine instructions by avoiding unnecessary memory operations.

Next-use information is critical for generating efficient code. It helps the compiler:

- Use registers smartly,
- Reduce memory traffic,
- And eliminate unnecessary computations.

what are the issue in code loop optimization:

- Cod emotion
- Induction variable
- Reduction in strength
- Loop unrolling
- Loop fusion

What is code flow analysis

- Code Flow Analysis is the process of analyzing the order and paths through which control (execution) moves within a program. It builds a Control Flow Graph (CFG) and helps the compiler understand how different parts of the code interact.

It is the foundation for many optimizations and analyses like:

- Dead code elimination
- Loop optimization
- Data flow analysis
- Register allocation

- Modern compilers don't just translate code — they try to generate the most efficient machine code possible. To do this, they must:
- Understand which instructions execute after which
- Know what conditions affect the flow of execution
- Detect unreachable or redundant code

Data flow analysis

- Data flow analysis is a process in which the values are computed using data flow properties.
- In this analysis, the analysis is made on data flow.
- A program's Control Flow Graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.

- A simple way to perform data flow analysis of programs is to set up data flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fix point.
- Reaching definitions is used by data flow analysis in code optimization.

Global Data flow Data analysis

Reaching definition

- A definition D reaches at point p if there is a path from D to p along which D is not killed.
- A definition D of variable x is killed when there is a redefinition of x.
- The definition d1 is said to a reaching definition for block B2. But the definition d1 is not a reaching definition in block B3, because it is killed by definition d2 in block B2.

ROLE OF MACRO IN PROGRAMMING LANGAUGE

Role	Description
Code Reusability	Define a block of code once and reuse it many times. Reduces duplication.
Simplification of Code	Make complex expressions or frequently used code **shorter and cleaner**.
Conditional Compilation	Enable/disable parts of code using **#ifdef, #ifndef** (in C/C++).
Parameterization	Accept arguments like functions (e.g., #define SQUARE(x) ((x)*(x))).
Portability	Write platform-specific code using macro conditions.
Metaprogramming	In languages like Lisp, macros allow programs to **generate code** at compile time.

EXAMPLE OF MACRO

```
#define MAX(a, b)
((a) > (b) ? (a) : (b))
int x = MAX(10,
20);
```

Viable prefix

- A viable prefix is a prefix of a right sentential form that does not continue past the "handle" of the sentential form
- A viable prefix has no handle or just one handle on the extreme right which can be reduced.
- S->XX, X->aX|b shift reduce parser (bottom up parser) perform

Right most derivation in Reverse order

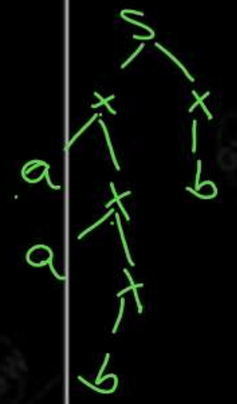
LHS

S → XX

X → aX | b

X → b

aabb



handle	Viable prefix
aabb	a, aa,aab
aaXb	a,aa,aaX
aXb	a,aX
Xb	X,Xb
XX	X,XX
S	

$S \to XX$

$X \to aX \mid b$

Viable prefix

{ a, aa, aab, aax, aXx, Xb, XX }

Stack	Input tape	Action
$	aabb$	Shift a
$a	abb$	Shift a
$aa	bb$	Shift b
$aab	b$	Reduce X→b
$aaX	b$	Reduce X→aX
$aX	b$	Reduce X→aX
$X	b$	Shift b
$Xb	$	Reduce X→b
$XX	$	
$S	$	

- The set of prefix of right sentential form that can appear on the Stack of Shift Reduce parser

Consider the following three address code segments :

PROD : = 0

1. I : = 1

2. T1 : = 4*I

3. T2 : = addr(A) – 4

4. T3 : = T2 [T1]

5. T4 : = addr(B) – 4

6. T5 : = T4[T1]

7. T6 : = T3*T5

8. PROD : = PROD + T6

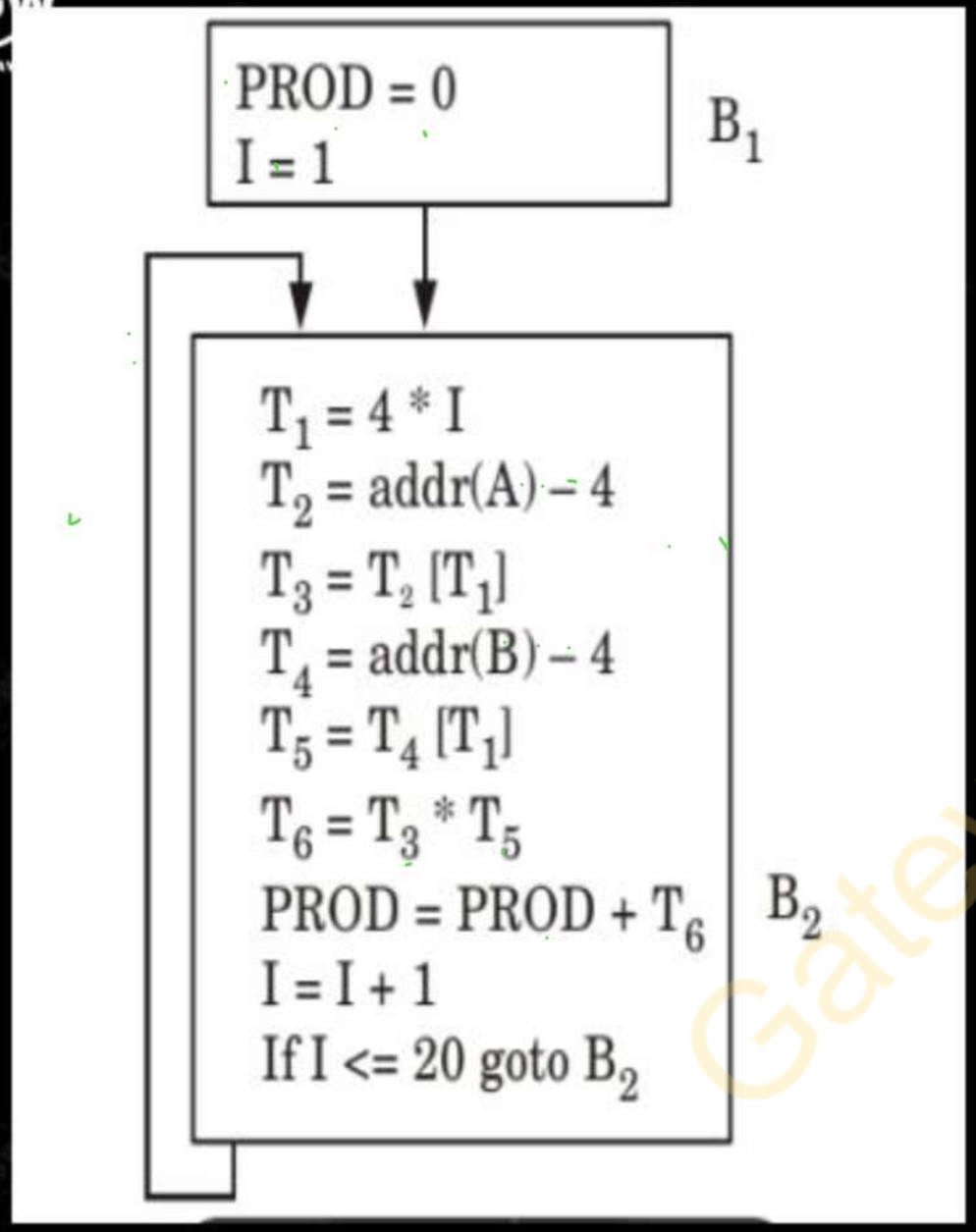
9. I : = I + 1

10. If I < = 20 goto (3)

a. Find the basic blocks and flow graph of above sequence.

b. Optimize the code sequence by applying function preserving transformation optimization technique.

c.Perform loop optimization.



Function preserving transformation :

1. Common sub-expression elimination : No any block has any sub expression which is used two times. So, no change in flow graphs.

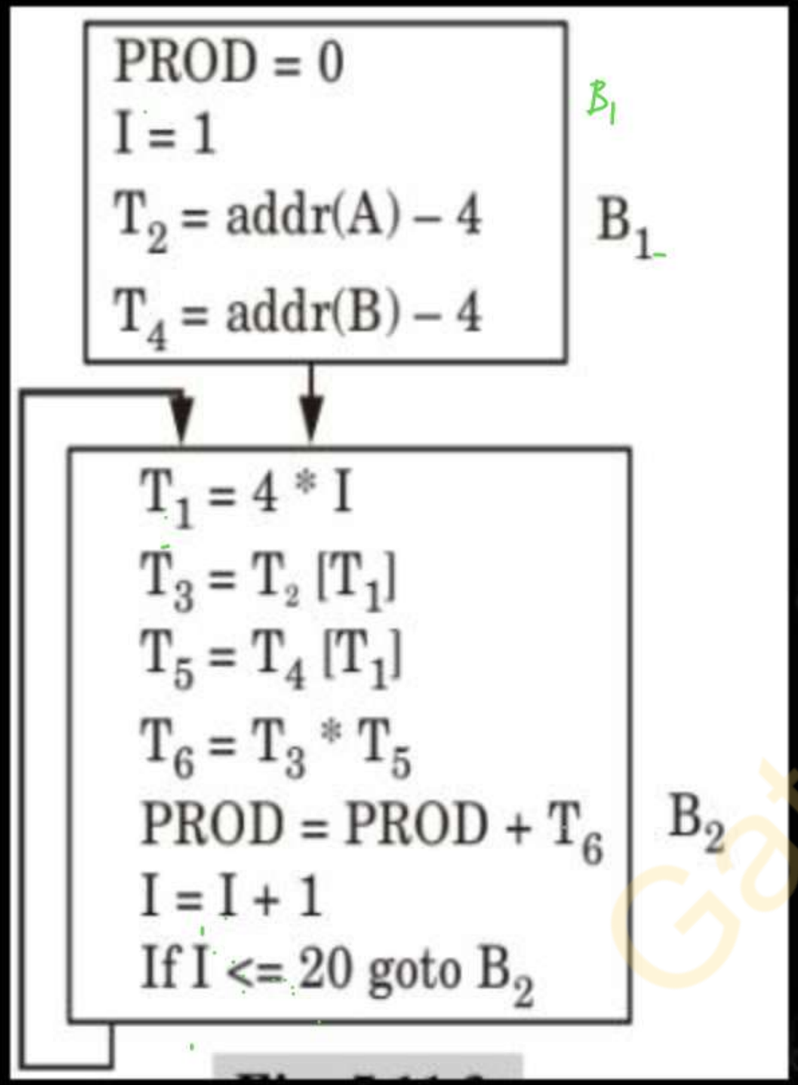
2. Copy propagation : No any instruction in the block B2 is direct assignment i.e., in the form of x = y. So, no change in flow graph and basic block.

3. Dead code elimination : No any instruction in the block B2 is dead. So, no change in flow graph and basic block.

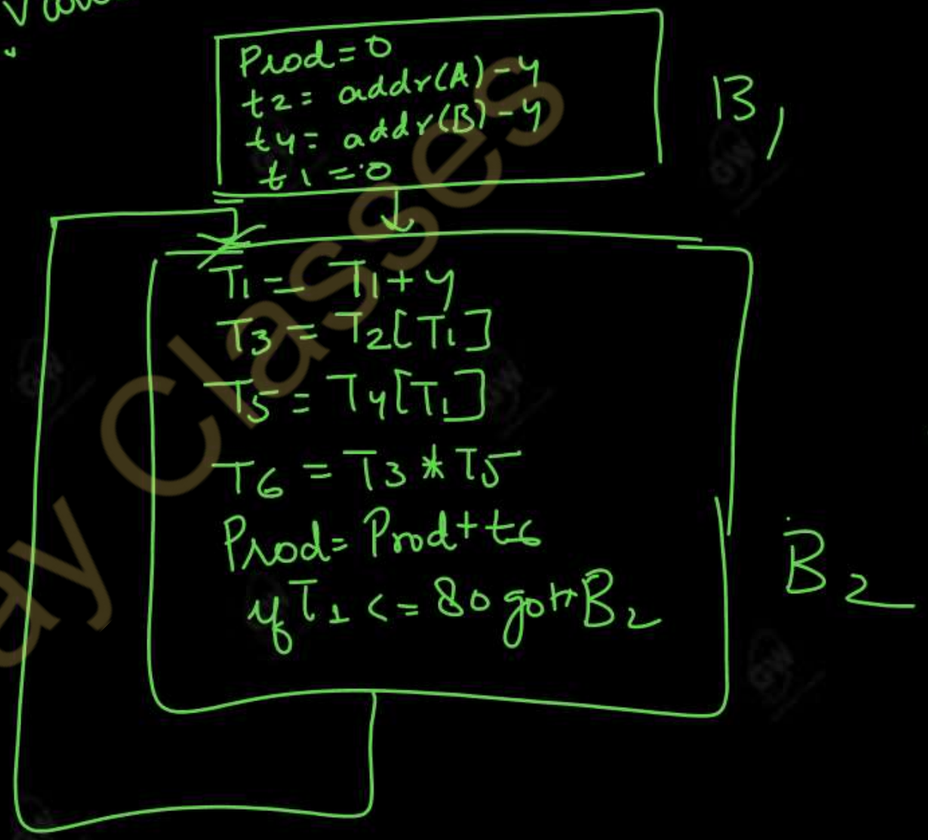
4. Constant folding : No any constant expression is present in basic block. So, no change in flow graph and basic block.

Code motion :

Loop optimization



Induction variable & Reduction in strength



How to perform register assignment for outerloop

- Register assignment for an outer loop means deciding which variables should be stored in CPU registers (instead of memory) during the execution of an outer loop to improve performance.

Construct Control Flow Graph (CFG)

Identify loops and especially the outer loop structure.

Perform Liveness Analysis

Determine which variables are live at each point in the loop. A variable is live if its value will be used later.

Find Loop Invariants

Variables that do not change inside the loop body can be computed outside the loop.

These can also be assigned to registers early.

Perform Usage Count

Count how many times each variable is used inside the loop.

Variables with higher usage frequency are better candidates for register allocation.

Build Interference Graph

Nodes: Variables

Edge: Between variables that are live at the same time

This graph helps in assigning non-conflicting variables to registers.

Graph Coloring

Apply a coloring algorithm to assign registers.

If registers are fewer than needed, spill some variables (store them in memory).

Assign Registers

Assign actual physical or virtual registers to variables.

Loop Nesting Consideration

Outer loop variables should be given priority over inner loop variables because they remain active over a larger region.

Gateway Classes

Empowering Learners, Trasforming Futures

Thank You

Helpline No- 7819 0058 53