Unit - IV

Machine Independent Optimization

A) The Principal Sources of Optimization :-

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 so that faster running machine code will result. 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 the performance.

There are five code optimization techniques :

1. Common Sub Expression Elimination.
2. Compile time Evaluation
   a. Constant folding
   b. Constant propagation
3. Code movement
4. Dead Code Elimination
5. Strength Reduction.

1. Common SubExpression Elimination:

It is an expression which appears repeatedly in the program, which is computed previously but the values of variables in expression haven't changed. This technique replaces the redundant expression each time it is executed.

Encountered.

unoptimized code	optimized code
a = b + c	a = b + c
b = a - d	b = a - d
c = b + c	c = b + c
d = a - d	

2. Compile time Evaluation: Evaluation is done at Compile time instead of Runtime.

a. Constant folding: Evaluate the Expression and Submit the result.

Ex: area = (22/7) * r * r

Here 22/7 is Calculated & result 3.14 is replaced.

So, area = 3.14 * r * r

b. Constant Propagation: Here constant replace a variable.

ex: $P_i = 3.14 \quad r = 5$

area = $P_i * r * r$

area = $3.4 * 5 * 5$

3. Code movement: It is also called as code motion. It is a technique which moves the code outside the loop if it won't have any difference if it executed inside (or) outside the loop.

Ex: Un optimized code

```
for(i=0; i<n; i++)
{
x = y+z;
a[i] = 6*i;
}
```

optimized code

```
x = y+z;
for(i=0; i<n; i++
{
a[i] = 6*i;
}
```

4. Dead Code elimination:

It includes eliminate those statements which are never executed (or) if executed o/p is never used.

Ex: Un optimized code

```
i = 0;
if(i==1)
{ a = x+i; }
```

optimized code

```
i=0;
```

Ex: int add(int x, int y)

{ int z;

z = x+y;

return z;

Printf("%d", z);

}

optimized code

int add(int x, int y)

{

int z;

z = x+y;

return z;

}

5. Strength Reduction:

Replace the expensive operator to cheaper operator.

Ex: b = a*2

b = a+2

B) Introduction to Data Flow Analysis :-

Data-flow analysis is a technique used in compiler design to analyze how data flows through a program. It involves tracking the values of variables and expressions as they are computed and used throughout the program. The basic idea behind data flow analysis is to model the program as a graph, where nodes represent program statements and edges represent data flow.

Types of Dataflow Analysis :

1. Reaching Definitions Analysis :

This analysis tracks the definition of a variable (or) expression and determine the point in the program where the definition 'reaches' a particular use of the variable (or) expression.

2. Live variable Analysis :

This analysis determine the point of the program where a variable (or) expression is "live" meaning that its value is still needed for some future computation.

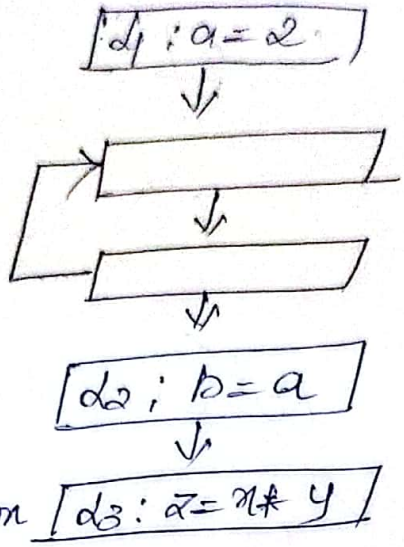
3. Available Expression Analysis:
This Analysis determines the point in the program where particular expressions is available, meaning that the it value has already been computed and can be reused.

4. Constant propagation Analysis:
This Analysis tracks the values of constants and determine the points of program where particular constant value is used.

Advantages:
1. Improved code quality
2. Better error detection
3. Increased understanding of program behaviours.

Basic Terminology:
* Definition point in a program containing some definition.
* Reference point in a program containing a reference to a data item.
* Evaluation points is a program

Containing evaluation of expression.

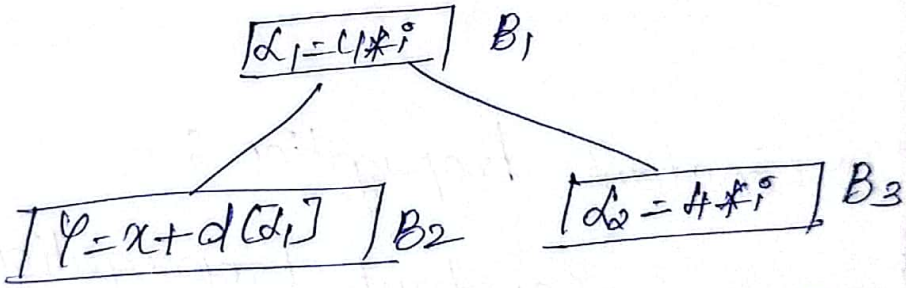


Definition point

Reference point -

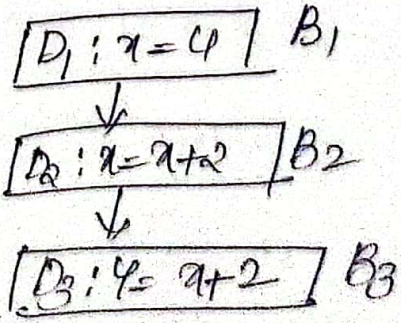
Evaluation

Available Expression:



Adv: It eliminates common sub expression.

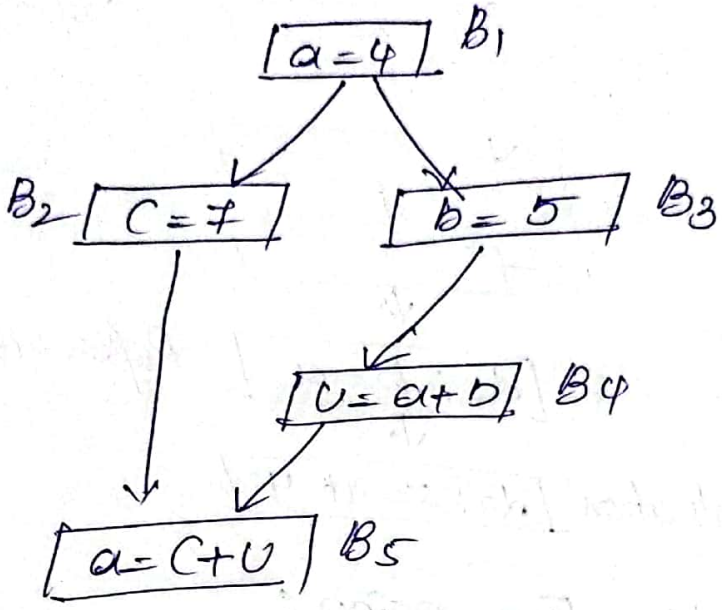
Reaching defination:



D_1 is reaching definition for B_2 but not for B_3 since it is killed by D_2.

Adv: It eliminates constant propagation.

Live Variable:



a is live in Block B_1, B_3, B_4 but killed at B_5

D) Constant Propagation:

Constant propagation is one of the local code optimization technique in Compiler Design. It can be defined as the process of replacing the constant value of variables in expression.

unoptimized code

Ex: a = 30

b = 20 - a/2

c = b * (30/a + 2) - a

Optimized Code:

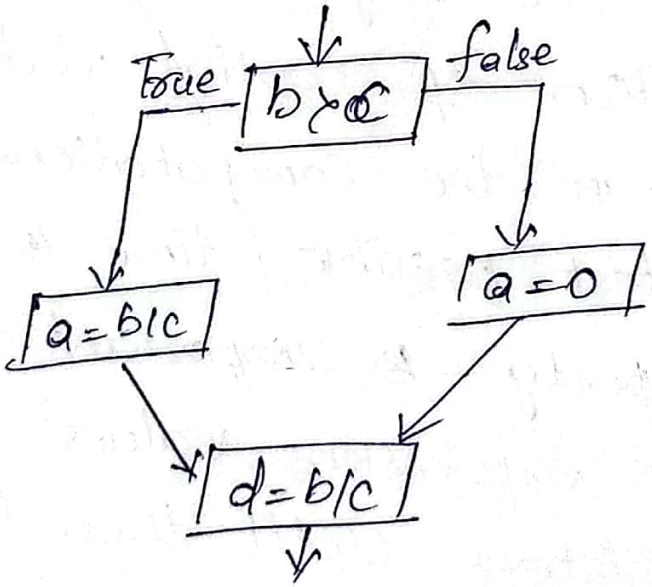
a = 30

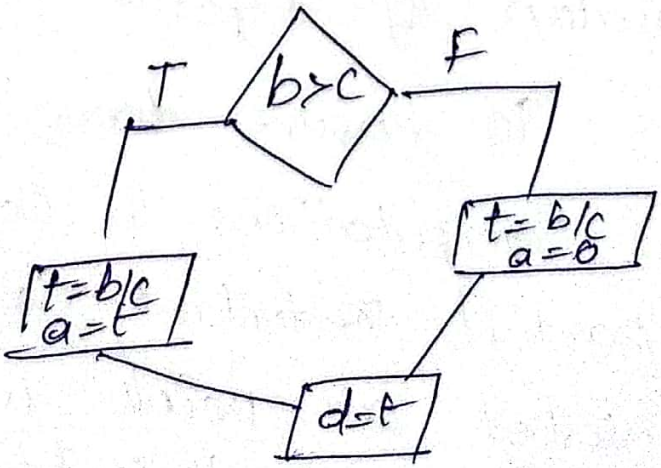
b = 20 - 30/2

c = b * (30/30 + 2) - 30

5) Partially Redundancy Elimination :-

A redundant piece of code contains expressions (or) statements that repeat themselves (or) produce the same result throughout the one execution flow of code. A partially redundant code has redundancy in one (or) more execution flows of code but not necessarily in all the paths of execution





The lazy Code Motion Algorithm:

→ All the common expressions that do not generate any duplication in code are eliminated.

→ The optimized code has not introduced any computations to the previous code.

→ The computation of expressions is done at latest possible time.

The lazy code motion is the optimization of partial redundancy to perform the computations at the latest possible time in the code. This property is important as the common expressions values are stored in registers until their last use.

Anticipation of Expressions:

To ensure there are no extra computations in code, copies of partial redundant code should be inserted at points where the expression is anticipated.

An Expression E is said to be anticipated at a point p if the values being used in E are available at that point p and all execution paths leading from p evaluate the value of E.

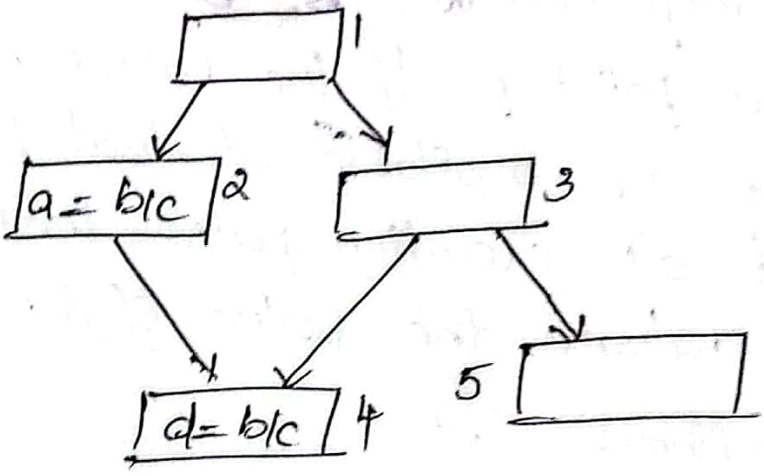
Algorithm:

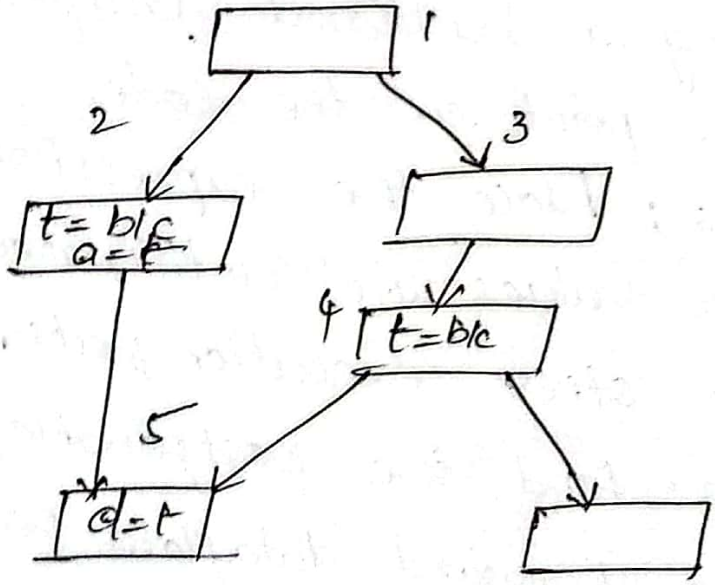
Step 1: Find all the anticipated expressions using a backward dataflow pass at each point in the code.

Step 2: Place the expressions where their values are anticipated along some other execution path.

Step 3: Find the postponable expressions using forward dataflow pass and place at a point in the code where they cannot be postponed further.

Step 4: Remove all the temporary variables assignment that are used only once in the complete code.





F) Loops in flow Graph:

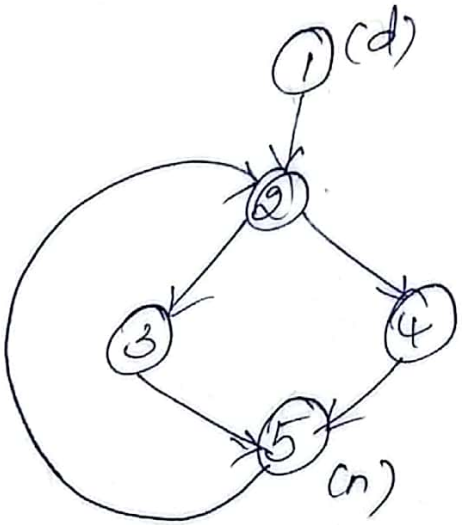
A graphical representation of three address code called flow graph, is useful for understanding code generation. Nodes in a flow graph represent computations and the edges represent the flow of control.

1. Dominator:

A node d is said to be dominate node 'n' in a flow graph if every path to node n from initial node goes through d only.

Every initial node dominates all the remaining nodes in a flow graph.

Every node also dominates by itself.



Example

Node 1 dominates Node 2, 3, 4, 5 in addition to itself

Node 2 dominate node 3, 4, 5 "

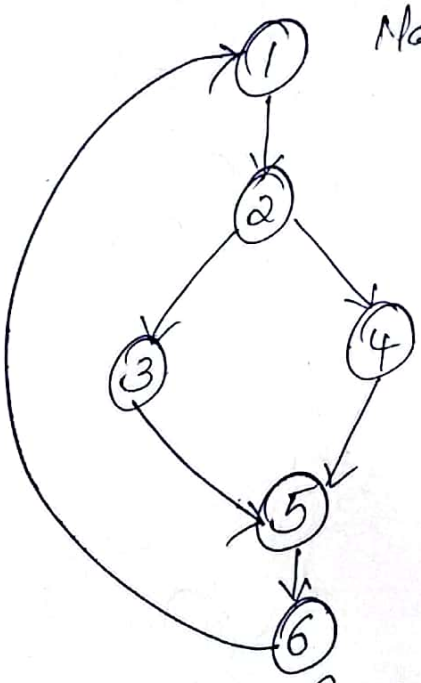
Node 3 dominate only itself.

Node 4 dominate only "

Node 5 dominates only "

2. Natural Loop:

A natural loop can be defined by a back edge n→d such that there exist a collection of all the nodes that can reach to 'n' without going through 'd'.



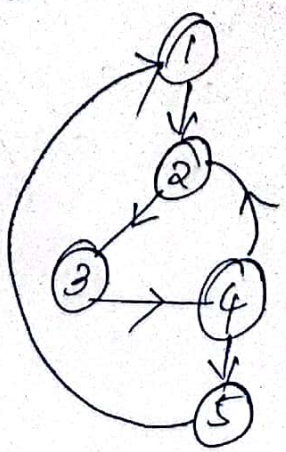
Naturaloop := d + {all nodes that can reach 'n' without going through 'd'}

Natural loop : 6→1

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

3. Inner Loop:

Innerloop is a loop that contains no other loop.



Innerloop = 4→2

= {2, 3, 4}

4. Preheader:

The preheader is a new block created such that successor of this block is header block. It is added to facilitate loop transformation computation.