Basic Blocks

- A basic block is a simple combination of statements. Except for entry and exit, the basic blocks do not have any branches like in and out.
- It means that the flow of control enters at the beginning and it always leaves at the end without any halt.
- The execution of a set of instructions of a basic block always takes place in the form of a sequence.
- The first step is to divide a group of three-address codes into the basic block.
- The new basic block always begins with the first instruction and continues to add instructions until it reaches a jump or a label. If no jumps or labels are identified, the control will flow from one instruction to the next in sequential order.

Algorithm: The algorithm used here is partitioning the three-address code into basic blocks.

Input: A sequence of three-address codes will be the input for the basic blocks.

Process: A list of basic blocks with each three address statements, in exactly one block, is considered as the output.

Method: We'll start by identifying the intermediate code's leaders. The following are some guidelines for identifying leaders:

1. The first instruction in the intermediate code is generally considered as a leader.
2. The instructions that target a conditional or unconditional jump statement can be considered as a leader.
3. Any instructions that are just after a conditional or unconditional jump statement can be considered as a leader.

- Each leader's basic block will contain all of the instructions from the leader until the instruction right before the following leader's start.

Example:

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

T1 = b + c
T2 = T1 - d
a = T2

This represents a basic block in which all the statements execute in a sequence one after the other.

Basic Block Construction:

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)

Using the algorithm given above, we can identify the number of basic blocks in the above three-address code easily-

There are two Basic Blocks in the above three-address code:

- **B1 –** Statement 1 to 4
- **B2 –** Statement 5 to 11

Transformations on Basic blocks:

Transformations on basic blocks can be applied to a basic block. While transformation, we don't need to change the set of expressions computed by the block.

There are two types of basic block transformations.

1. Structure-Preserving Transformations

Structure preserving transformations can be achieved by the following methods:

1. Common sub-expression elimination
2. Dead code elimination
3. Renaming of temporary variables
4. Interchange of two independent adjacent statements

2. Algebraic Transformations

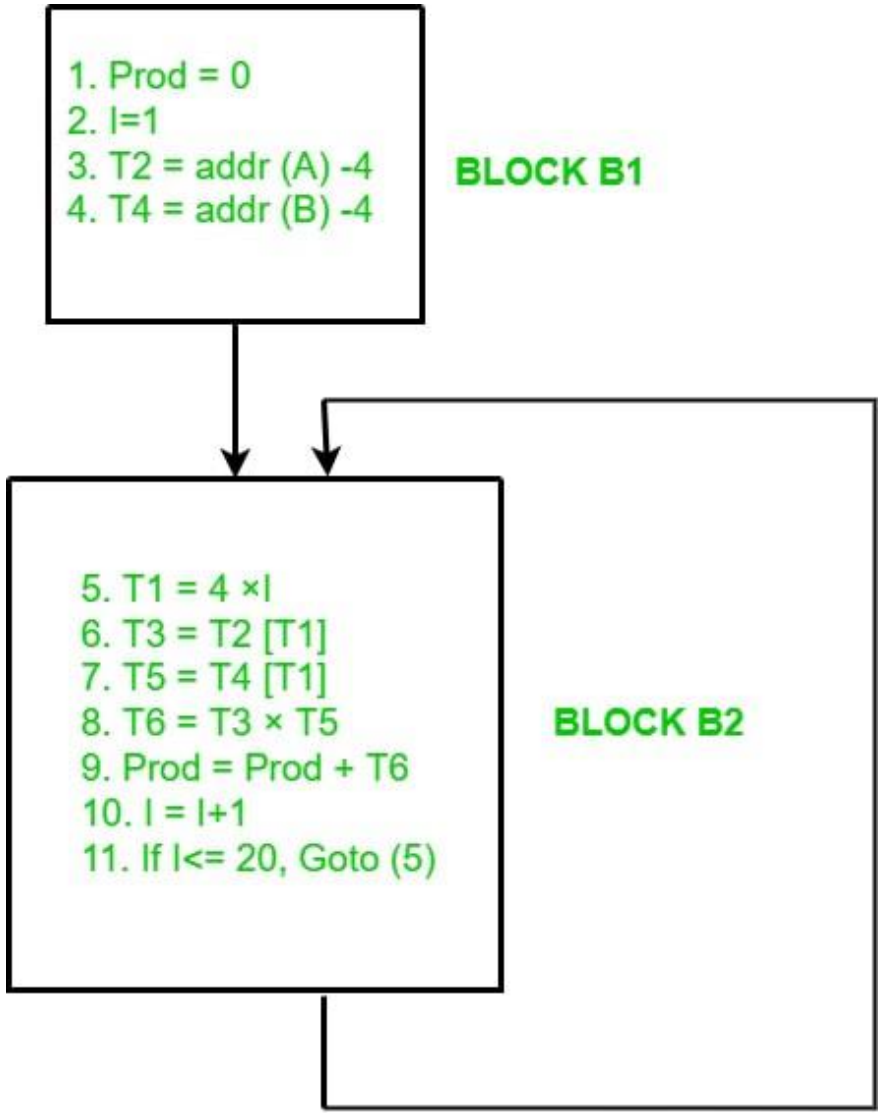
In the case of algebraic transformation, we basically change the set of expressions into an algebraically equivalent set.

For example, and expression

Flow Graph

- A flow graph is simply a directed graph. For the set of basic blocks, a flow graph shows the flow of control information.
- A control flow graph is used to depict how the program control is being parsed among the blocks.
- A flow graph is used to illustrate the flow of control between basic blocks once an intermediate code has been partitioned into basic blocks.
- When the beginning instruction of the Y block follows the last instruction of the X block, an edge might flow from one block X to another block Y.

Let’s make the flow graph of the example that we used for basic block formation:



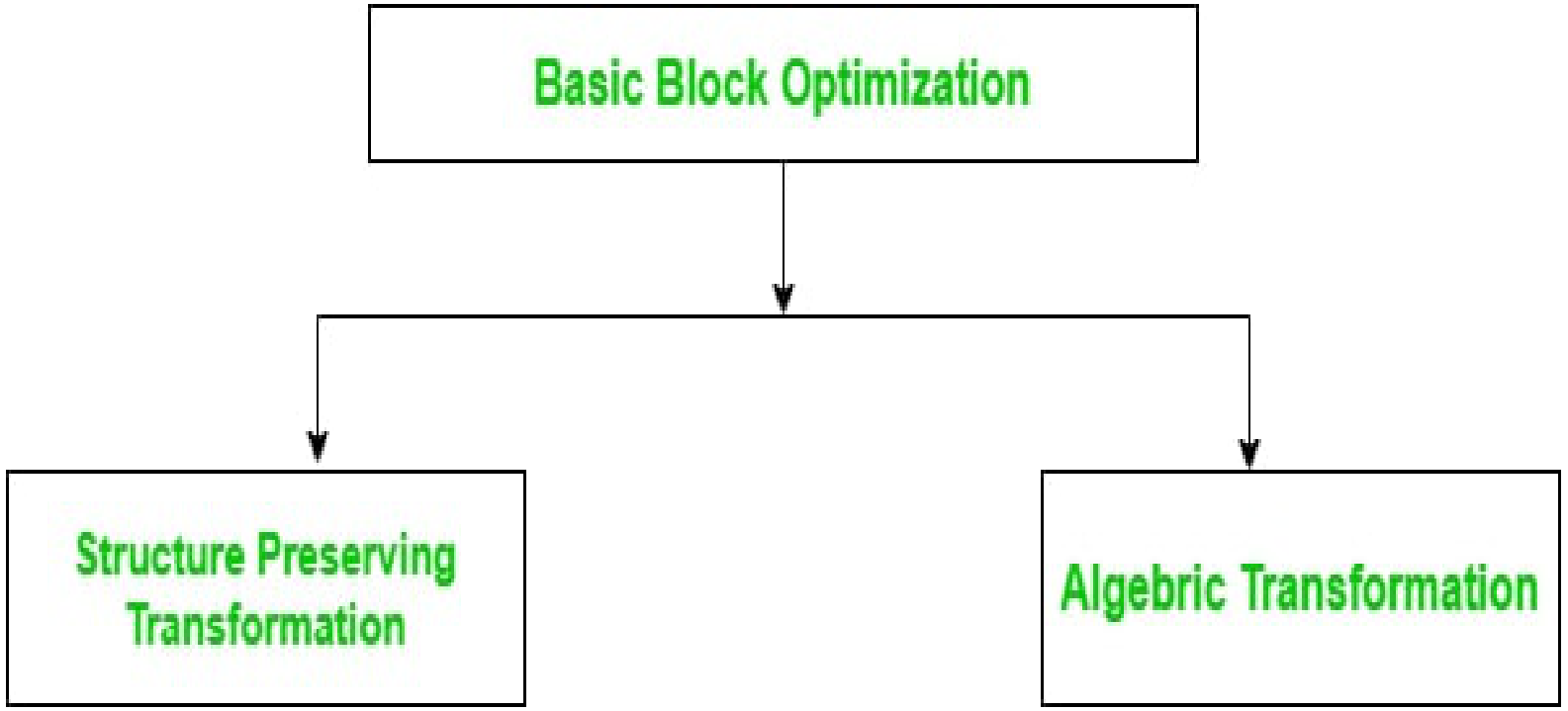
Optimization of Basic Blocks

- 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:

1.Structure preserving transformations

2.Algebraic transformations



Structure-Preserving Transformations:

The structure-preserving transformation on basic blocks includes:

1.Dead Code Elimination

2.Common Subexpression Elimination

3.Renaming of Temporary variables

4.Interchange of two independent adjacent statements

1.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.

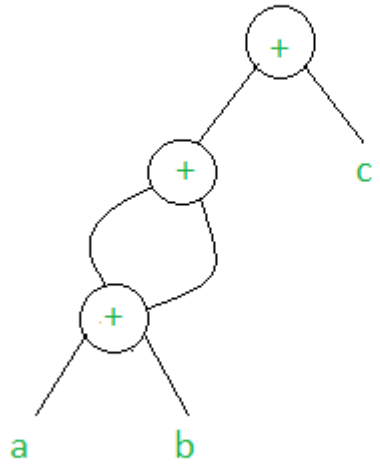
```
// Program with Dead code
int main()
{
   x = 2
   if (x > 2)
     cout << "code"; // Dead code
   else
     cout << "Optimization";
   return 0;
}
```

```
// Optimized Program without dead code
int main()
{
   x = 2;
   cout << "Optimization"; // Dead Code Eliminated
   return 0;
}
```

2.Common Subexpression Elimination:

In this technique, the sub-expression which are common are used frequently are calculated only once and reused when needed. DAG (Directed Acyclic Graph) is used to eliminate common subexpressions.

x=(a+b)+(a+b)+c



6 Nodes
6 Edges

DAG Representation

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.

```
t1 = a + b
t2 = c + d
```

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:

1.Constant Folding
2.Copy Propagation
3.Strength Reduction

1. Constant Folding:

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

Example:

$x = 2 * 3 + y \Rightarrow x = 6 + y$ (Optimized code)

2. Copy Propagation:

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

Variable Propagation:

$x = y \quad \Rightarrow z = y + 2$ (Optimized code)

$z = x + 2$

Constant Propagation:

$x = 3 \quad \Rightarrow z = 3 + a$ (Optimized code)

$z = x + a$

3. Strength Reduction:

Replace expensive statement/ instruction with cheaper ones.

x = 2 * y (costly) $\Rightarrow$ x = y + y (cheaper)

x = 2 * y (costly) $\Rightarrow$ x = y << 1 (cheaper)

Loop Optimization

Loop optimization includes the following strategies:

1. Code motion & Frequency Reduction
2. Induction variable elimination
3. Loop merging/combining
4. Loop Unrolling

1. Code Motion & Frequency Reduction

Move loop invariant code outside of the loop.

```
// Program with loop variant inside loop
int main()
{
   for (i = 0; i < n; i++) {
      x = 10;
      y = y + i;
   }
   return 0;
}
// Program with loop variant outside loop
int main()
{
   x = 10;
   for (i = 0; i < n; i++)
      y = y + i;
   return 0;
}
```

2. Induction Variable Elimination:

Eliminate various unnecessary induction variables used in the loop.

```
// Program with multiple induction variables
int main()
{
    i1 = 0;
    i2 = 0;
    for (i = 0; i < n; i++) {
        A[i1++] = B[i2++];
    }
    return 0;
}
// Program with one induction variable
int main()
{
    for (i = 0; i < n; i++) {
        A[i] = B[i]; // Only one induction variable
    }
    return 0;
}
```

3. Loop Merging/Combining:

If the operations performed can be done in a single loop then, merge or combine the loops.

```
// Program with multiple loops
int main()
{
   for (i = 0; i < n; i++)
      A[i] = i + 1;
   for (j = 0; j < n; j++)
      B[j] = j - 1;
   return 0;
}
// Program with one loop when multiple loops are merged
int main()
{
   for (i = 0; i < n; i++) {
      A[i] = i + 1;
      B[i] = i - 1;
   }
   return 0;
}
```

4. Loop Unrolling:

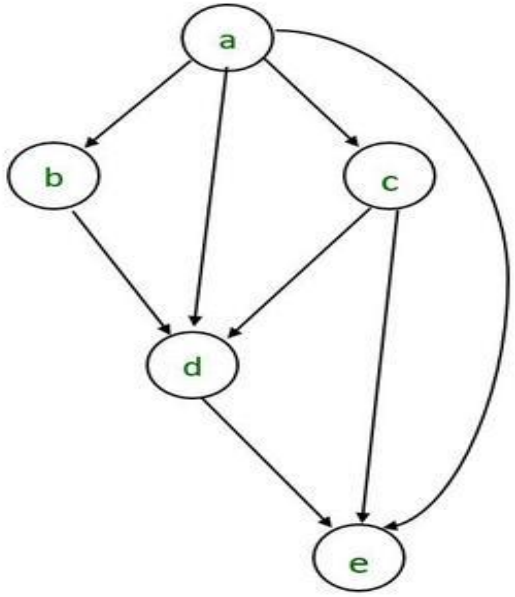
If there exists simple code which can reduce the number of times the loop executes then, the loop can be replaced with these codes.

```
// Program with loops
int main()
{
    for (i = 0; i < 3; i++)
        cout << "Cd";
    return 0;
}
// Program with simple code without loops
int main()
{
    cout << "Cd";
    cout << "Cd";
    cout << "Cd";
    return 0;
}
```

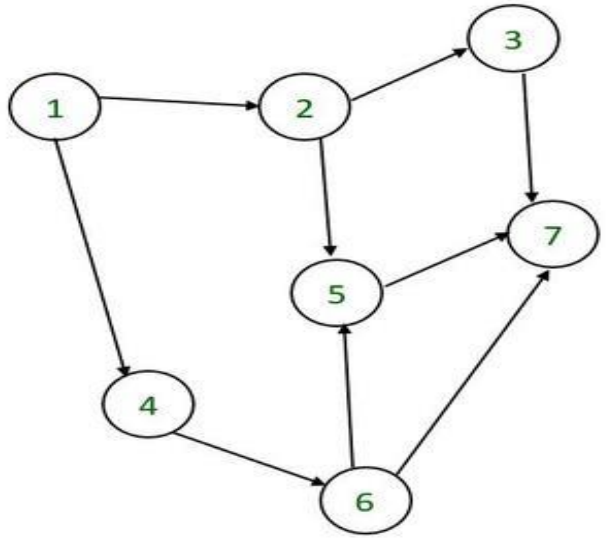
DAG representation of Basic Blocks

- A DAG is a graph that contains directed edges but no cycles, ensuring that no path leads back to the starting node.
- DAGs are particularly useful in eliminating redundant computations and detecting common sub-expressions, making the execution of programs more efficient.
- This graphical representation helps in optimizing the Intermediate Code Generation phase of a compiler.

- The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.
- 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)



(B)

Algorithm for Construction of Directed Acyclic Graph:

There are three possible scenarios for building a DAG on three address codes:

Case 1 – x = y op z
Case 2 – x = op y
Case 3 – x = y

Directed Acyclic Graph for the above cases can be built as follows :

Step 1:

If the y operand is not defined, then create a node (y).
If the z operand is not defined, create a node for case(1) as node(z).

Step 2:

Create node(OP) for case(1), with node(z) as its right child and node(OP) as its left child (y).
For the case (2), see if there is a node operator (OP) with one child node (y).
Node n will be node(y) in case (3).

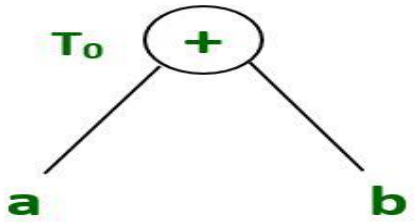
Step 3:

Remove x from the list of node identifiers. Step 2: Add x to the list of attached identifiers for node n.

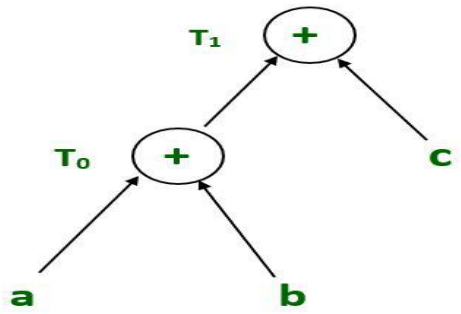
Example 1:

T0 = a + b —Expression 1
T1 = T0 + c —-Expression 2
d = T0 + T1 ——Expression 3

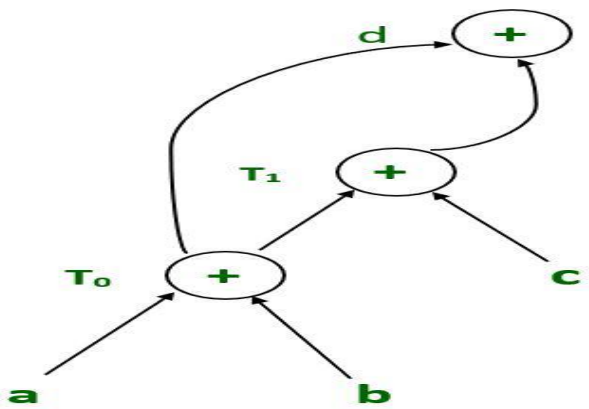
Expression 1 : T0 = a + b



Expression 2: T1 = T0 + c



Expression 3 : d = T0 + T1



Simple Code Generator

- “Simple Code Generator” (SCG), which implements several functions that make it easy to create simple code generators for any programming language.
- The SCG component consists of two parts: firstly it contains a parser that transforms textual inputs into an abstract syntax tree; secondly, its generated AST has expressions in a symbolic form wherever possible instead of merely representing them as strings like most other compilers do

- **A code generator is a compiler that translates the intermediate representation of the source program into the target program.**
- In other words, a code generator translates an abstract syntax tree into machine-dependent executable code.
- The process of generating machine-dependent output from an abstract syntax tree involves two steps: one for constructing the abstract syntax tree and another for generating its corresponding machine code.

- The first step involves constructing an Abstract Syntax Tree (AST) by traversing all possible paths through your input file(s).
- This tree will contain information about every bit of data in your program as they are encountered during parsing or execution time; it's important to note that this can take place both at compile time (as part of compiling) or runtime (in some cases).

Register Descriptor

- Register descriptors are data structures that store information about the registers used in the program.
- This includes the registration number and its name, along with its type.
- The compiler uses this information when generating machine code for your program, so it’s important to keep it up-to-date while writing code!
- The compiler uses the register file to determine what values will be available for use in your program.
- This is done by walking through each of the registers and determining if they contain valid data or not. If there’s nothing in a register, then it can be used for other purposes!

Address Descriptor

- An address descriptor is used to represent the memory locations used by a program. Address descriptors are created by the **getReg function**, which returns a structure containing information about how to access memory.
- Address descriptors can be created for any instruction in your program's code and stored in registers or on the stack; however, only one instance of an address descriptor will exist at any given time (unless another thread is executing).
- When the user wants to retrieve data from an arbitrary location within the program's source code using getReg, call this method with two arguments:
- The first argument specifies which register contains your desired value (e.g., 'M'),
- While the second argument specifies where exactly within this register should it be placed back onto its original storage location on disk/memory before returning it back up into main memory again after successfully accessing its contents via indirect calls like LoadFromBuffer() or StoreToBuffer().

Register Allocations in Code Generation

- Registers are the fastest locations in the memory hierarchy. But unfortunately, this resource is limited.
- It comes under the most constrained resources of the target processor. Register allocation is an NP-complete problem.
- However, this problem can be reduced to graph coloring to achieve allocation and assignment.
- Therefore a good register allocator computes an effective approximate solution to a hard problem.

- The register allocator determines which values will reside in the register and which register will hold each of those values.
- It takes as its input a program with an arbitrary number of registers and produces a program with a finite register set that can fit into the target machine.

Allocation vs Assignment:

Allocation –

Maps an unlimited namespace onto that register set of the target machine.

•**Reg. to Reg. Model:** Maps virtual registers to physical registers but spills excess amount to memory.

•**Mem. to Mem. Model:** Maps some subset of the memory location to a set of names that models the physical register set.

Allocation ensures that code will fit the target machine's reg. set at each instruction.

Assignment –

Maps an allocated name set to the physical register set of the target machine.

•Assumes allocation has been done so that code will fit into the set of physical registers.

•No more than **'k'** values are designated into the registers, where 'k' is the no. of physical registers.

Local Register Allocation And Assignment:

Allocation just inside a basic block is called Local Reg. Allocation. Two approaches for local reg. allocation: Top-down approach and bottom-up approach.

Top-Down Approach is a simple approach based on ‘Frequency Count’. Identify the values which should be kept in registers and which should be kept in memory.

Algorithm:

1.Compute a priority for each virtual register.
2.Sort the registers into priority order.
3.Assign registers in priority order.
4.Rewrite the code.

Moving beyond single Blocks:

•More complicated because the control flow enters the picture.
•Liveness and Live Ranges: Live ranges consist of a set of definitions and uses that are related to each other as they i.e. no single register can be common in a such couple of instruction/data.

Following is a way to find out Live ranges in a block. A live range is represented as an interval [i,j], where i is the definition and j is the last use.

Global Register Allocation and Assignment:

1. The main issue of a register allocator is minimizing the impact of spill code;
•Execution time for spill code.
•Code space for spill operation.
•Data space for spilled values.
2. Global allocation can't guarantee an optimal solution for the execution time of spill code.
3. Prime differences between Local and Global Allocation:
•The structure of a global live range is naturally more complex than the local one.
•Within a global live range, distinct references may execute a different number of times. (When basic blocks form a loop)
4. To make the decision about allocation and assignments, the global allocator mostly uses graph coloring by building an interference graph.
5. Register allocator then attempts to construct a k-coloring for that graph where 'k' is the no. of physical registers.
•In case, the compiler can't directly construct a k-coloring for that graph, it modifies the underlying code by spilling some values to memory and tries again.
•Spilling actually simplifies that graph which ensures that the algorithm will halt.
6. Global Allocator uses several approaches, however, we'll see top-down and bottom-up allocations strategies. Subproblems associated with the above approaches.
•Discovering Global live ranges.
•Estimating Spilling Costs.
•Building an Interference graph.

Peephole Optimization in Compiler Design

- Peephole optimization is a type of code Optimization performed on a small part of the code. It is performed on a very small set of instructions in a segment of code.

- The small set of instructions or small part of code on which peephole optimization is performed is known as peephole or window.

- It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without a change in output. The peephole is machine-dependent optimization.

Objectives of Peephole Optimization:

The objective of peephole optimization is as follows:

- To improve performance
- To reduce memory footprint
- To reduce code size

Peephole Optimization Techniques:

A. Redundant load and store elimination: In this technique, redundancy is eliminated.

Initial code:

```
y = x + 5;
i = y;
z = i;
w = z * 3;
```

Optimized code:

```
y = x + 5;
w = y * 3; //* there is no i now
//* We've removed two redundant variables i & z whose value were just being copied from one another.
```

B. Constant folding: The code that can be simplified by the user itself, is simplified. Here simplification to be done at runtime are replaced with simplified code to avoid additional computation.

Initial code:

```
x = 2 * 3;
```

Optimized code:

```
x = 6;
```

C. Strength Reduction: The operators that consume higher execution time are replaced by the operators consuming less execution time.

Initial code:

```
y = x * 2;
```

Optimized code:

```
y = x + x;    or    y = x << 1;
```

Initial code:

```
y = x / 2;
```

Optimized code:

```
y = x >> 1;
```

D. Null sequences/ Simplify Algebraic Expressions : Useless operations are deleted.

```
a := a + 0;
a := a * 1;
a := a/1;
a := a - 0;
```

E. Combine operations: Several operations are replaced by a single equivalent operation.

F. Deadcode Elimination: Dead code refers to portions of the program that are never executed or do not affect the program's observable behavior. Eliminating dead code helps improve the efficiency and performance of the compiled program by reducing unnecessary computations and memory usage.

Initial Code:-

```
int Dead(void)
{
   int a=10;
   int z=50;
   int c;
   c=z*5;
   printf(c);
   a=20;
   a=a*10; //No need of These Two Lines
   return 0;
   }
```

Optimized Code:-

```
int Dead(void)
{
   int a=10;
   int z=50;
   int c;
   c=z*5;
   printf(c);
   return 0;
   }
```