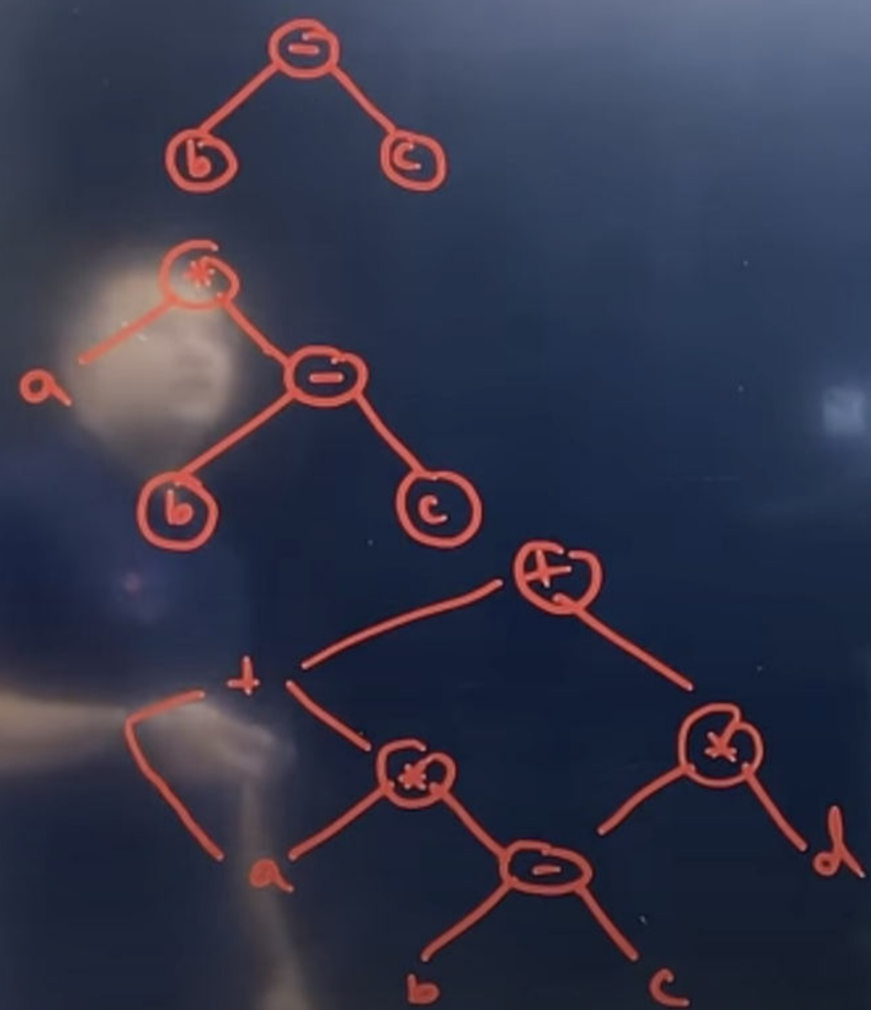
Complete CD Compiler Design in one shot | Semester Exam | Hindi

$a + a * (b-c) + (b-c) * d$



Chapter-5 (CODE GENRATION)

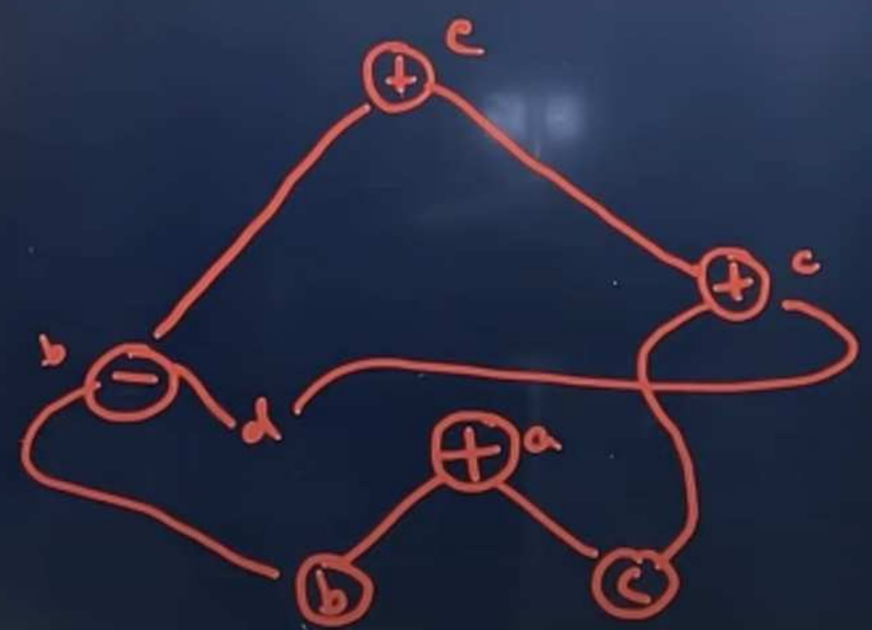
Direct Acyclic Graph (DAG)

$a = b + c$

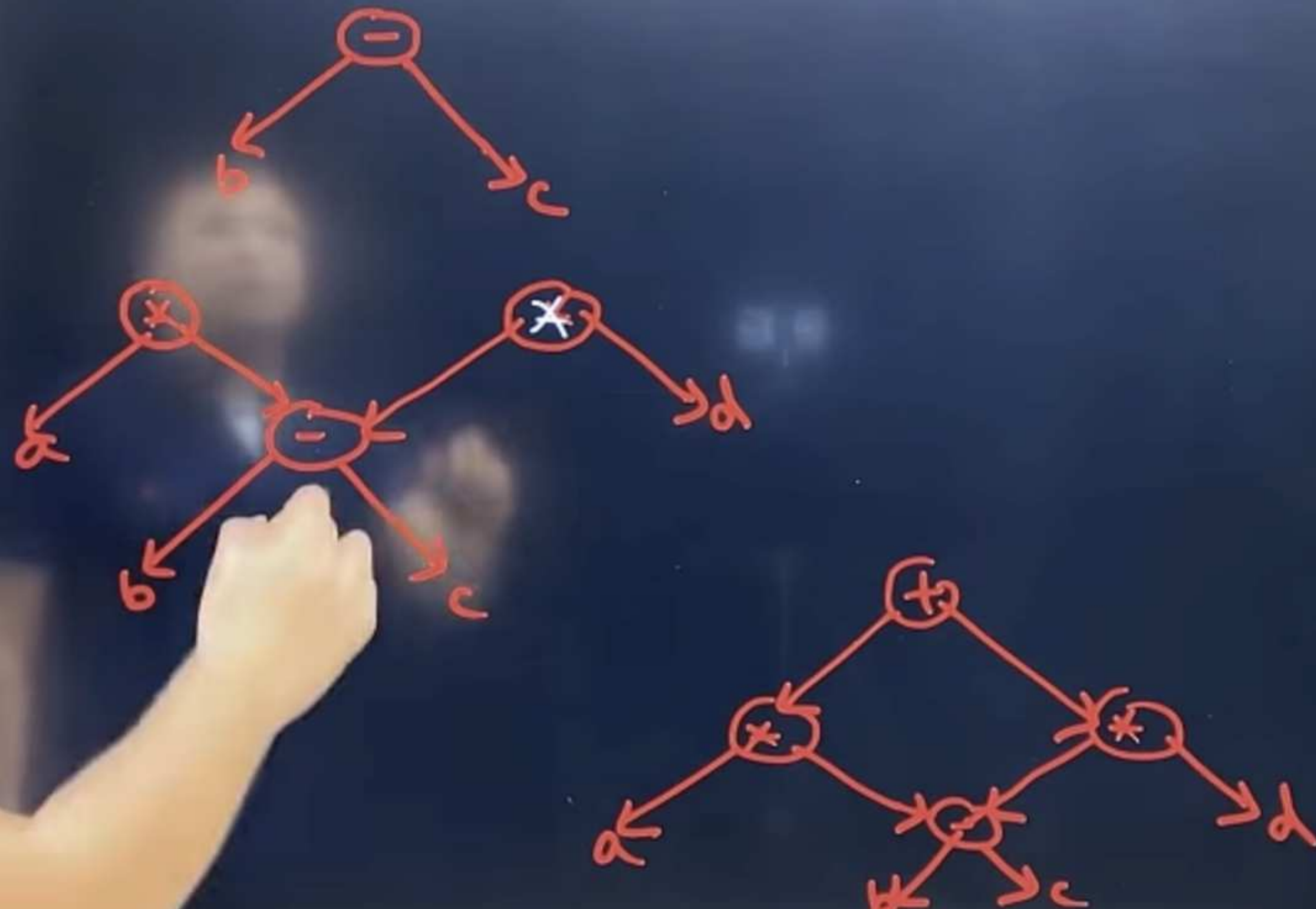
$b = b - d$

$c = c + d$

$e = b + c$



$a * (b-c) + (b-c) * d$

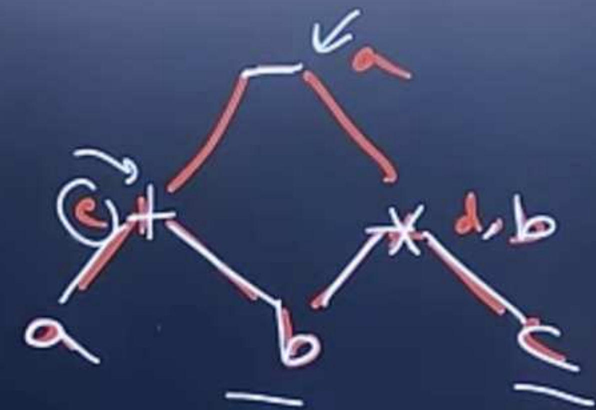


$a = b * c$

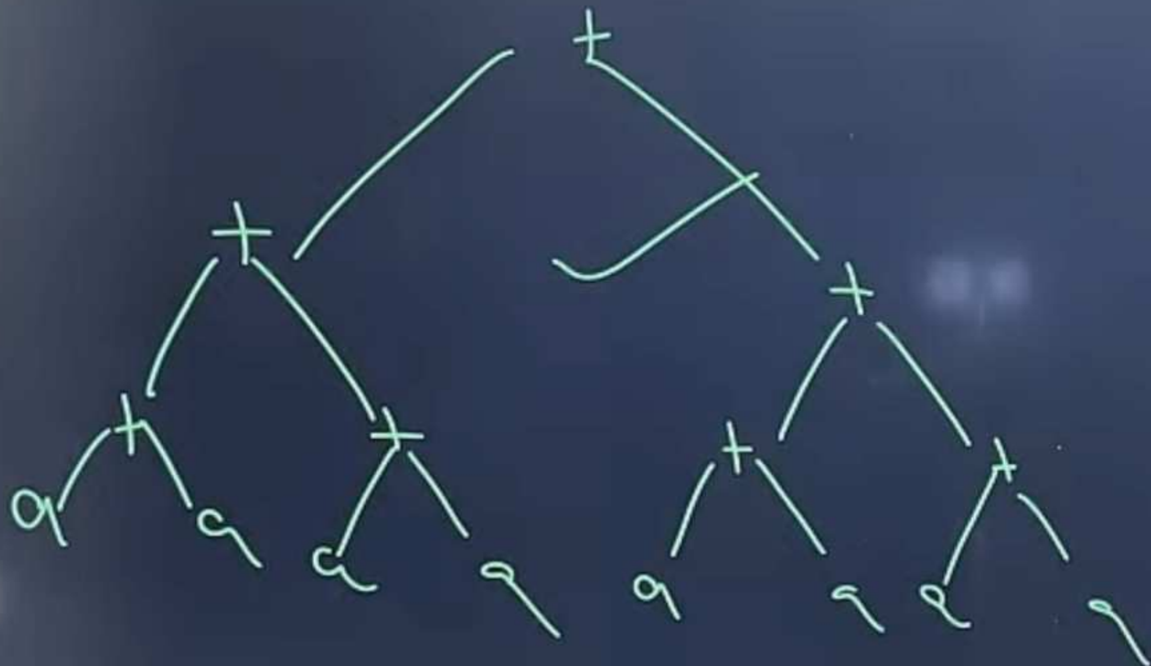
$c = a + b$

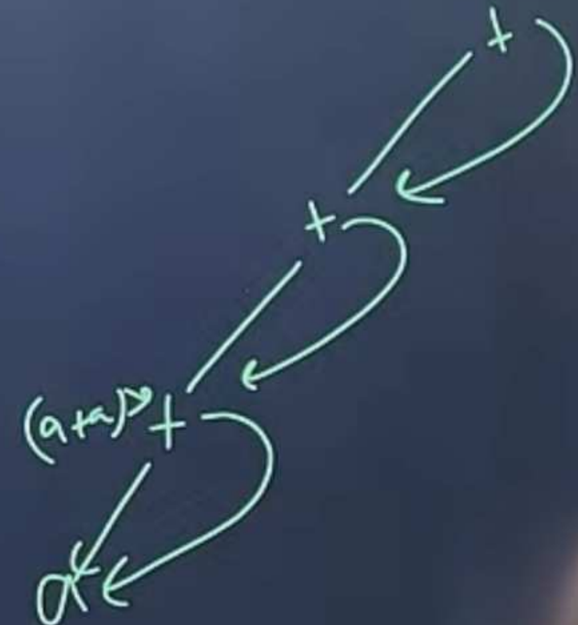
$b = b * c$

$a = c - d$



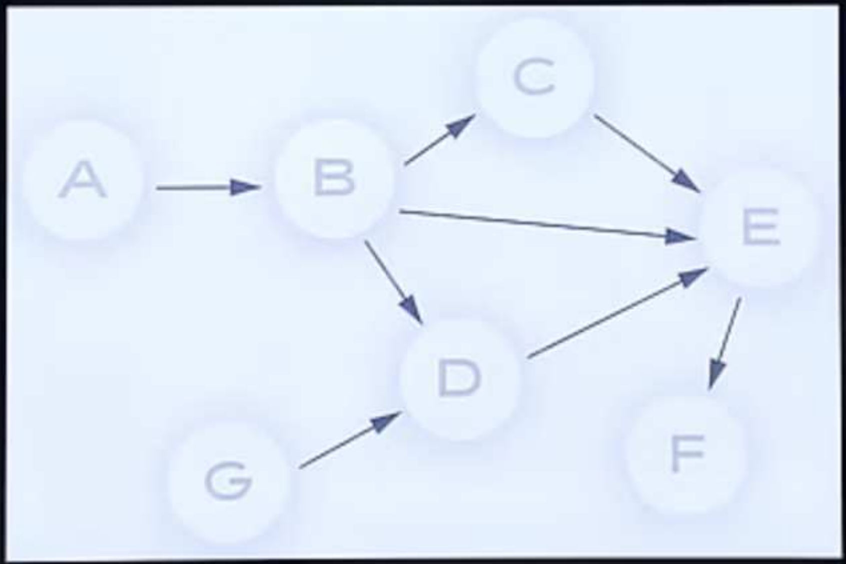
$$((a+a) + (a+a)) + ((a+a) + (a+a))$$





Direct Acyclic Graph (DAG)

- A Direct Acyclic Graph is a graph that is directed and contains no cycles. So it is impossible to start at any vertex v and follow a sequence of edges that eventually loops back to v again.
- By representing expressions and operations in a DAG, compilers can easily identify and eliminate redundant calculations, thus optimizing the code.
  - We automatically detect common sub-expressions with the help of DAG algorithm.
  - We can determine which identifiers have their values used in the block.
  - We can determine which statements compute values which could be used outside the block.



Algebraic Simplification: Basic laws of math's which can be solved directly.

a = b*1

a=b

a=b+0

a=b

$a * 1 = a$

$a + 0 =$

Redundant code Elimination / Common subexpression elimination:

Avoiding the evaluation of any expression more than once is redundant code elimination.

$x = a + b$

$y = b + a$

a+b = b+a

$x = a + b$

$y = x$

Strength reduction: replacing the costly operator by cheaper operator, this process is called strength reduction.

$y = 2 * x$

$y = x + x$

+ 1

− 2

÷ 6

× 3

Constant Propagation: replacing the value of constant before compile time, is called as constant propagation.

pi = 3.1415

x = 360 / pi

x = 360/3.1415

Constant Folding: Replacing the value of expression before compilation is called as constant folding

$x = a + b + \underbrace{2 * 3 + 4}$

$x = a + b + 10$

Optimization of basic blocks

- Algebraic Simplification
- Redundant code Elimination / Common subexpression elimination
- Strength reduction
- Constant Propagation
- Constant Folding

Code movement(Loop invariant computation): removing those code out from the loop which is not related to loop.

```
int i=1;
while(i<=100)
{
    a =b+c
    print(i)
    i++
}
```

b=1
c=2
a=3

Loop Unrolling: getting the same output with less no of iteration is called loop unrolling

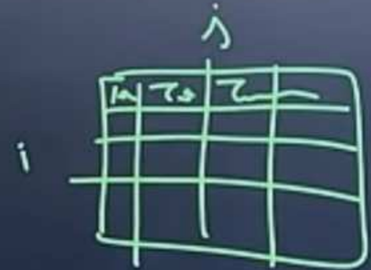
```
int i=1;
while(i<=100)
{
    print(i)
    i++
}
```

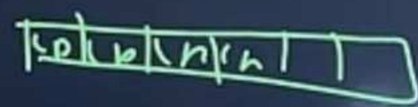
1 2 3

```
int i=1;
while(i<=100)
{
    print(i)
    i++
    print(i)
    i++
}
```

Loop Jamming: combining the bodies of two loops, whenever they share the same index and same no of variables

```
for (int i=0; i<=10; i++)
    for (int j=0; j<=10; j++)
        x[i, j]="TOC"
for (int j=0; j<=10; j++)
        y[i]="CD"
```





```
for (int i=0; i<=10; i++)
{
    for (int j=0; j<=10; j++)
    {
        x[i, j]="TOC"
    }
    y[i]="CD"
}
```

Basic block & flow graph

1) f=1; ← L
2) i=2

B1

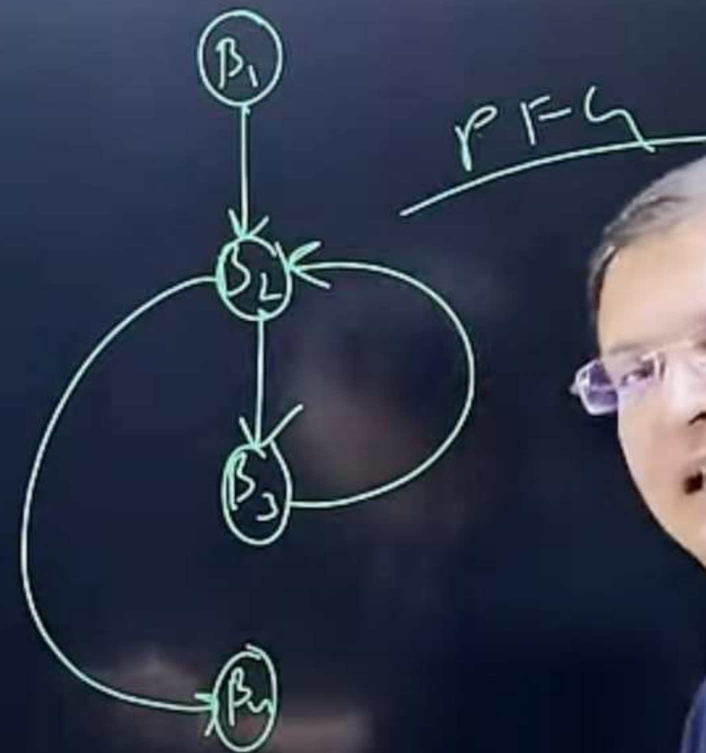
3) if(i>x), goto 9 ← L — B2

4) t1=f*i; ← L
5) f=t1;
6) t2=i+1;
7) i=t2;
8) goto(3)

B3

9) goto calling program ← L

B4



Basic block & flow graph

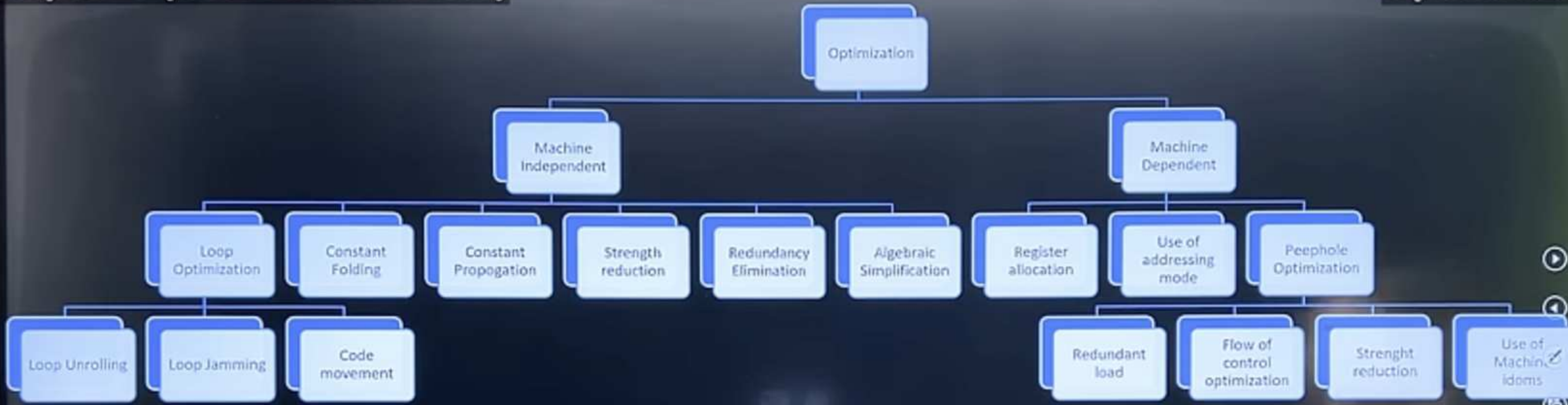
- The block can be identified with the help of leader
  - Finding the leader
  - Finding the bocks
  - Construct PFG

- In order to find the basic blocks, we need to finds the leader in the program then a basic block will start from one leader to the next leader but not including next leader.
- identifying leaders in a basic block
  - First statement is a leader
  - Statement that is the target of conditional or unconditional statement is a leader
  - Statement that follow immediately a conditional or unconditional statement is a leader

Algorithm to partition a sequence of three address statements into basic blocks

- Loop Optimization
  - To apply loop optimization, we must first detect loops.
  - For detecting loops, we use control flow analysis (CFA) using program flow graph (PFG)
  - To find PFG, we need to find basic blocks.
  - A Basic block is a sequence of 3-adress statemen here control enters at the beginning and leaves only at the end without a umps or halts

Aspect	Machine-Independent Optimization	Machine-Dependent Optimization
Definition	Optimizations that are not specific to any processor or machine architecture.	Optimizations tailored to the specifics of a particular machine or processor architecture.
Focus	Concentrates on improving the logic and efficiency of the code at a high level, often in the intermediate code.	Focuses on enhancing performance by taking advantage of the unique features and instructions of the target machine.
Examples	Common subexpression elimination, Code motion, Dead code elimination, Loop unrolling, Loop fusion	Instruction scheduling, Register allocation, Pipeline optimization, Use of machine-specific instructions, Cache optimization
Portability	Highly portable across different architectures as they do not rely on machine-specific features.	Not portable; optimizations must be re-applied or altered for different architectures.
Stage of Application	Applied before the code is mapped to the target machine's instruction set, often during the intermediate code generation phase.	Applied during or after the generation of the target machine code, tailoring the optimizations to the specifics of the machine's hardware.


Optimization
Machine Independent
Machine Dependent
Loop Optimization
Constant Folding
Constant Propogation
Strength reduction
Redundancy Elimination
Algebraic Simplification
Register allocation
Use of addressing mode
Peephole Optimization
Loop Unrolling
Loop Jamming
Code movement
Redundant load
Flow of control optimization
Strenght reduction
Use of Machine idoms

- **Optimization**: During code generation, various optimization techniques are applied to improve efficiency and performance. This could include optimizing for speed, memory usage, or even power consumption.

- **Target Platform**: The generated code is specific to the target platform's architecture, such as x86, ARM, etc. This means the same high-level code will have different generated machine code for different platforms.

Design Issues

- **Code Generator Input**:
  - Uses the source program's intermediate representation (IR) and symbol table data.
- **Intermediate Representation (IR)**:
  - Consists of three-address and graphical forms.
- **Target Program**:
  - Influenced by the target machine's instruction set architecture (ISA).
  - Common ISAs: RISC, CISC, and stack-based.
- **Instruction Selection**:
  - Converts IR to executable code for the target machine.
  - High-level IR may use code templates for translation.
- **Register Allocation**:
  - Critical to decide which values to store in registers.
  - Non-registered values stay in memory.
  - Register use leads to shorter, faster instructions.
  - Involves two steps: i. Choosing variables for register storage. ii. Assigning specific registe these variables.
- **Evaluation Order**:
  - The sequence of computations impacts code efficiency.
  - Some sequences minimize register usage for intermediate results.

Code Generation

- Code generation is the process of converting the intermediate representation (IR) of source code into a target code(assembly level). Which is also optimized.
- It involves translating the syntax and semantics of the high-level language into assembly level code, typically after the source code has passed through lexical analysis, syntax analysis, semantic analysis, and intermediate code generation.

int main() {	t1 = 5	MOV R1, 5
int a = 5;	t2 = 3	MOV R2, 3
int b = 3;	t3 = t1 + t2	ADD R3, R1, R2;
int sum = a + b;		
}		