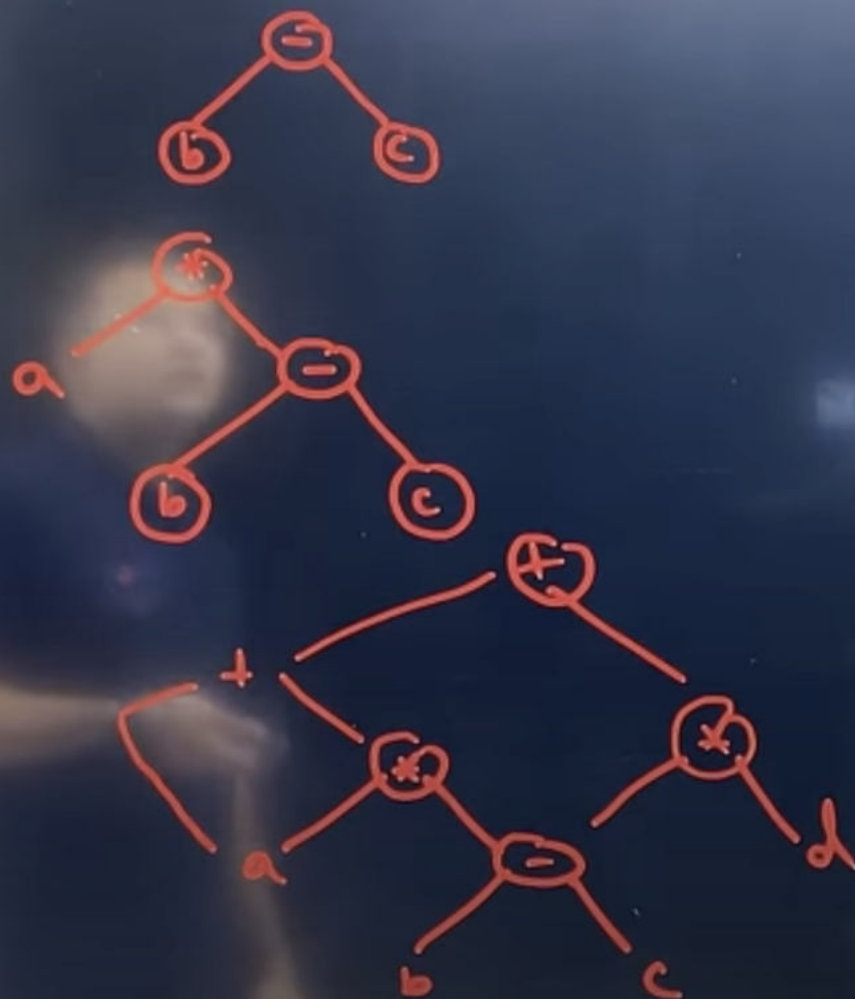


$$a + a * (b - c) + (b - c) * d$$



KG



$$a = b + c$$

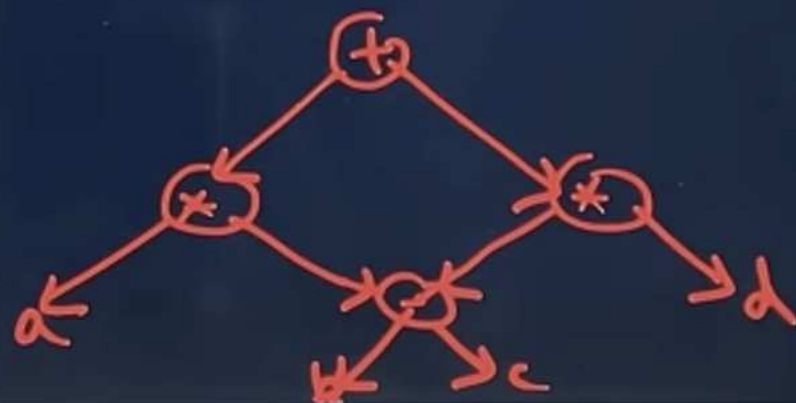
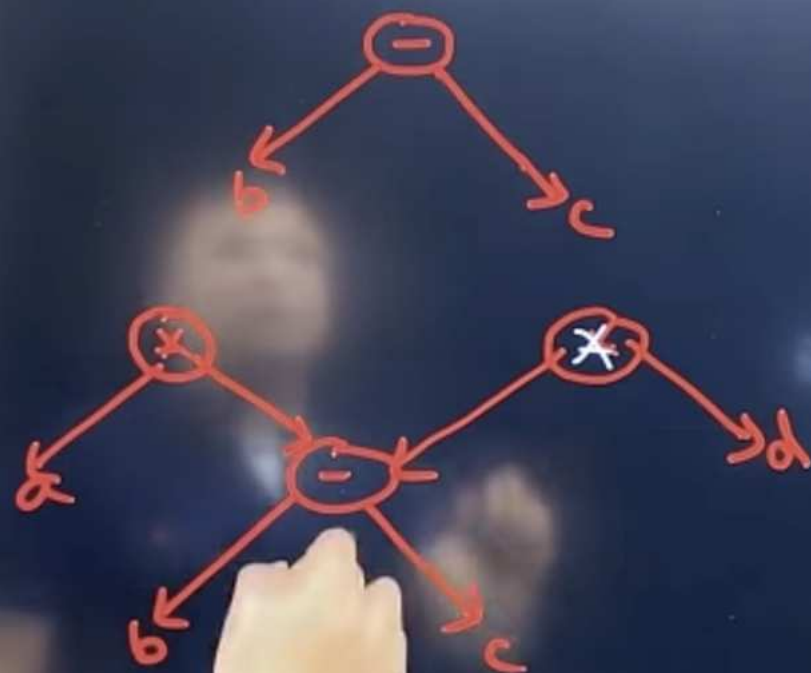
$$b = b - d$$

$$c = c + d$$

$$e = b + c$$



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

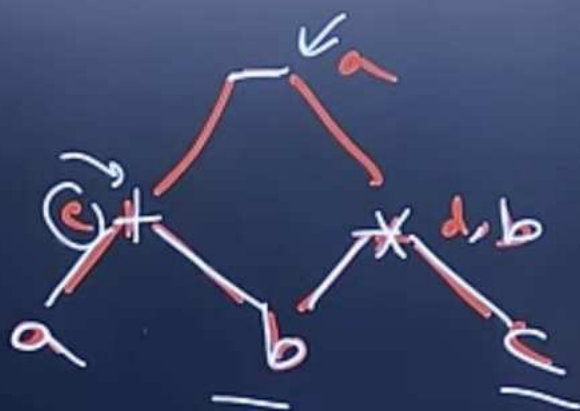


$$d = b * c$$

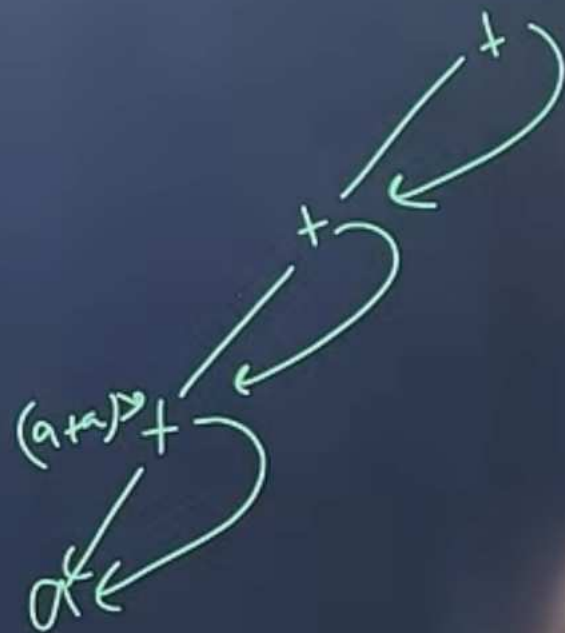
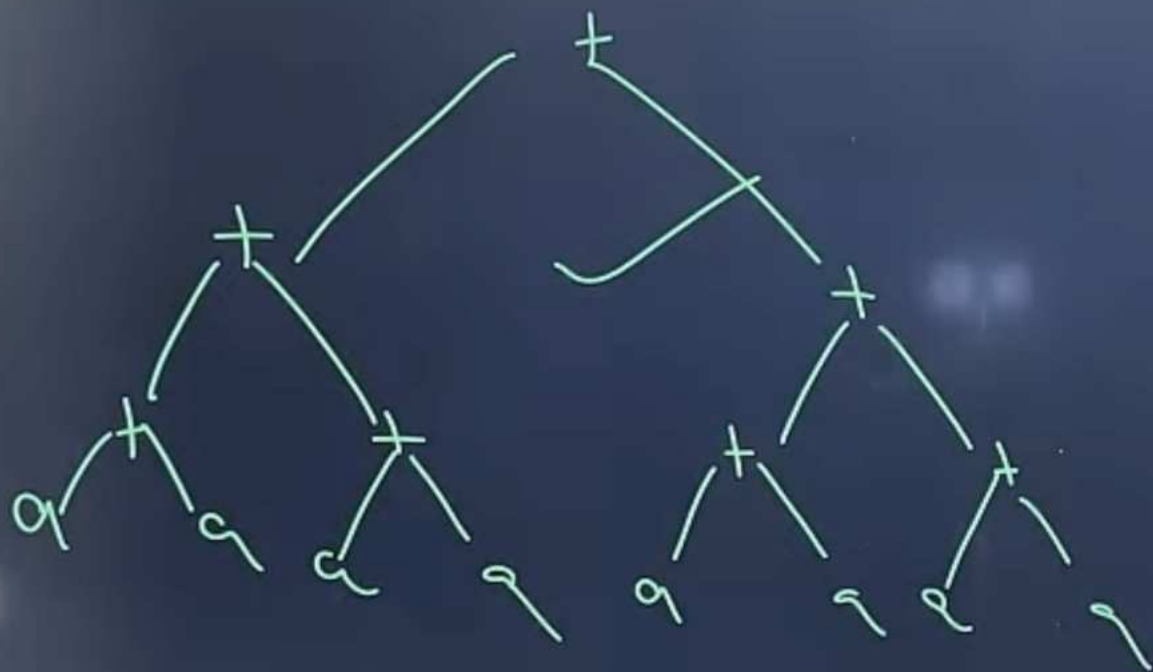
$$c = a + b$$

$$b = b * c$$

$$a = c - d$$

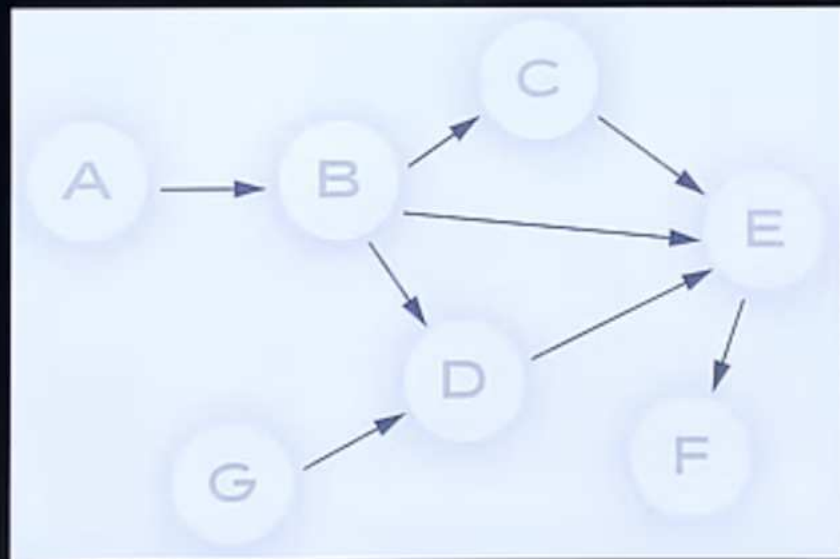


$((a+a) + (a+a)) + ((a+a) + (a+a))$



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.

$$a = b * 1$$

$$a = b$$

$$a = b + 0$$

$$a = b$$

$$a \times 1 = a$$

$$a + 0 = a$$

Redundant code Elimination / Common subexpression elimination:

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

$x = a + b$ ✓

$y = b + a$

$x = a + b$ ✓

$y = x$ ✓

$a + b = b + a$



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

$$\begin{array}{l} y = 2 * x \\ \downarrow \\ y = x + x \end{array}$$

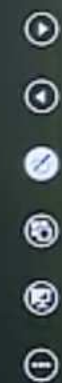
+	✓	1
-		2
÷		6
x		3

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

$\pi = 3.1415$ ✓

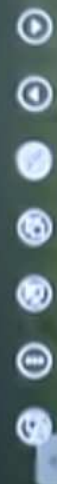
$x = 360 / \pi$

$x = 360/3.1415$



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

$x = a + b + 2 * 3 + 4$
 $x = a + b + 10$



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

Chapter-5 (CODE GENERATION)

Loop optimization

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

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

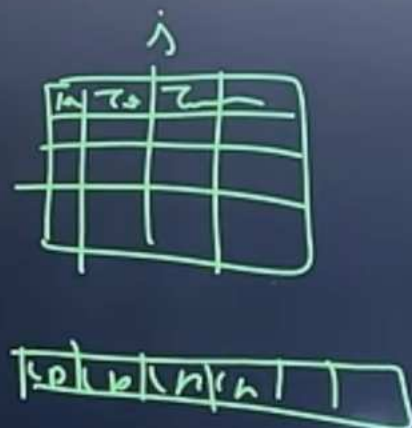
1 2 3 →

```
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"  
}
```

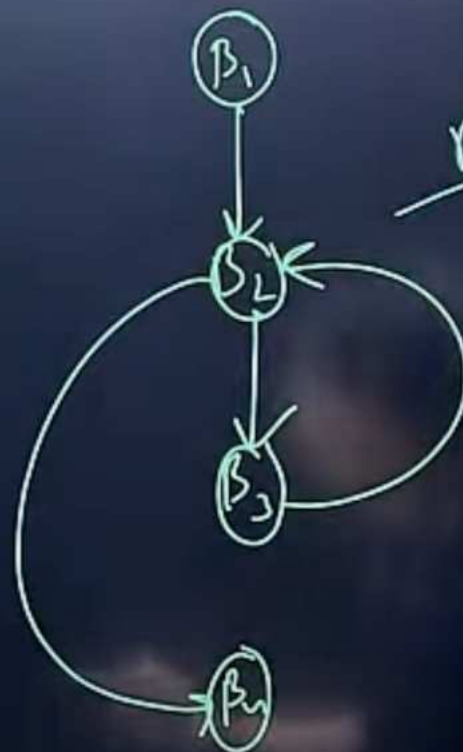

1) $f=1;$ $\leftarrow L$
2) $i=2$

3) $\text{if}(i>x), \text{goto } 9$ $\leftarrow L$

4) $t1=f*i;$ $\leftarrow L$
5) $f=t1;$
6) $t2=i+1;$
7) $i=t2;$

8) $\text{goto}(3)$

9) $\text{goto calling program}$ $\leftarrow L$



$P F = L$

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



- In order to find the basic blocks, we need to find 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



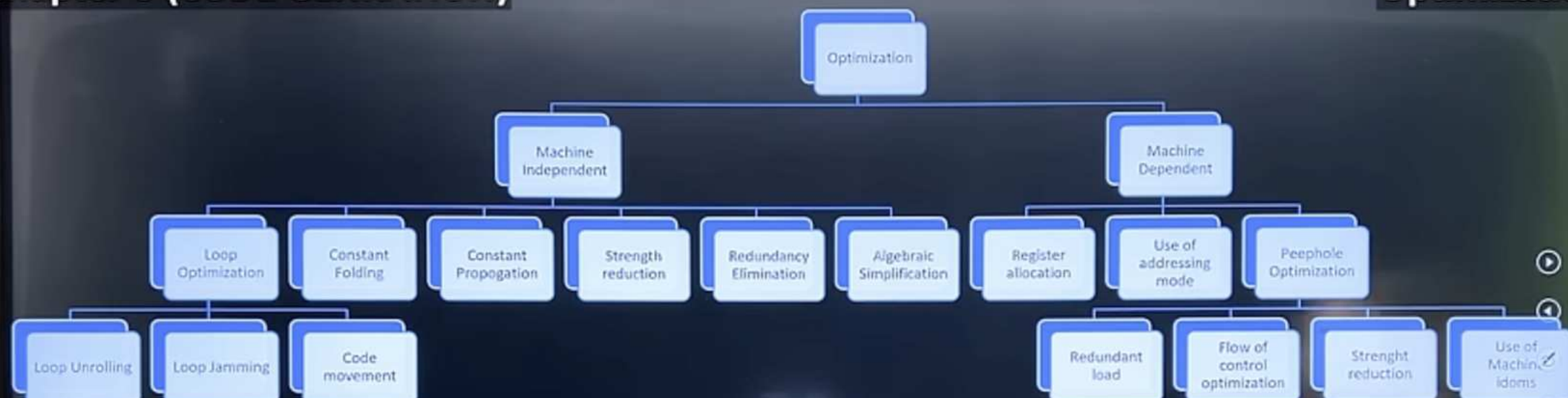
Algorithm to partition a sequence of three address statements into basic blocks

- 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-address statements where control enters at the beginning and leaves only at the end without a jumps or halts



Block	Statements	Control Flow
Block 1	Statement 1	Block 2
Block 2	Statement 2	Block 3
Block 3	Statement 3	Block 4
Block 4	Statement 4	Block 5
Block 5	Statement 5	Block 6
Block 6	Statement 6	Block 7
Block 7	Statement 7	Block 8
Block 8	Statement 8	Block 9
Block 9	Statement 9	Block 10
Block 10	Statement 10	Block 11
Block 11	Statement 11	Block 12
Block 12	Statement 12	Block 13
Block 13	Statement 13	Block 14
Block 14	Statement 14	Block 15
Block 15	Statement 15	Block 16
Block 16	Statement 16	Block 17
Block 17	Statement 17	Block 18
Block 18	Statement 18	Block 19
Block 19	Statement 19	Block 20
Block 20	Statement 20	Block 21
Block 21	Statement 21	Block 22
Block 22	Statement 22	Block 23
Block 23	Statement 23	Block 24
Block 24	Statement 24	Block 25
Block 25	Statement 25	Block 26
Block 26	Statement 26	Block 27
Block 27	Statement 27	Block 28
Block 28	Statement 28	Block 29
Block 29	Statement 29	Block 30
Block 30	Statement 30	Block 31
Block 31	Statement 31	Block 32
Block 32	Statement 32	Block 33
Block 33	Statement 33	Block 34
Block 34	Statement 34	Block 35
Block 35	Statement 35	Block 36
Block 36	Statement 36	Block 37
Block 37	Statement 37	Block 38
Block 38	Statement 38	Block 39
Block 39	Statement 39	Block 40
Block 40	Statement 40	Block 41
Block 41	Statement 41	Block 42
Block 42	Statement 42	Block 43
Block 43	Statement 43	Block 44
Block 44	Statement 44	Block 45
Block 45	Statement 45	Block 46
Block 46	Statement 46	Block 47
Block 47	Statement 47	Block 48
Block 48	Statement 48	Block 49
Block 49	Statement 49	Block 50
Block 50	Statement 50	Block 51
Block 51	Statement 51	Block 52
Block 52	Statement 52	Block 53
Block 53	Statement 53	Block 54
Block 54	Statement 54	Block 55
Block 55	Statement 55	Block 56
Block 56	Statement 56	Block 57
Block 57	Statement 57	Block 58
Block 58	Statement 58	Block 59
Block 59	Statement 59	Block 60
Block 60	Statement 60	Block 61
Block 61	Statement 61	Block 62
Block 62	Statement 62	Block 63
Block 63	Statement 63	Block 64
Block 64	Statement 64	Block 65
Block 65	Statement 65	Block 66
Block 66	Statement 66	Block 67
Block 67	Statement 67	Block 68
Block 68	Statement 68	Block 69
Block 69	Statement 69	Block 70
Block 70	Statement 70	Block 71
Block 71	Statement 71	Block 72
Block 72	Statement 72	Block 73
Block 73	Statement 73	Block 74
Block 74	Statement 74	Block 75
Block 75	Statement 75	Block 76
Block 76	Statement 76	Block 77
Block 77	Statement 77	Block 78
Block 78	Statement 78	Block 79
Block 79	Statement 79	Block 80
Block 80	Statement 80	Block 81
Block 81	Statement 81	Block 82
Block 82	Statement 82	Block 83
Block 83	Statement 83	Block 84
Block 84	Statement 84	Block 85
Block 85	Statement 85	Block 86
Block 86	Statement 86	Block 87
Block 87	Statement 87	Block 88
Block 88	Statement 88	Block 89
Block 89	Statement 89	Block 90
Block 90	Statement 90	Block 91
Block 91	Statement 91	Block 92
Block 92	Statement 92	Block 93
Block 93	Statement 93	Block 94
Block 94	Statement 94	Block 95
Block 95	Statement 95	Block 96
Block 96	Statement 96	Block 97
Block 97	Statement 97	Block 98
Block 98	Statement 98	Block 99
Block 99	Statement 99	Block 100

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**: 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.

- **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 registers to 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;		
}		