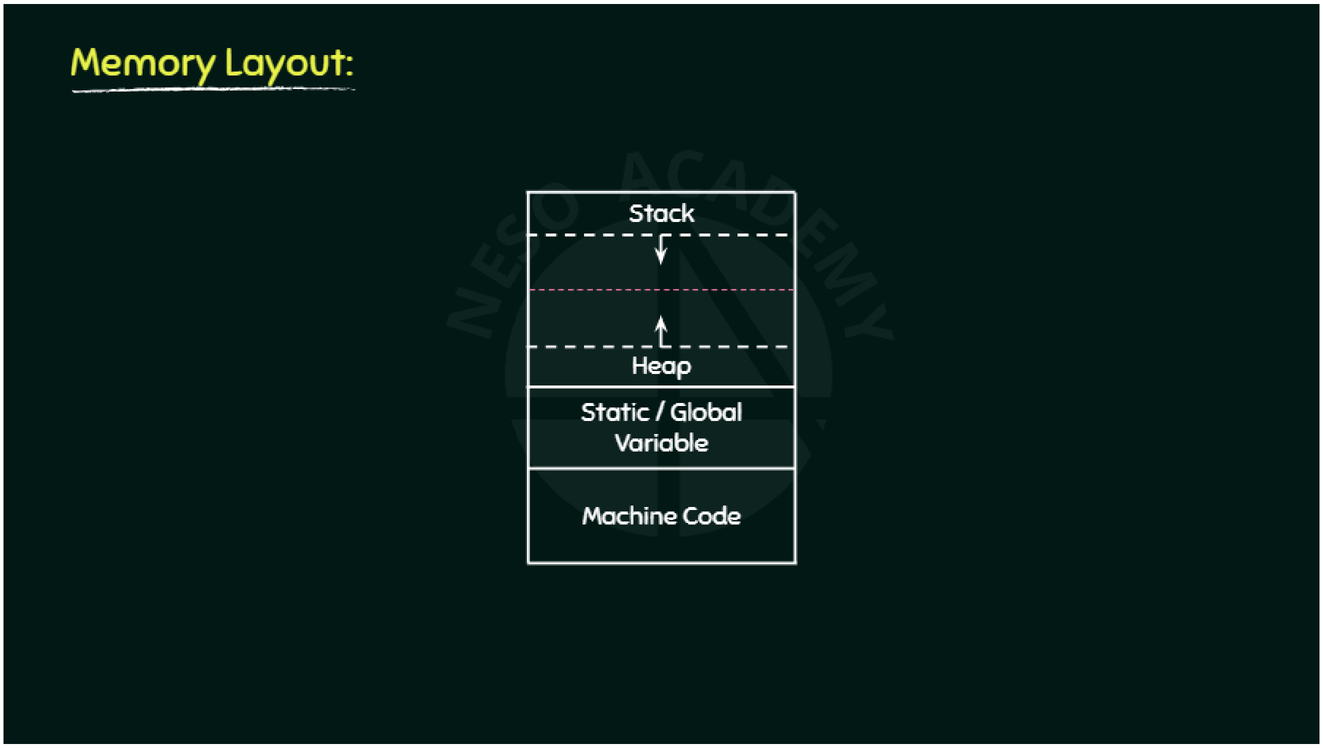
Runtime Environment & Code Optimization

nesoacademy.org/cs/12-compiler-design/ppts/07-runtimeenvironment&codeoptimization

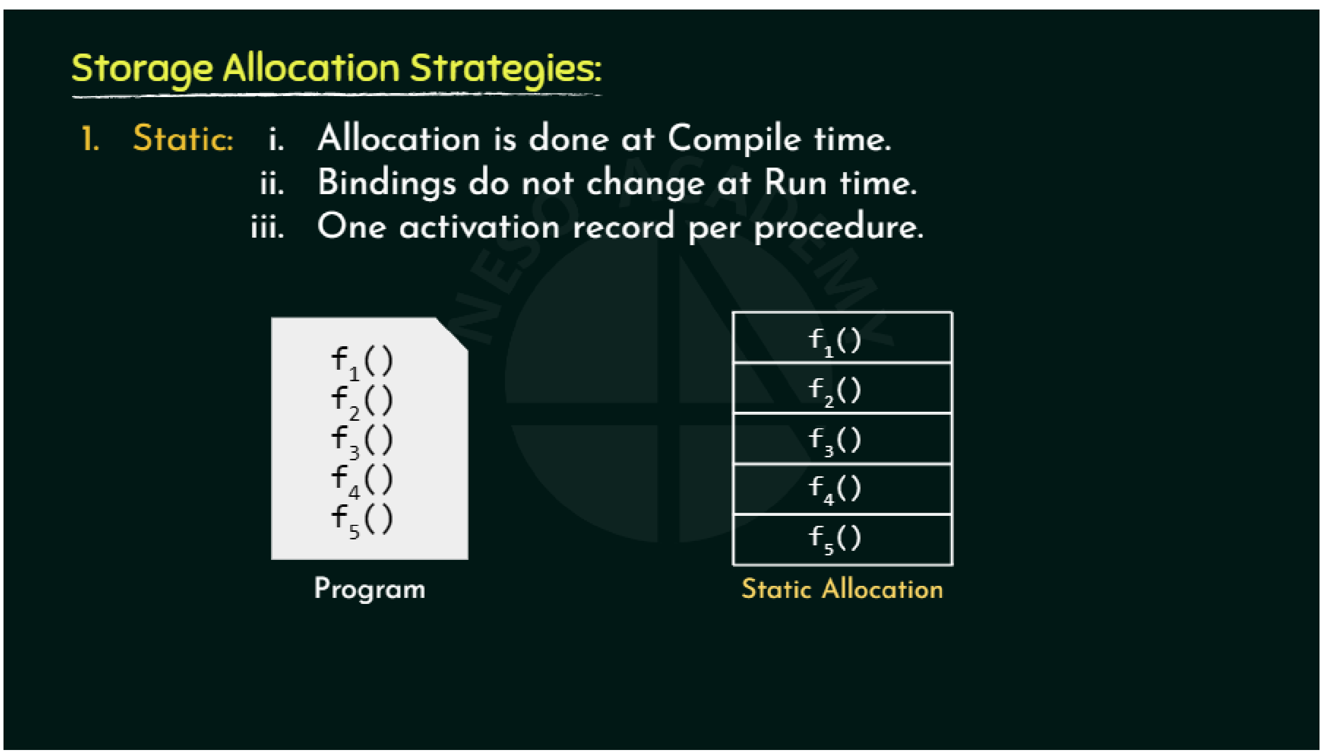
Runtime Environment And Code OptimizationNeso AcademyCHAPTER-7

Compiler Design Runtime Environment

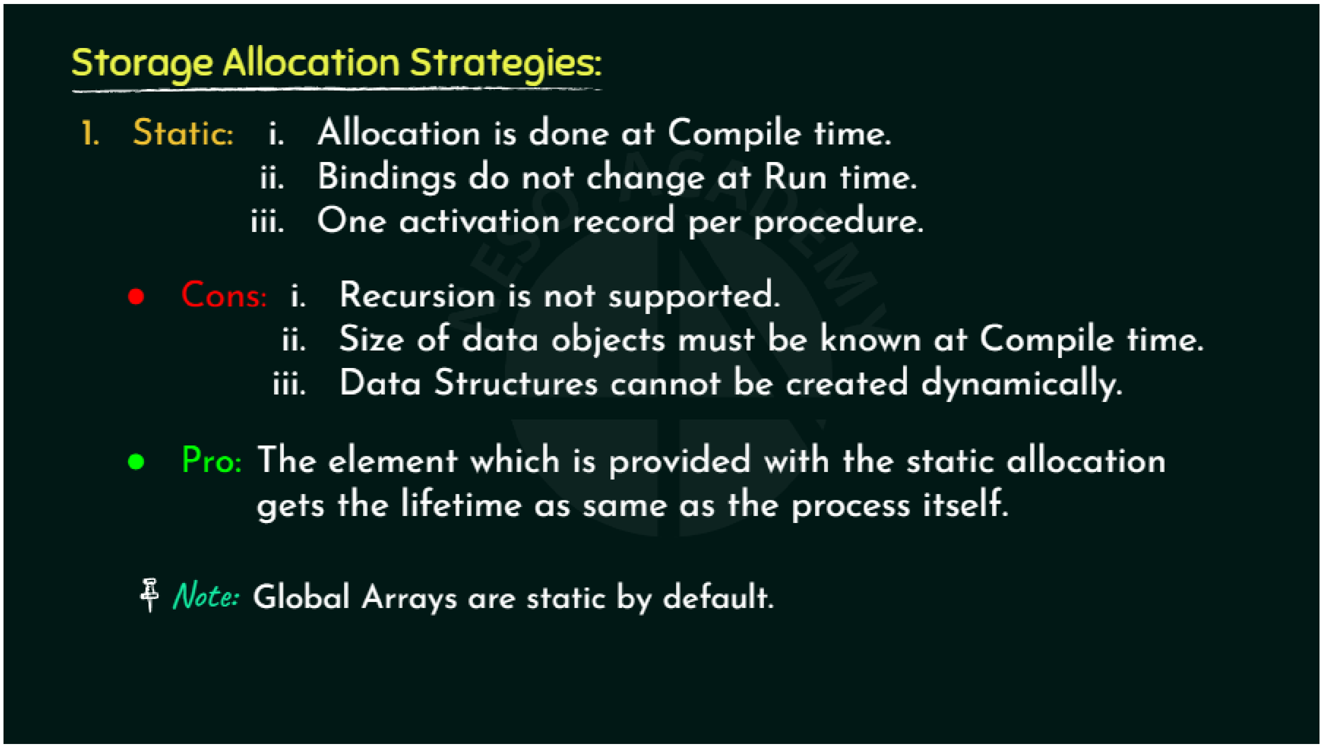
Outcome☆Different Storage Allocation Strategies.



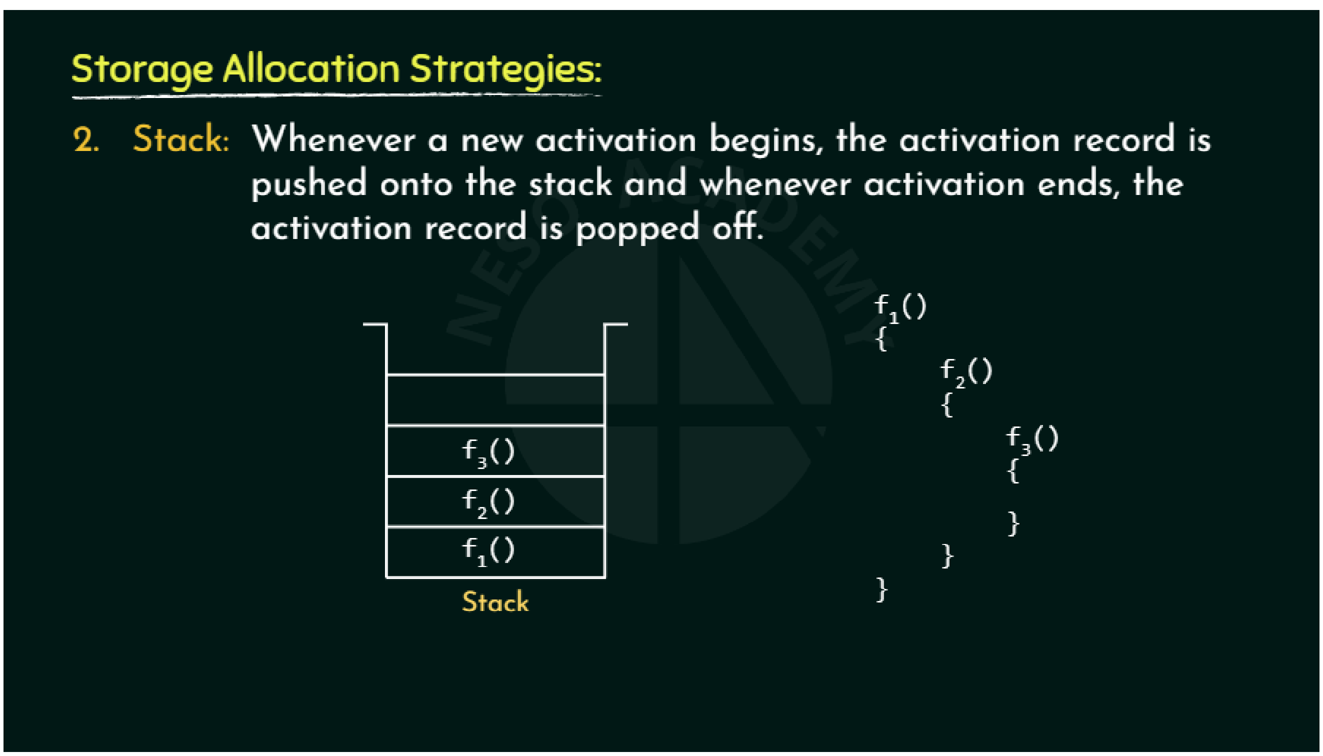
Static / Global VariableStackHeapMachine CodeMemory Layout:



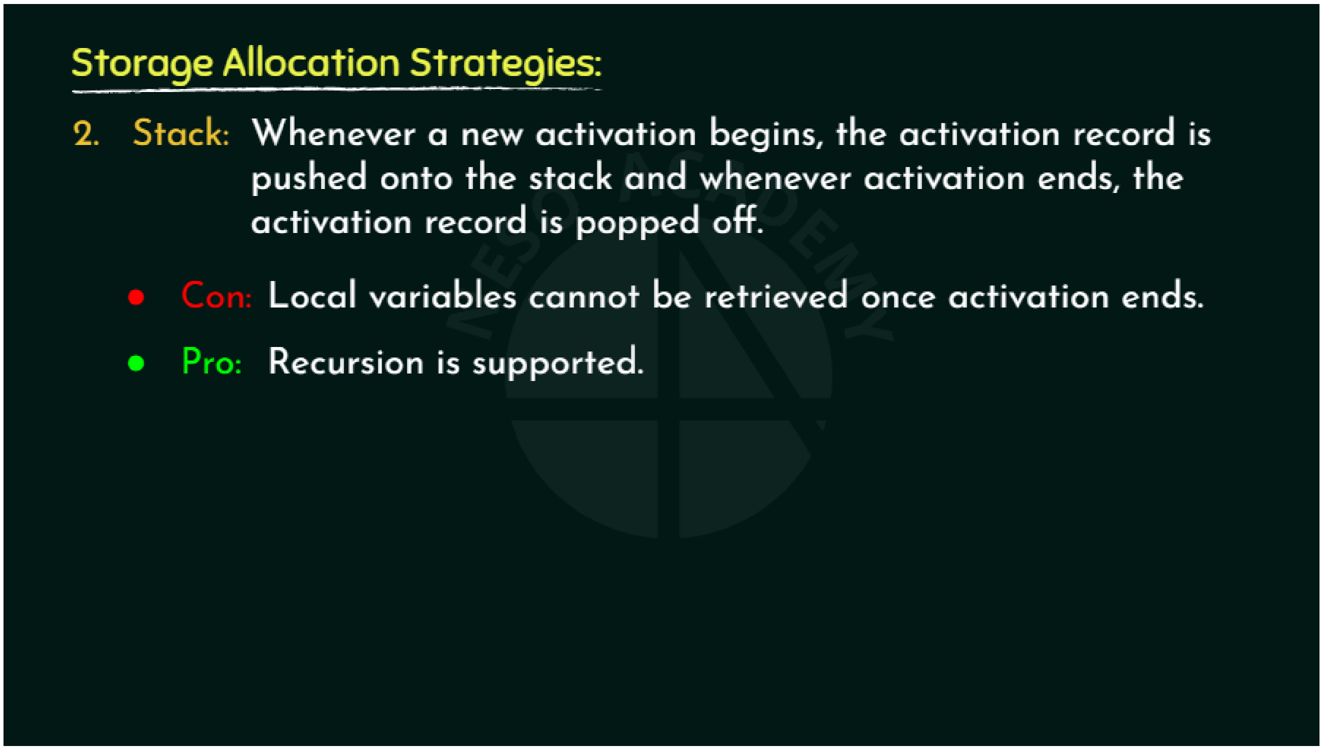
Storage Allocation Strategies:1.Static:i.Allocation is done at Compile time.ii.Bindings do not change at Run time.iii.One activation record per procedure. f1()f2()f3()f4()f5()Static Allocationf1()f2()f3()f4()f5()Program



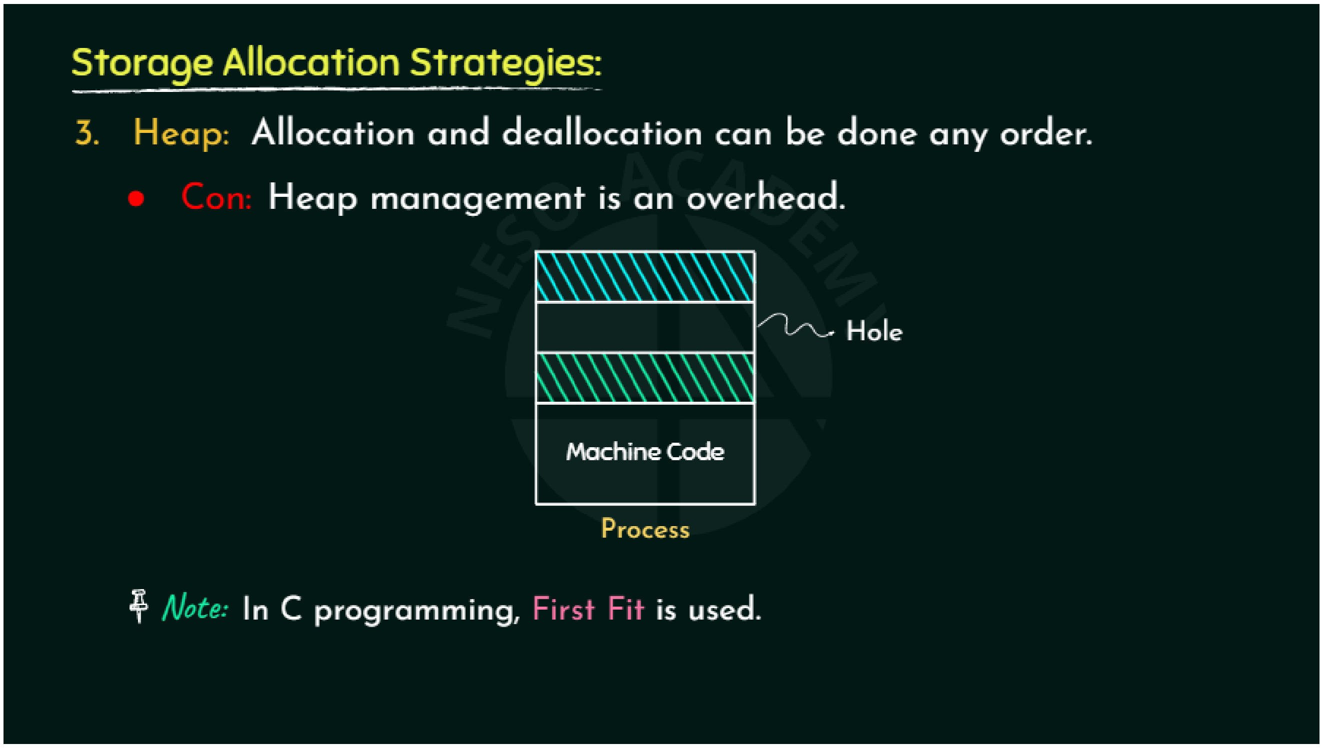
Storage Allocation Strategies:1.Static:i.Allocation is done at Compile time.ii.Bindings do not change at Run time.iii.One activation record per procedure. ●Cons:i.Recursion is not supported.ii.Size of data objects must be known at Compile time.iii.Data Structures cannot be created dynamically. ●Pro:The element which is provided with the static allocation gets the lifetime as same as the process itself.Note:Global Arrays are static by default.



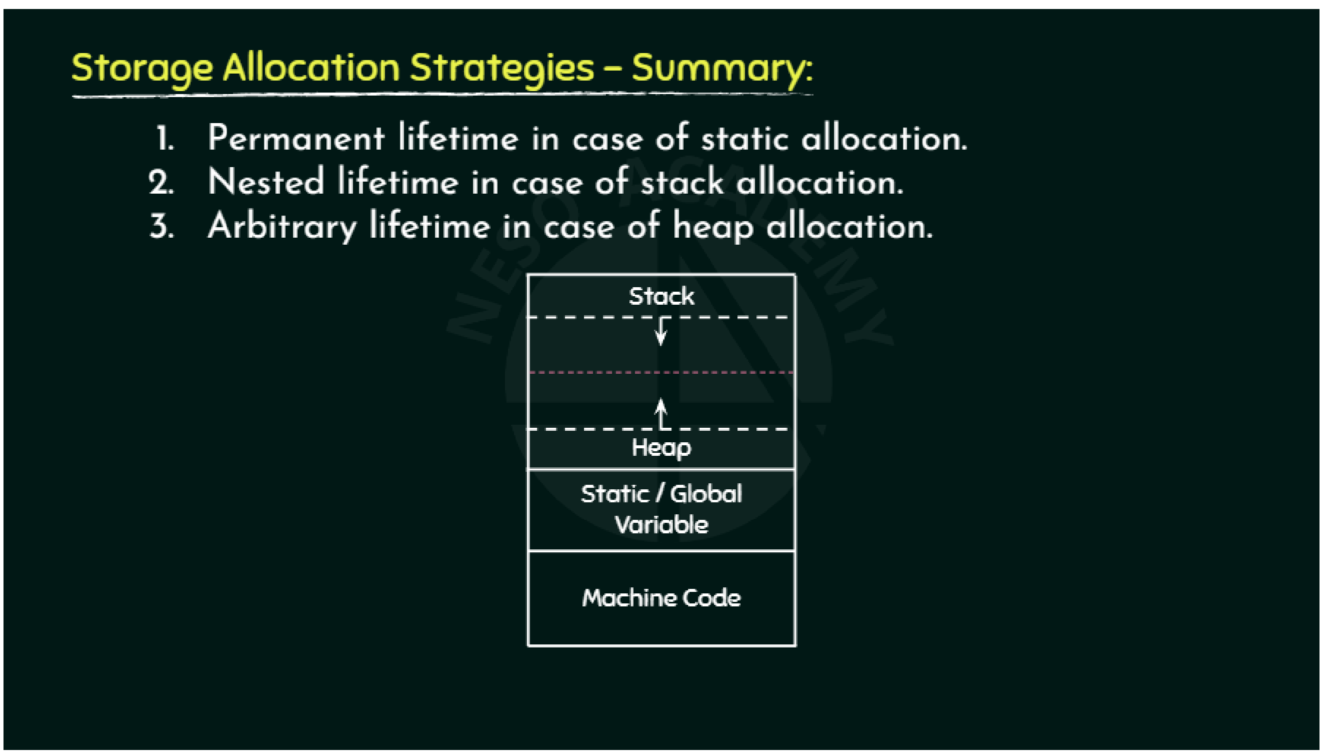
Storage Allocation Strategies:2.Stack:Whenever a new activation begins, the activation record is pushed onto the stack and whenever activation ends, the activation record is popped off.f1()f2()f3()Stackf2(){f3(){}}f3(){}f1(){f2(){f3(){}}}



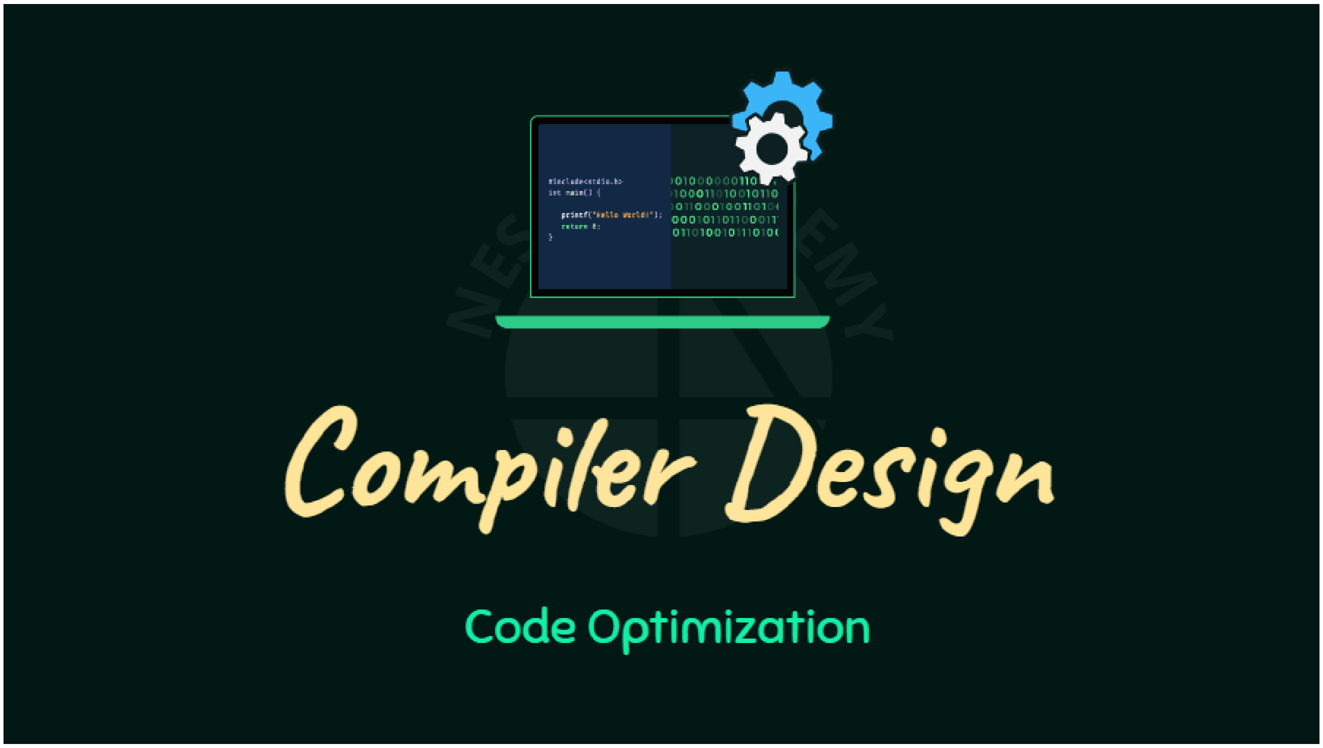
Storage Allocation Strategies:Whenever a new activation begins, the activation record is pushed onto the stack and whenever activation ends, the activation record is popped off.●Con:Local variables cannot be retrieved once activation ends. ●Pro:Recursion is supported.2.Stack:



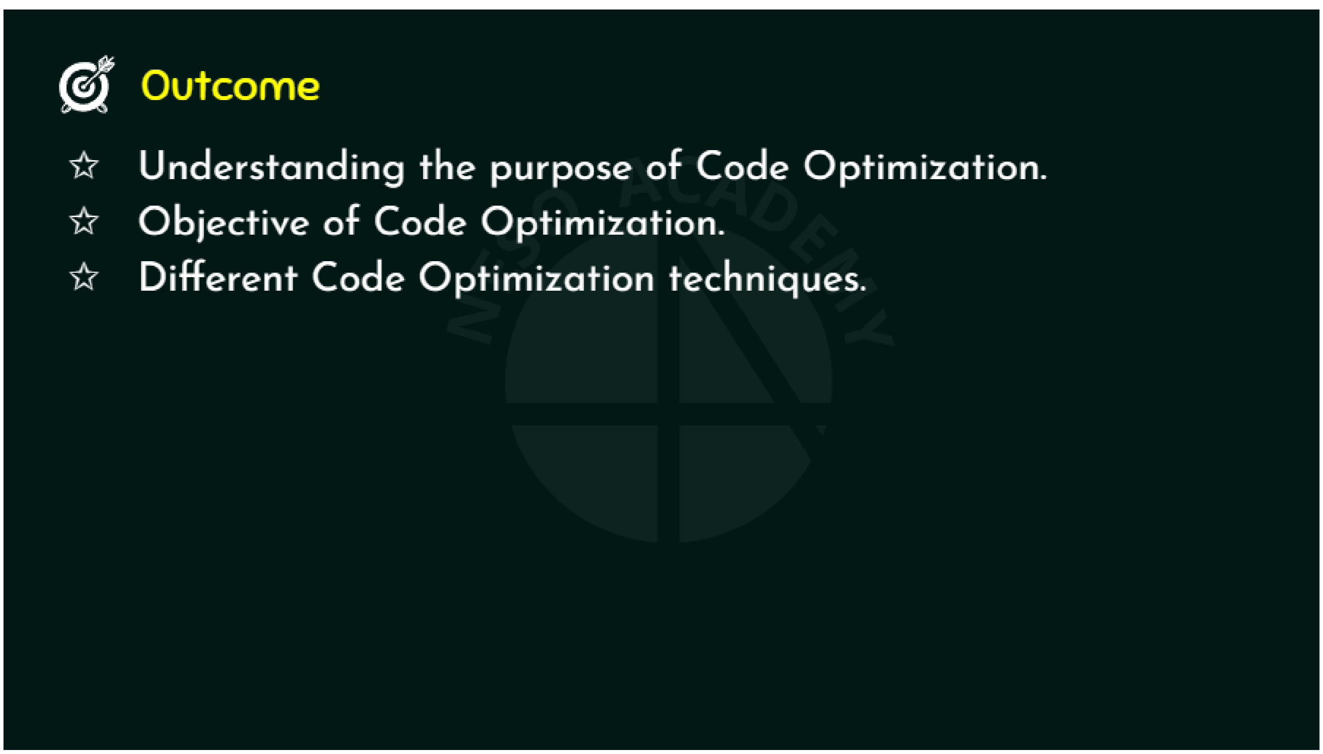
Storage Allocation Strategies:Allocation and deallocation can be done any order.●Con:Heap management is an overhead. 3.Heap:ProcessMachine CodeHoleNote:In C programming, First Fit is used.



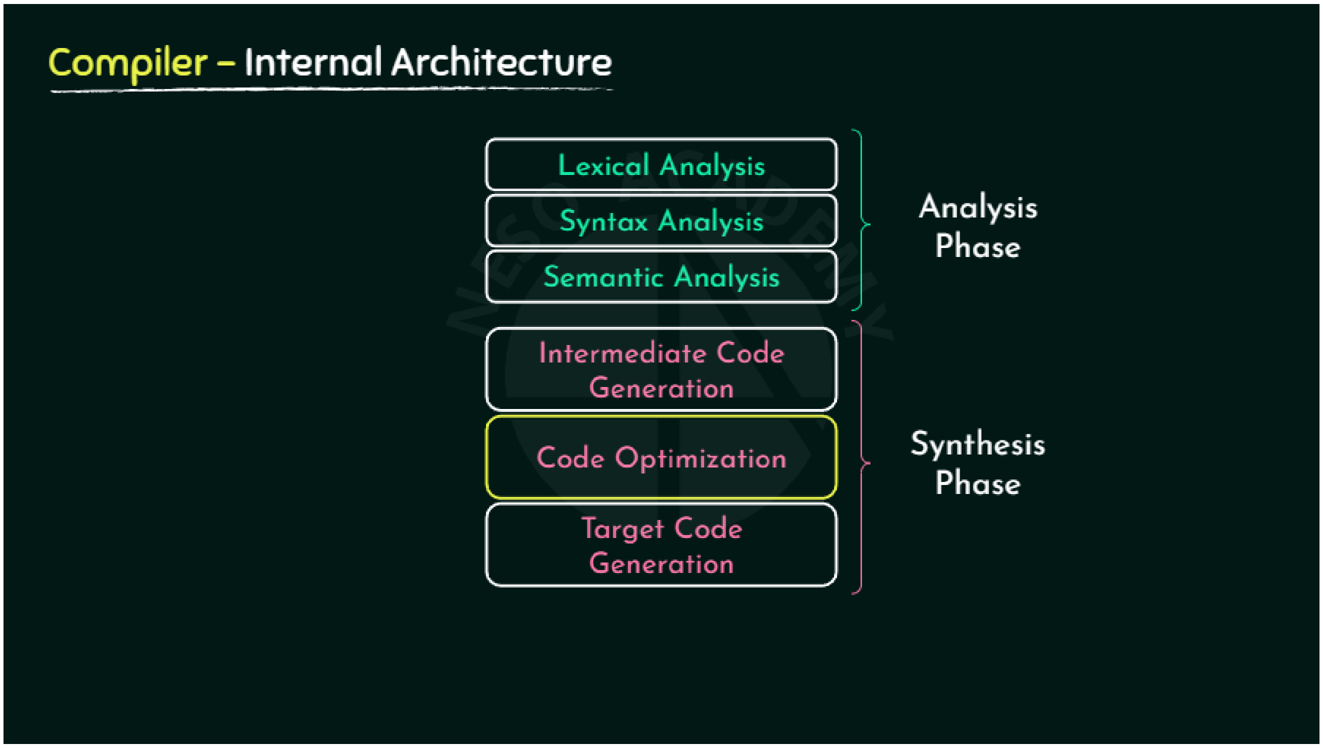
Storage Allocation Strategies - Summary:1.Permanent lifetime in case of static allocation.2.Nested lifetime in case of stack allocation.3.Arbitrary lifetime in case of heap allocation.Static / Global VariableStackHeapMachine Code



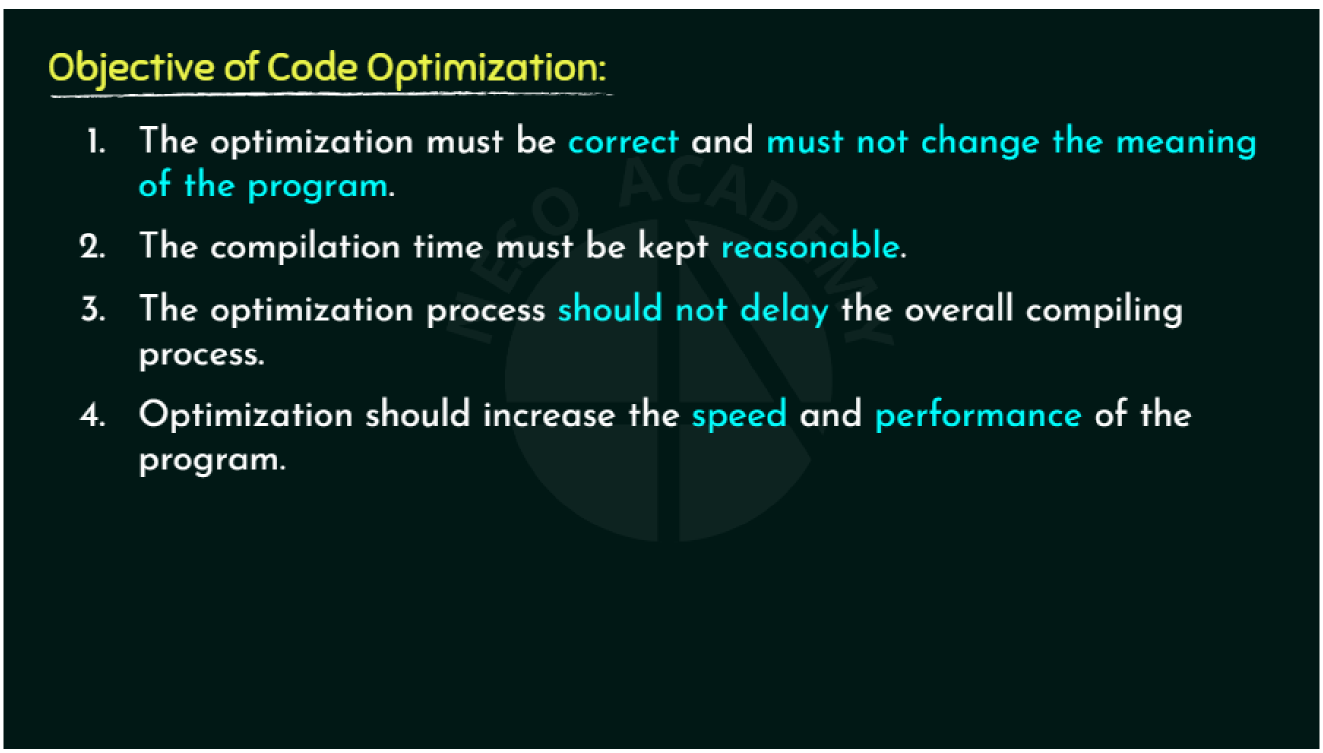
Compiler Design Code Optimization



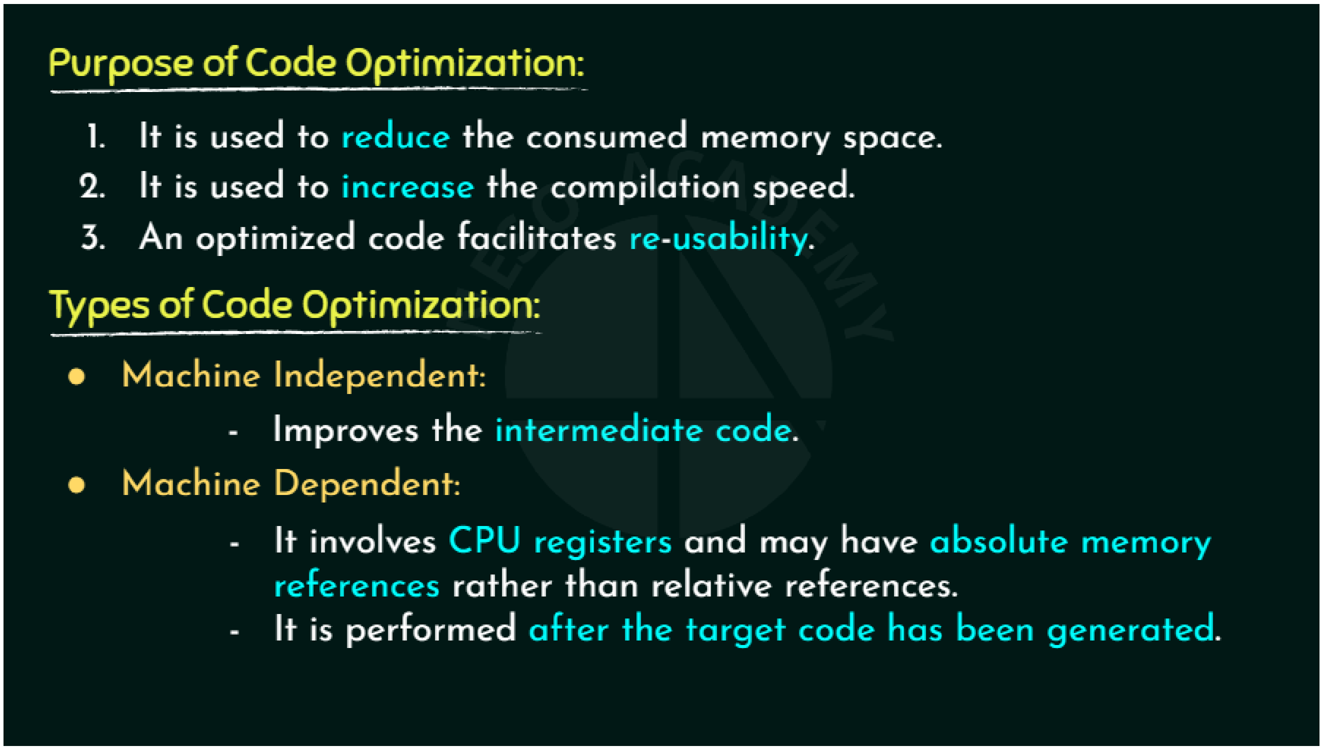
Outcome☆Understanding the purpose of Code Optimization.☆Objective of Code Optimization.☆Different Code Optimization techniques.



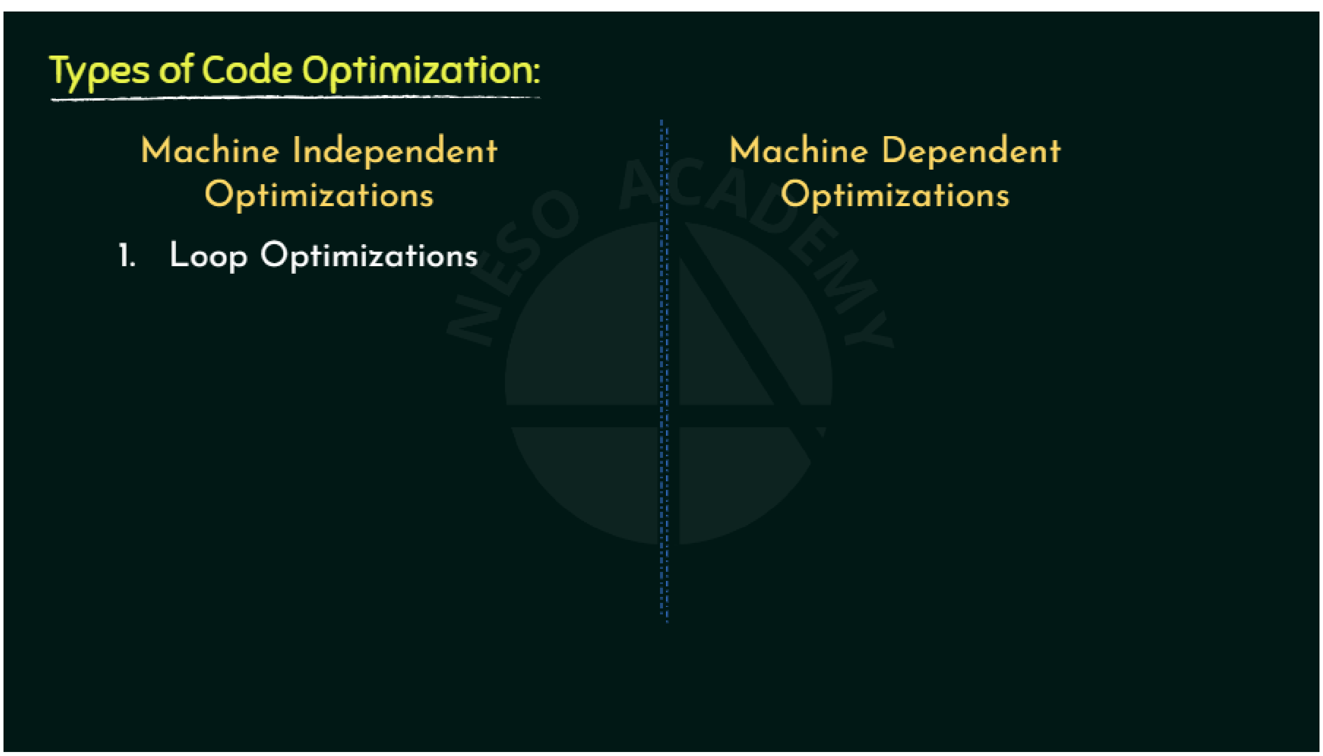
Lexical AnalysisSyntax AnalysisSemantic AnalysisIntermediate Code GenerationCode OptimizationTarget Code GenerationAnalysis PhaseSynthesis PhaseCompiler - Internal Architecture



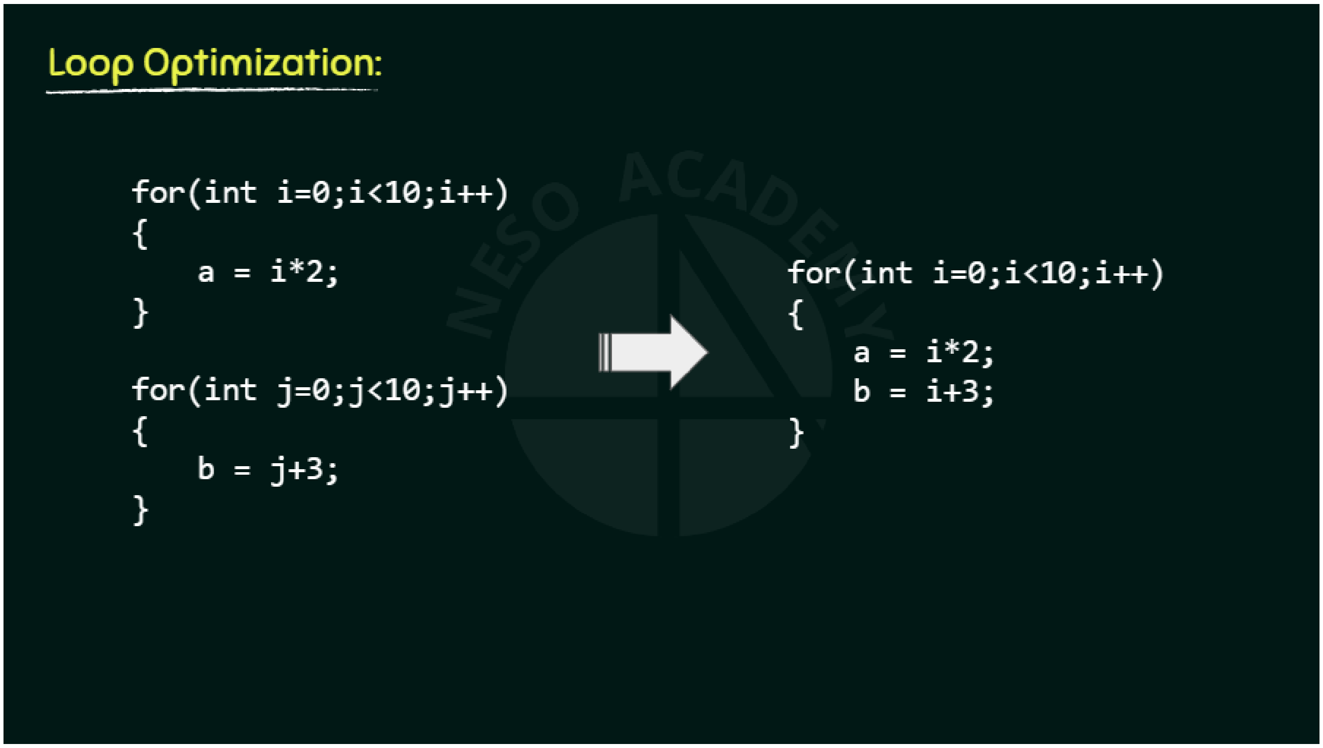
Objective of Code Optimization:1.The optimization must be correct and must not change the meaning of the program.2.The compilation time must be kept reasonable.3.The optimization process should not delay the overall compiling process.4.Optimization should increase the speed and performance of the program.



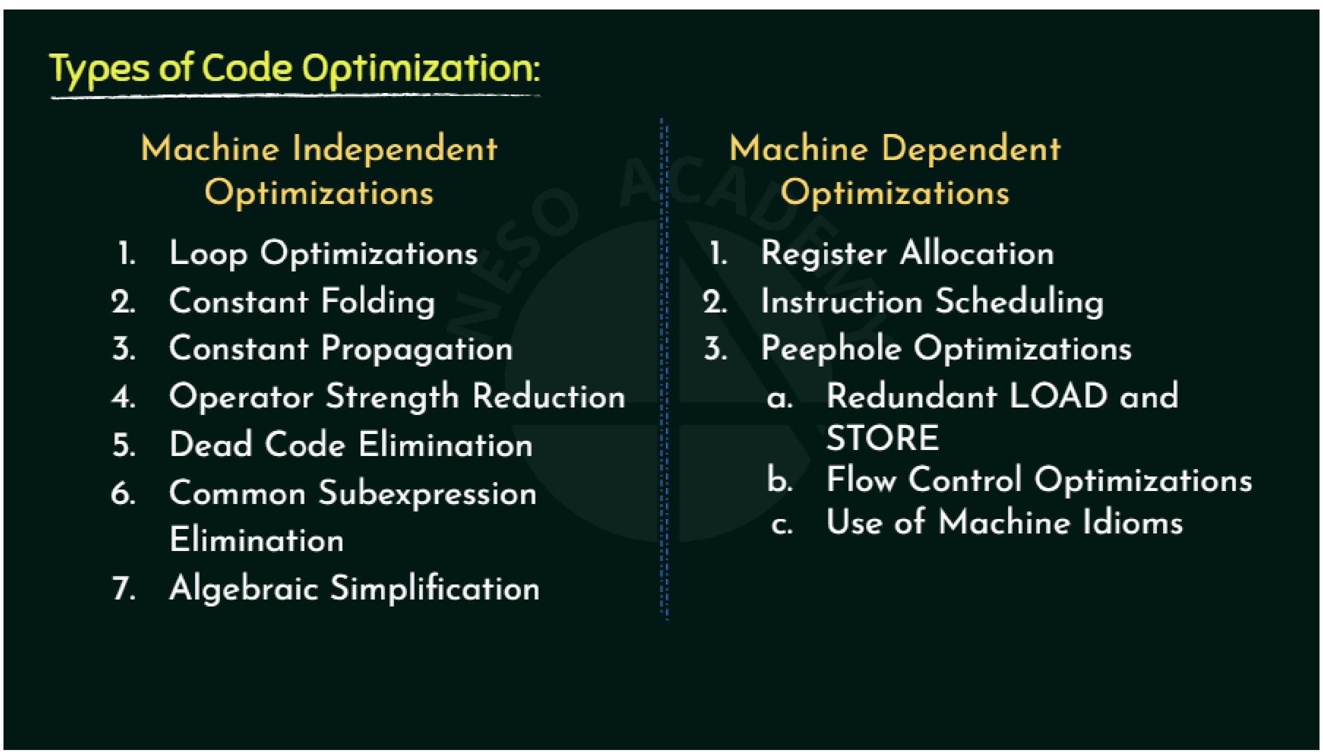
Purpose of Code Optimization:1.It is used to reduce the consumed memory space.2.It is used to increase the compilation speed.3.An optimized code facilitates re-usability.Types of Code Optimization:●Machine Independent:-Improves the intermediate code.●Machine Dependent:-It involves CPU registers and may have absolute memory references rather than relative references.-It is performed after the target code has been generated.



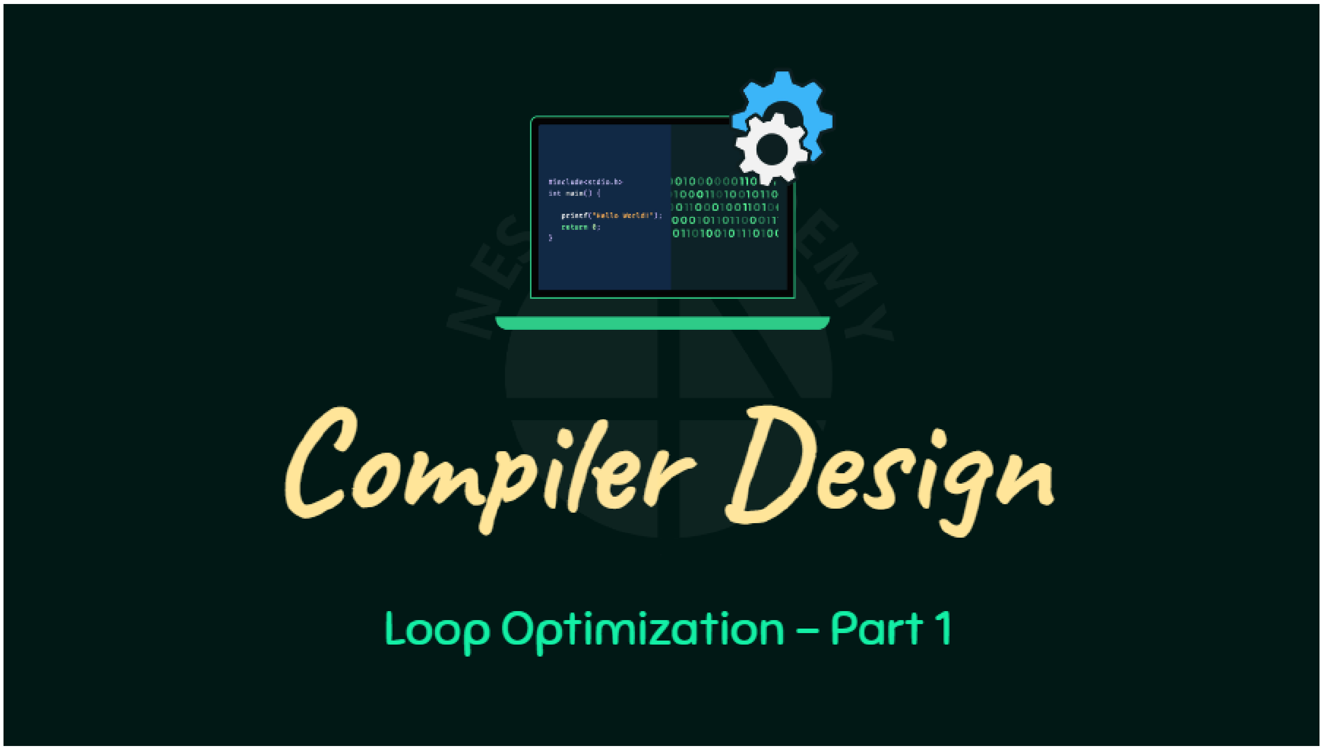
Types of Code Optimization:Machine Independent Optimizations1.Loop OptimizationsMachine Dependent Optimizations



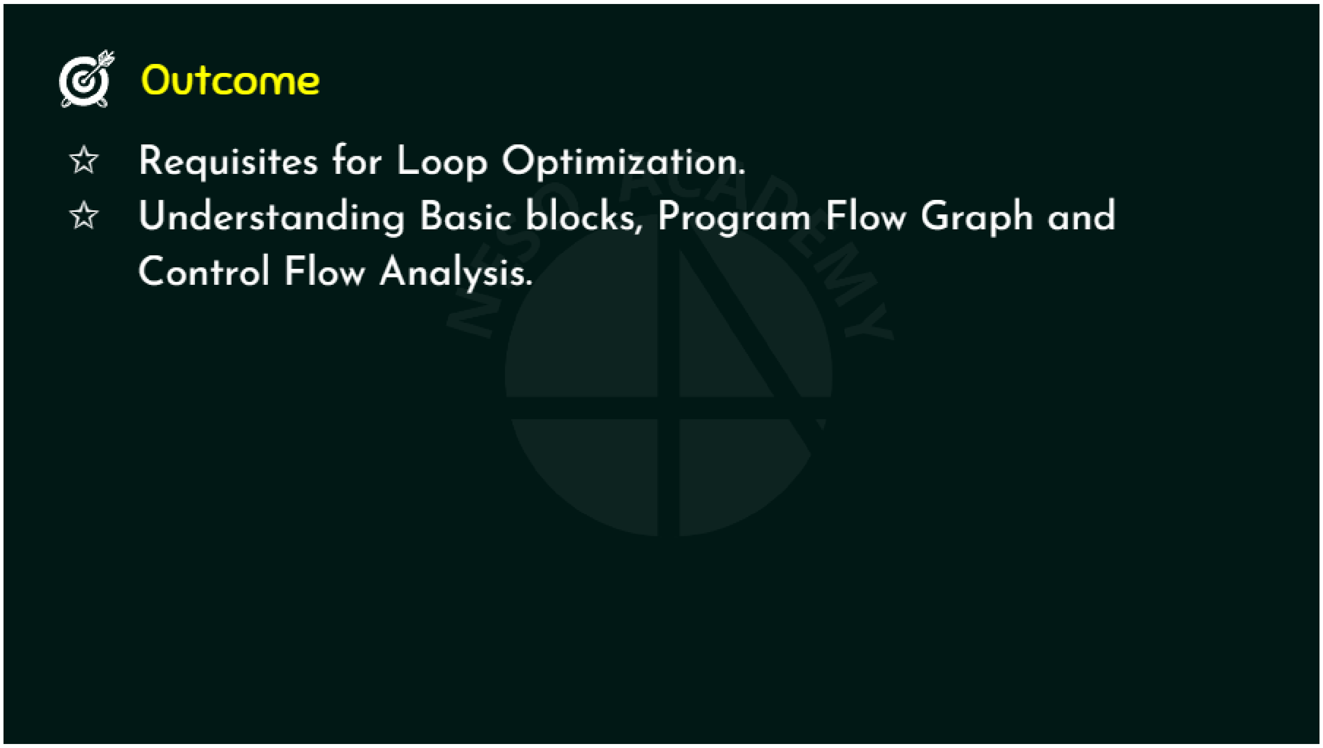
Loop Optimization:for(int i=0;i<10;i++){ a = i*2;} for(int j=0;j<10;j++){ b = j+3;}for(int i=0;i<10;i++){ a = i*2;b = i+3;}



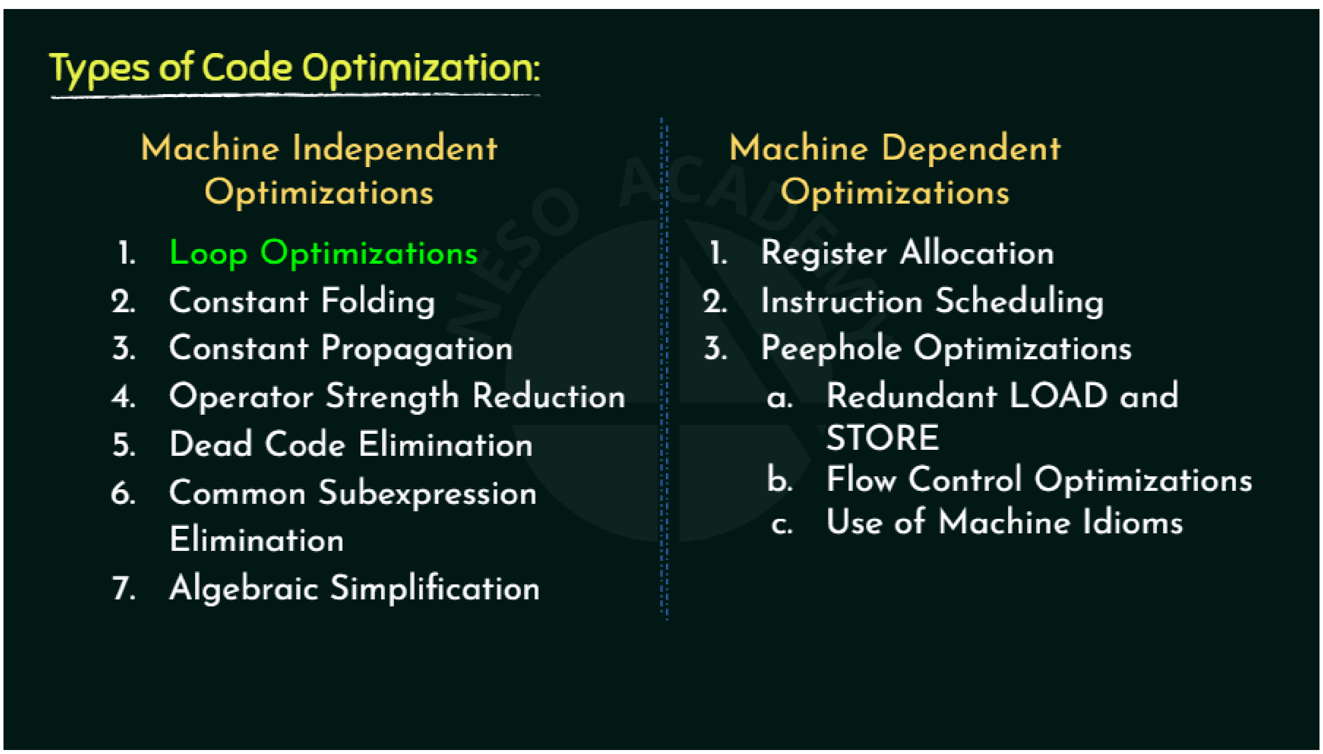
Types of Code Optimization:Machine Independent OptimizationsMachine Dependent Optimizations1.Register Allocation2.Instruction Scheduling 3.Peephole Optimizations a.Redundant LOAD and STOREb.Flow Control Optimizationsc.Use of Machine Idioms1.Loop Optimizations2.Constant Folding3.Constant Propagation4.Operator Strength Reduction5.Dead Code Elimination 6.Common Subexpression Elimination7.Algebraic Simplification1.Loop Optimizations



Compiler Design Loop Optimization - Part 1

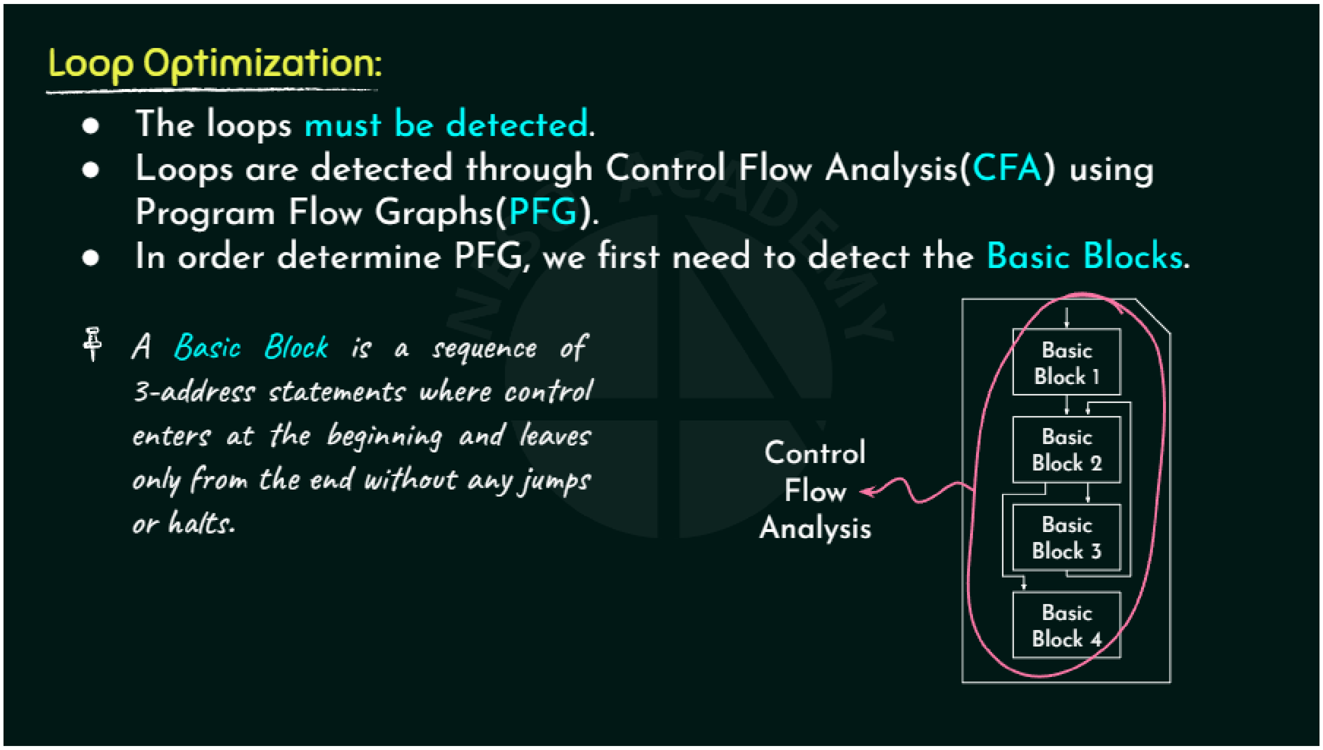


Outcome☆Requisites for Loop Optimization.☆Understanding Basic blocks, Program Flow Graph and Control Flow Analysis.

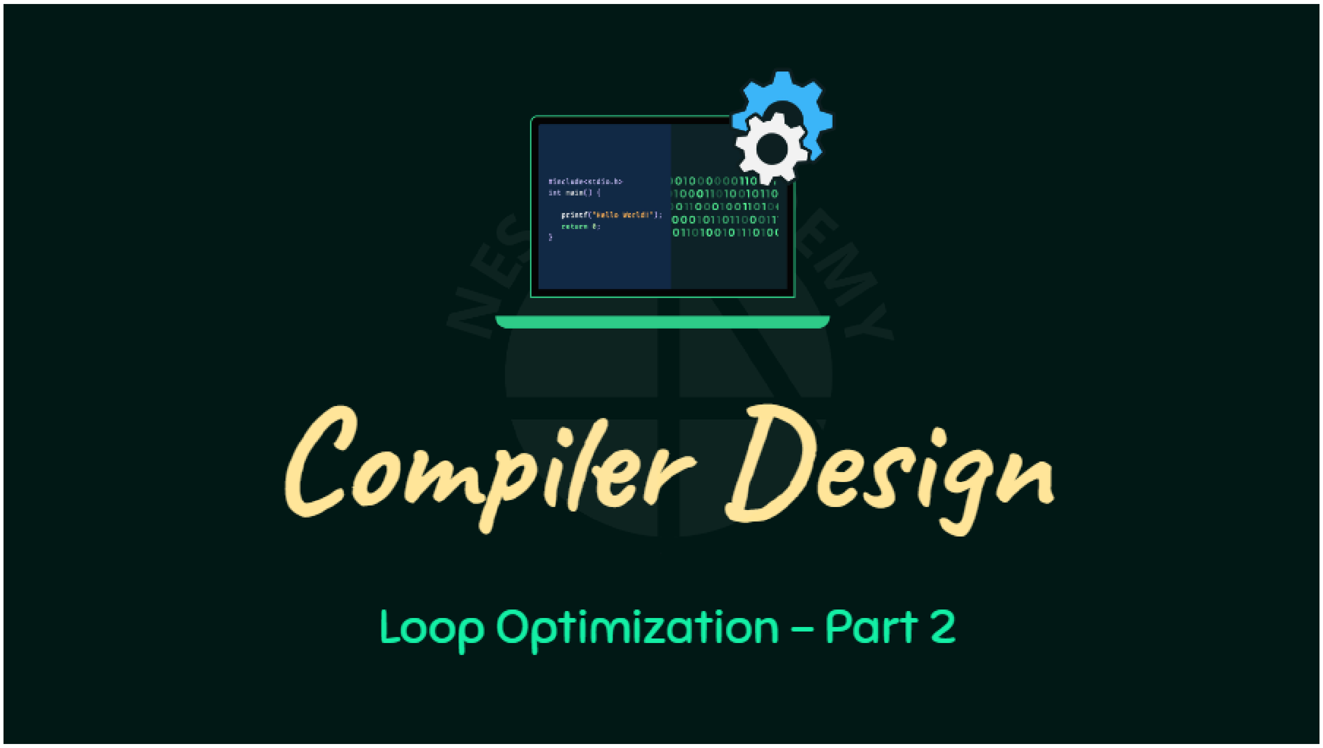


Types of Code Optimization:Machine Independent OptimizationsMachine Dependent Optimizations1.Register Allocation2.Instruction Scheduling 3.Peephole Optimizations a.Redundant LOAD and STOREb.Flow Control Optimizationsc.Use of Machine Idioms1.Loop

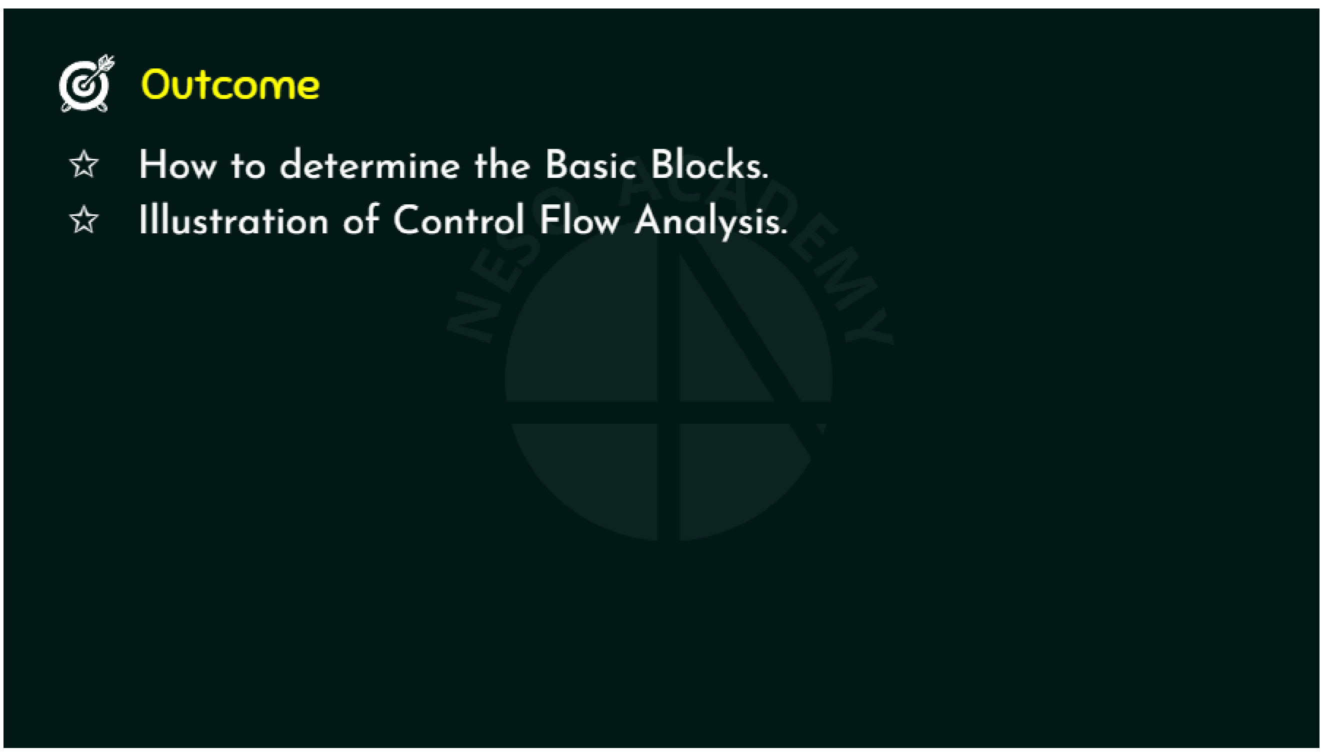
Optimizations2.Constant Folding3.Constant Propagation4.Operator Strength Reduction5.Dead Code Elimination 6.Common Subexpression Elimination7.Algebraic Simplification



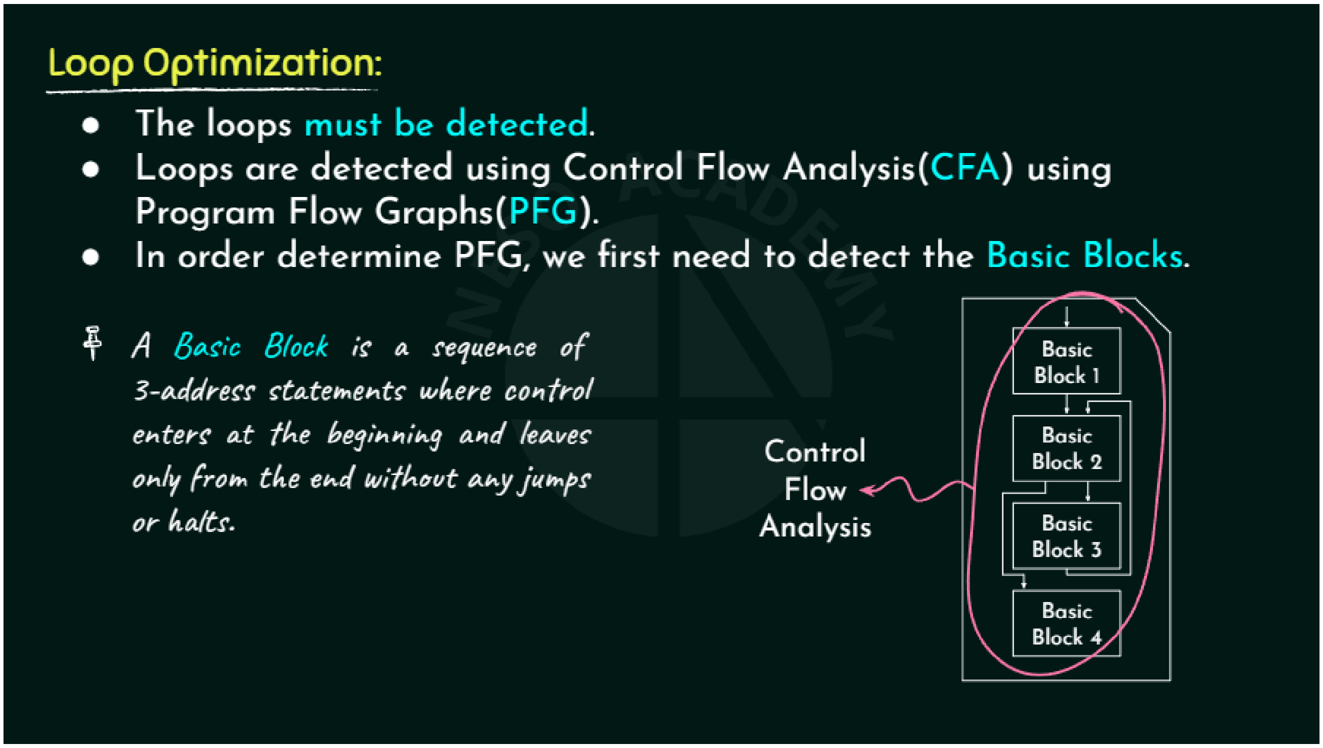
Loop Optimization:●The loops must be detected.●Loops are detected through Control Flow Analysis(CFA) using Program Flow Graphs(PFG).●In order determine PFG, we first need to detect the Basic Blocks. A Basic Block is a sequence of 3-address statements where control enters at the beginning and leaves only from the end without any jumps or halts.Basic Block 1Basic Block 2Basic Block 3Basic Block 4Control Flow Analysis



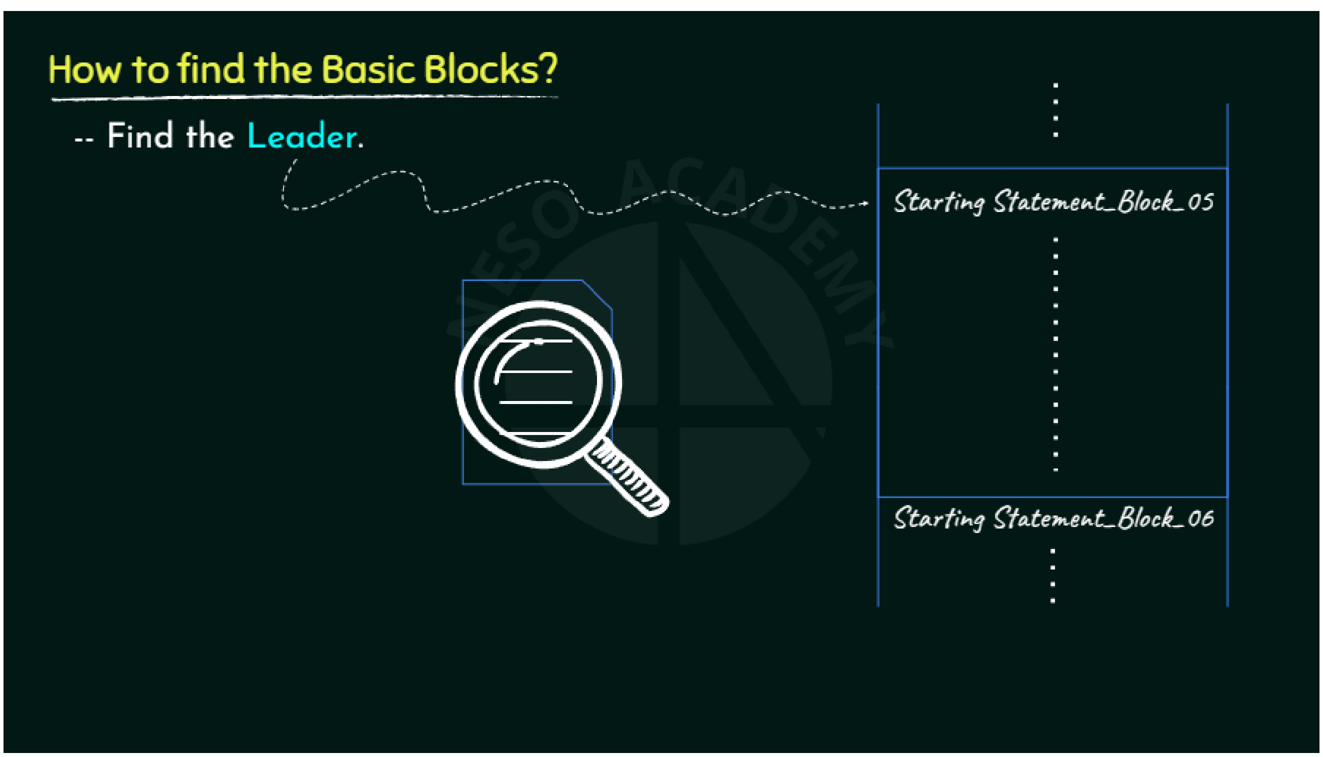
Compiler Design Loop Optimization - Part 2



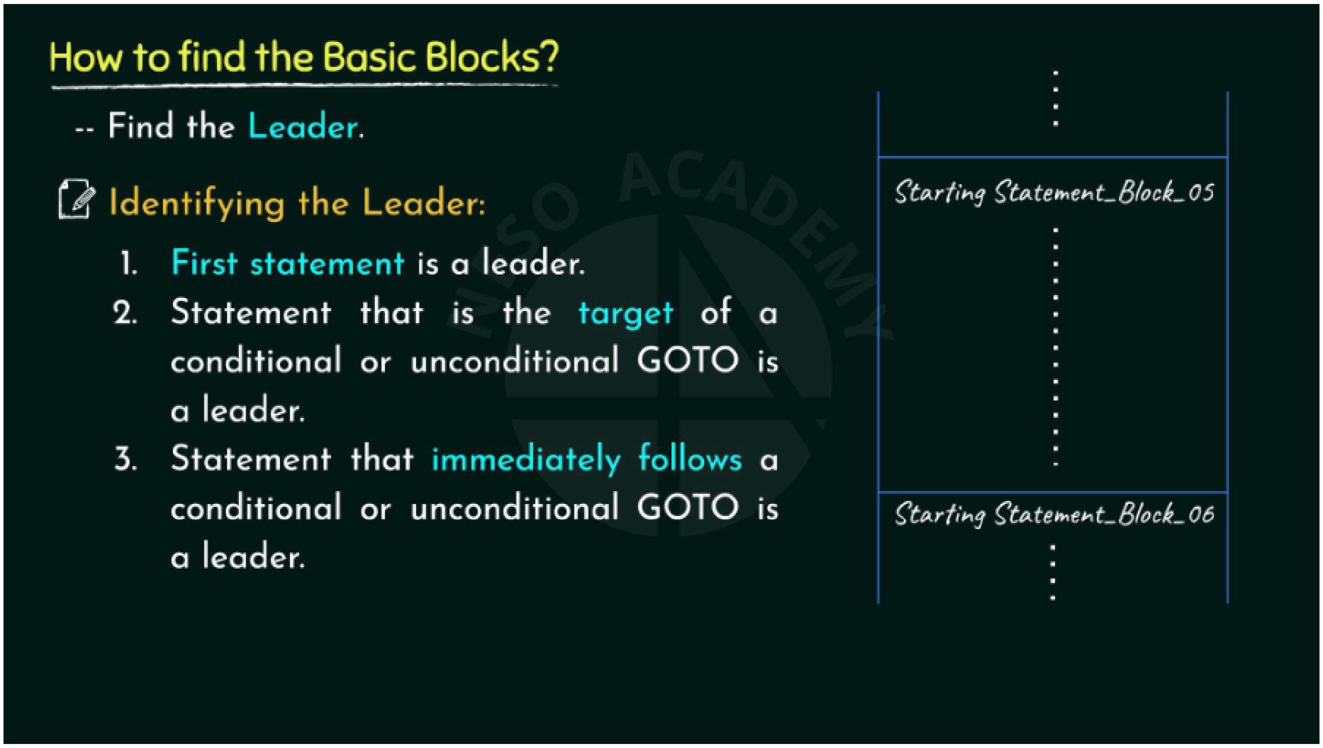
Outcome☆How to determine the Basic Blocks.☆Illustration of Control Flow Analysis.

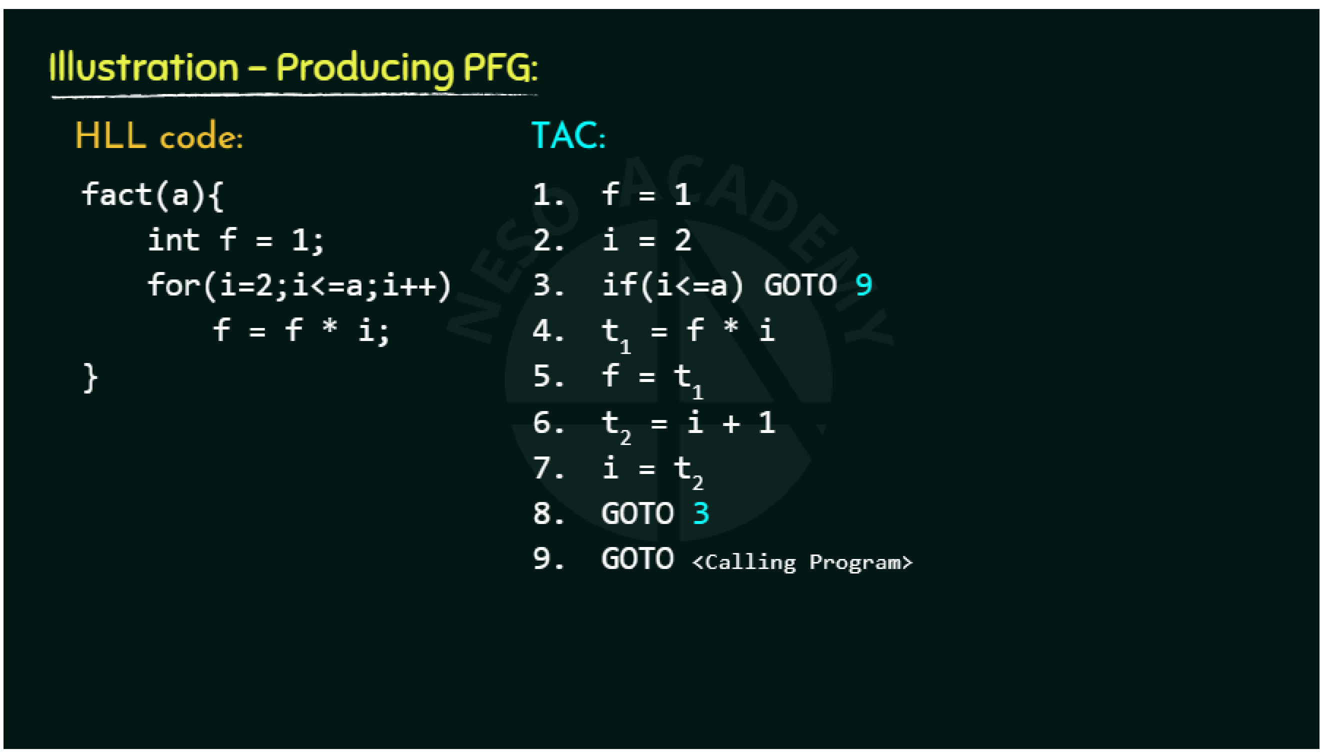


Loop Optimization:●The loops must be detected.●Loops are detected using Control Flow Analysis(CFA) using Program Flow Graphs(PFG).●In order determine PFG, we first need to detect the Basic Blocks. A Basic Block is a sequence of 3-address statements where control enters at the beginning and leaves only from the end without any jumps or halts.Basic Block 1Basic Block 2Basic Block 3Basic Block 4Control Flow Analysis

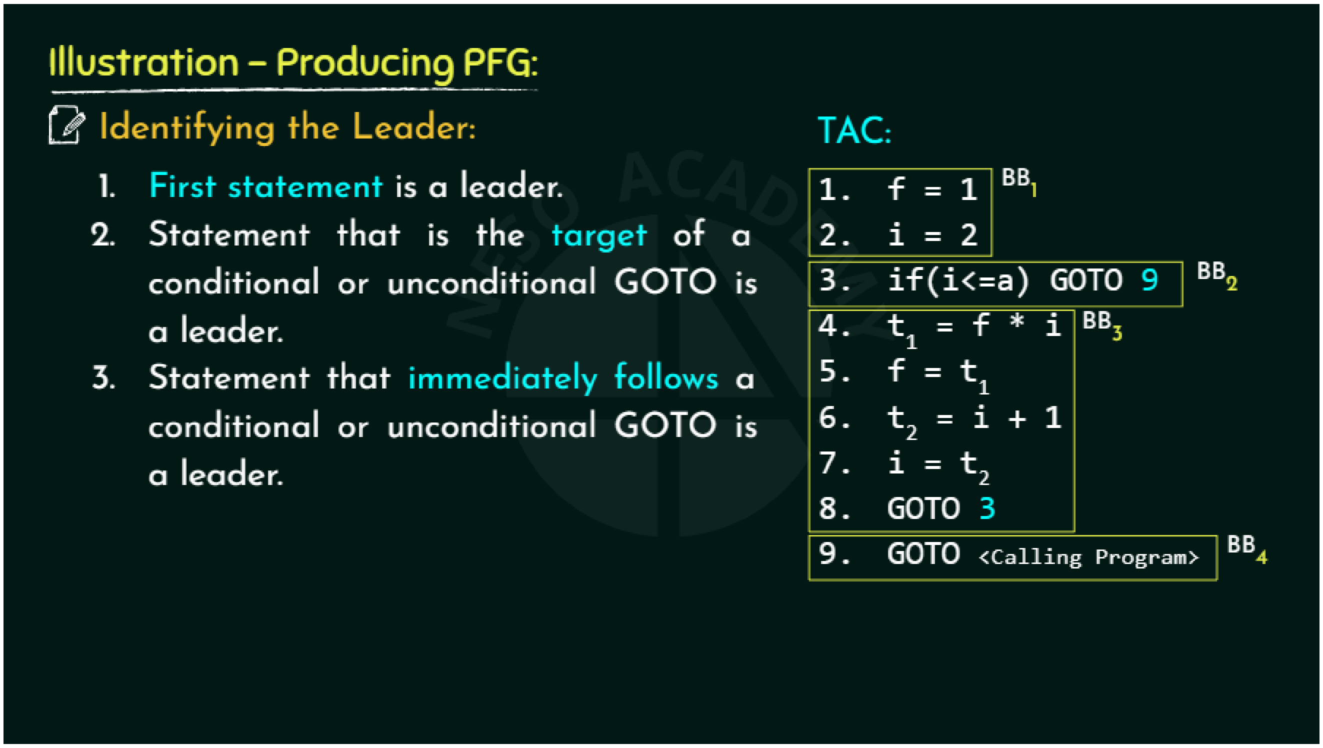


How to find the Basic Blocks?-- Find the Leader.Starting Statement_Block_05Starting Statement_Block_06

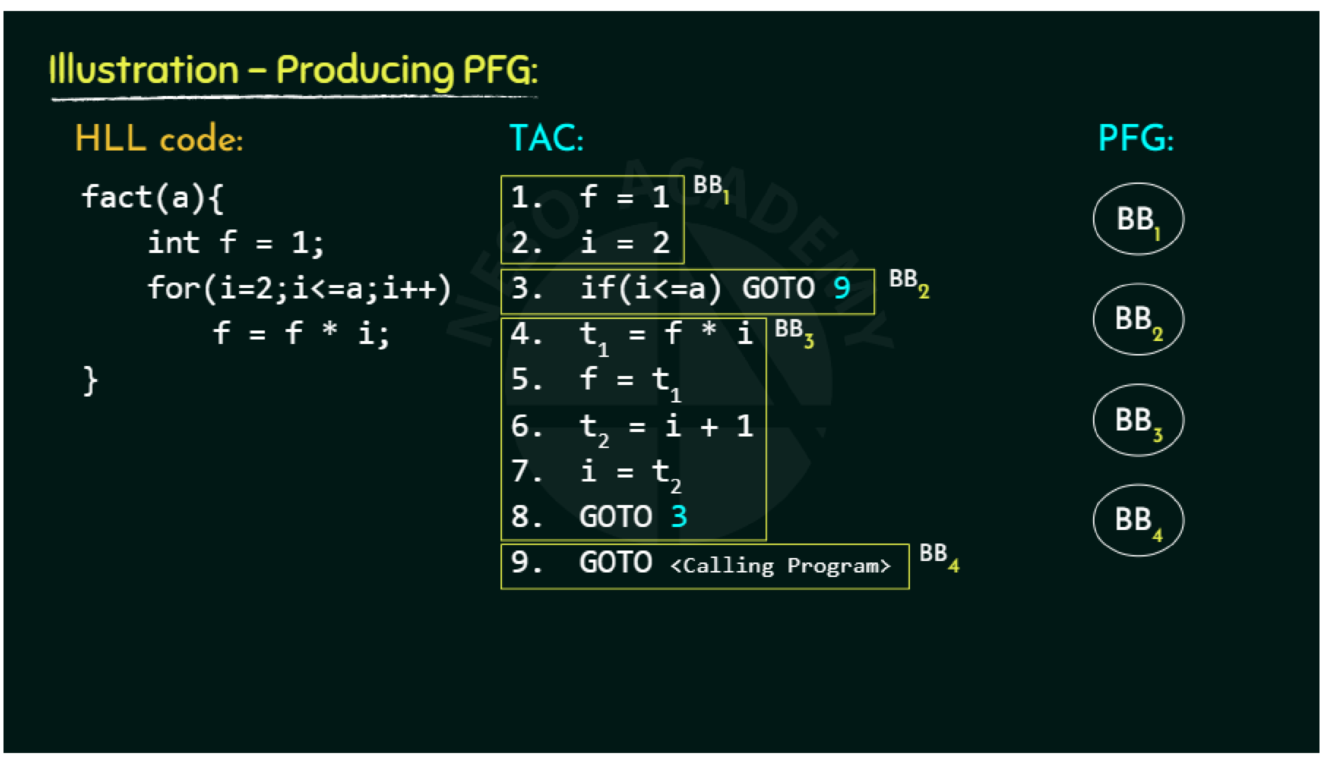




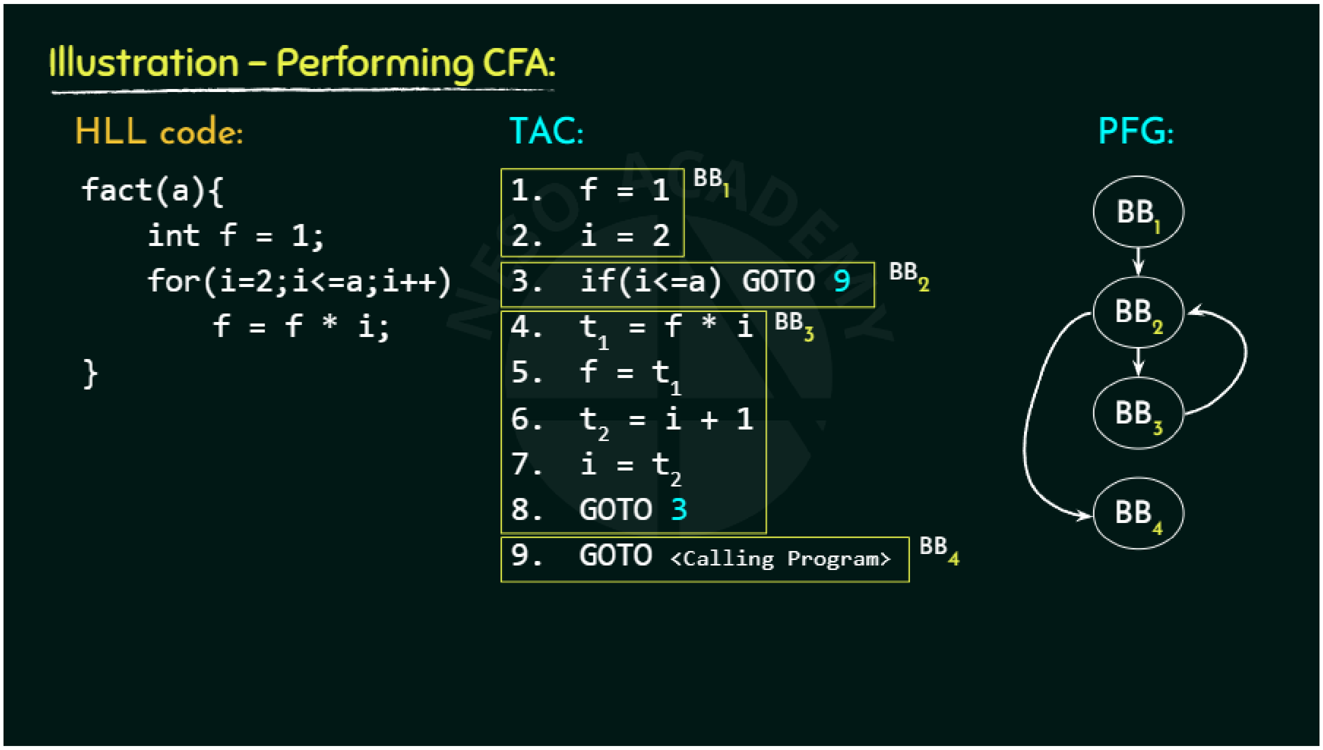
HLL code:fact(a){int f = 1;for(i=2;i<=a;i++)f = f * i;}TAC:1.f = 12.i = 23.if(i<=a) GOTO 94.t1 = f * i5.f = t1 6.t2 = i + 17.i = t2 8.GOTO 39.GOTO <Calling Program>Illustration - Producing PFG:



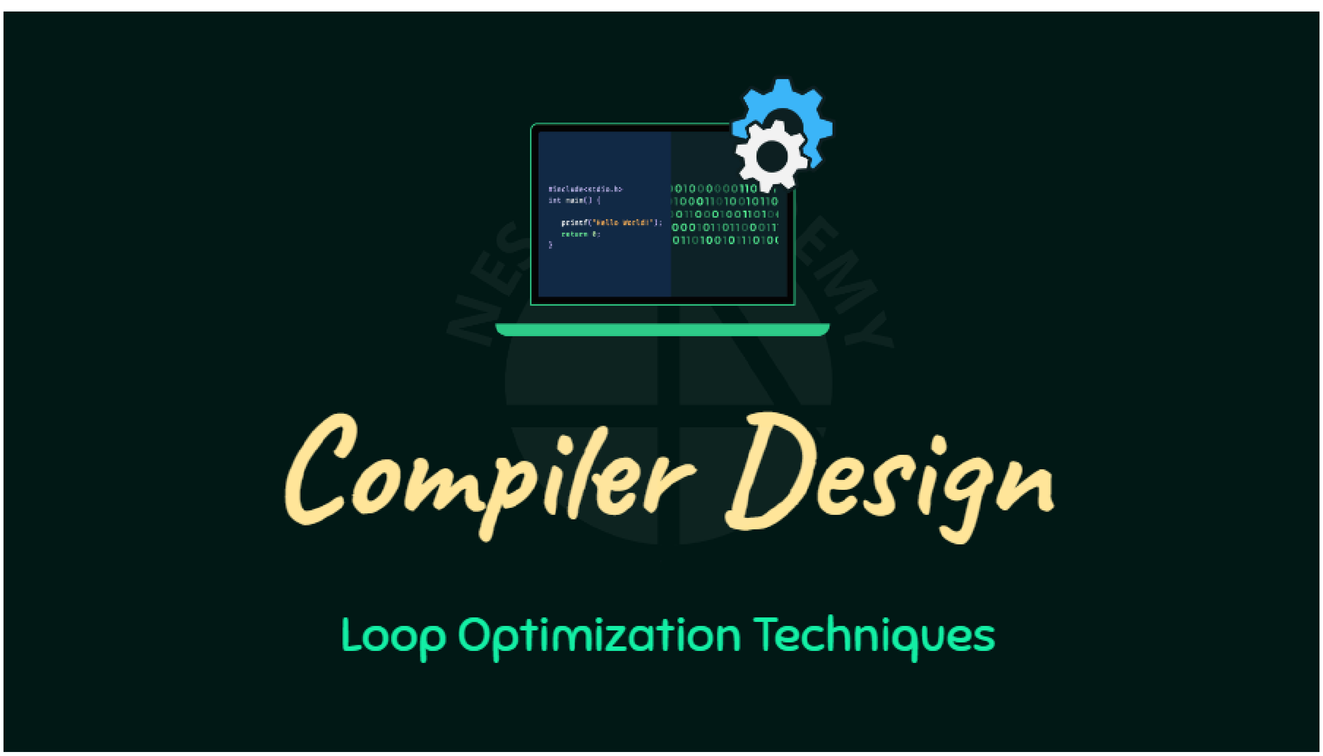
TAC:1.f = 12.i = 23.if(i<=a) GOTO 94.t1 = f * i5.f = t1 6.t2 = i + 17.i = t2 8.GOTO 39.GOTO <Calling Program>BB1BB2BB3BB4Identifying the Leader:1.First statement is a leader.2.Statement that is the target of a conditional or unconditional GOTO is a leader.3.Statement that immediately follows a conditional or unconditional GOTO is a leader.Illustration - Producing PFG:

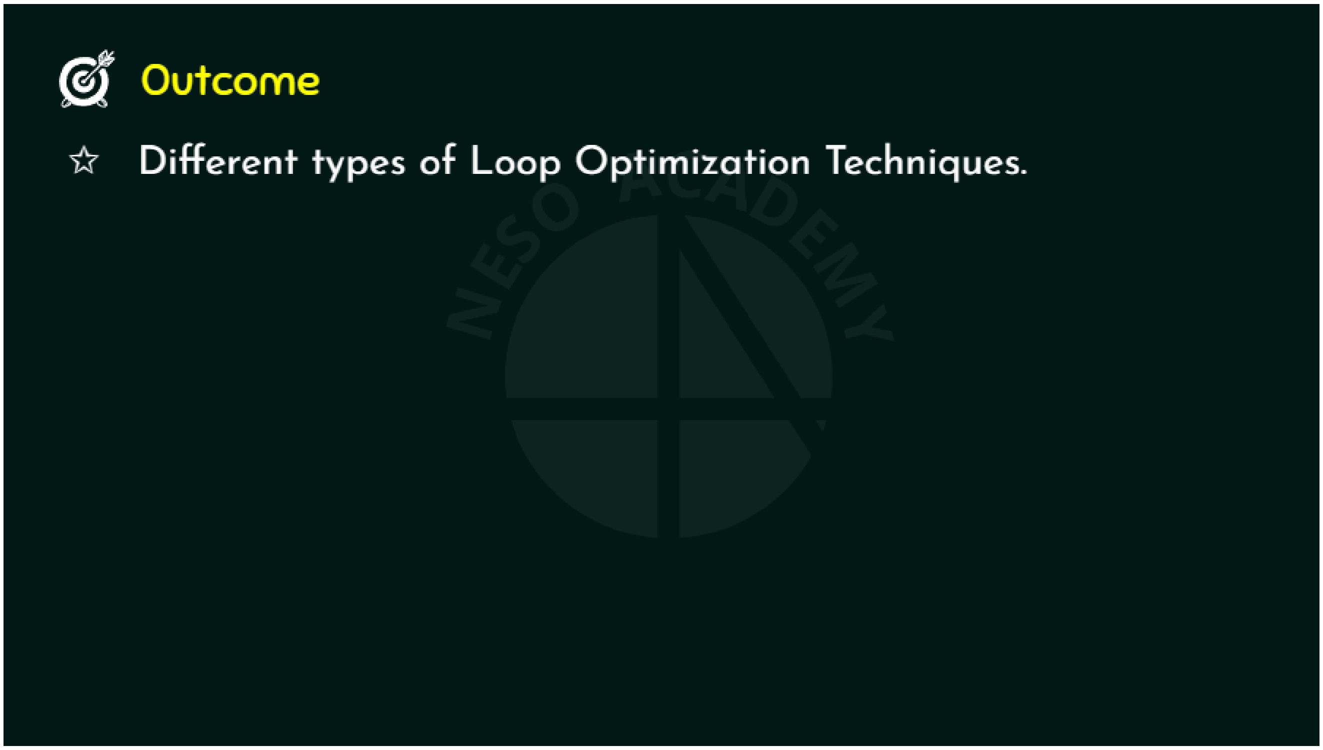


TAC:1.f = 12.i = 23.if(i<=a) GOTO 94.t1 = f * i5.f = t1 6.t2 = i + 17.i = t2 8.GOTO 39.GOTO <Calling Program>BB1BB2BB3BB4HLL code:fact(a){int f = 1;for(i=2;i<=a;i++)f = f * i;}BB1BB2BB3BB4PFG:Illustration - Producing PFG:

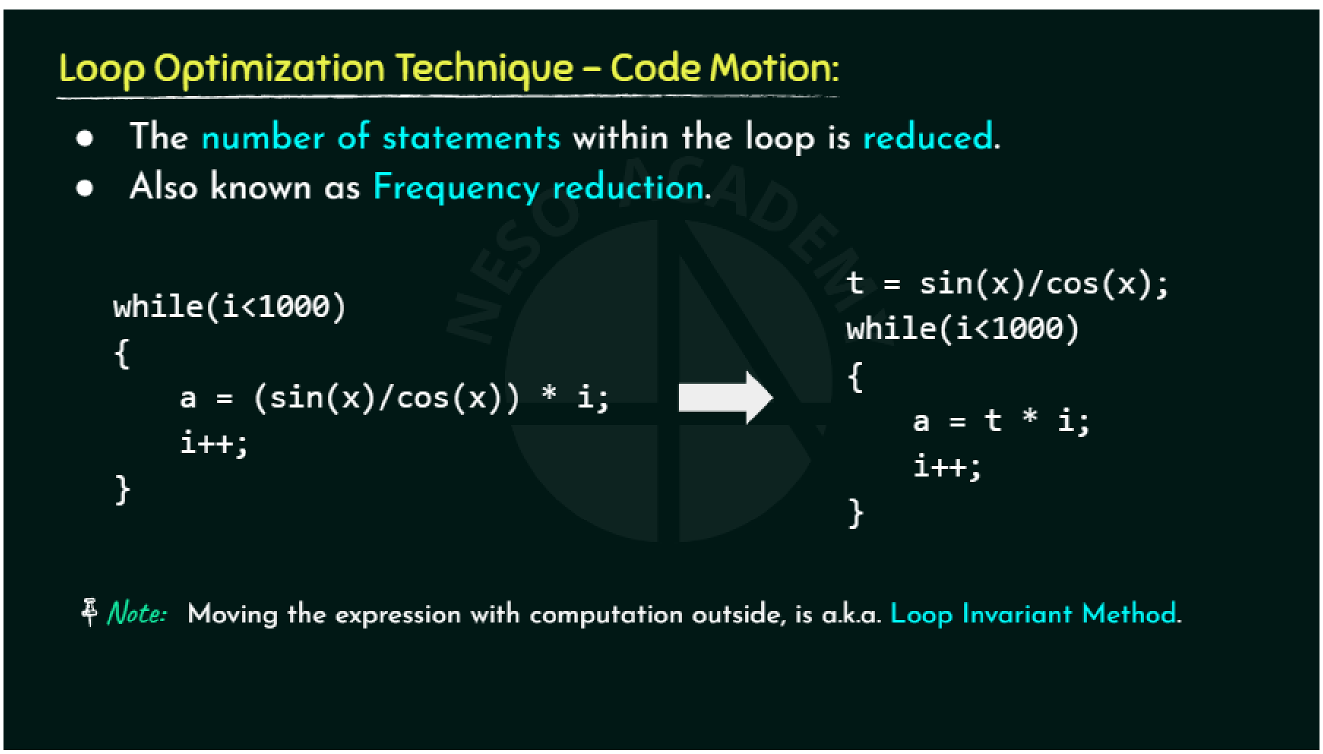


TAC:1.f = 12.i = 23.if(i<=a) GOTO 94.t1 = f * i5.f = t1 6.t2 = i + 17.i = t2 8.GOTO 39.GOTO <Calling Program>BB1BB2BB3BB4HLL code:fact(a){int f = 1;for(i=2;i<=a;i++)f = f * i;}BB1BB2BB3BB4PFG:Illustration - Performing CFA:

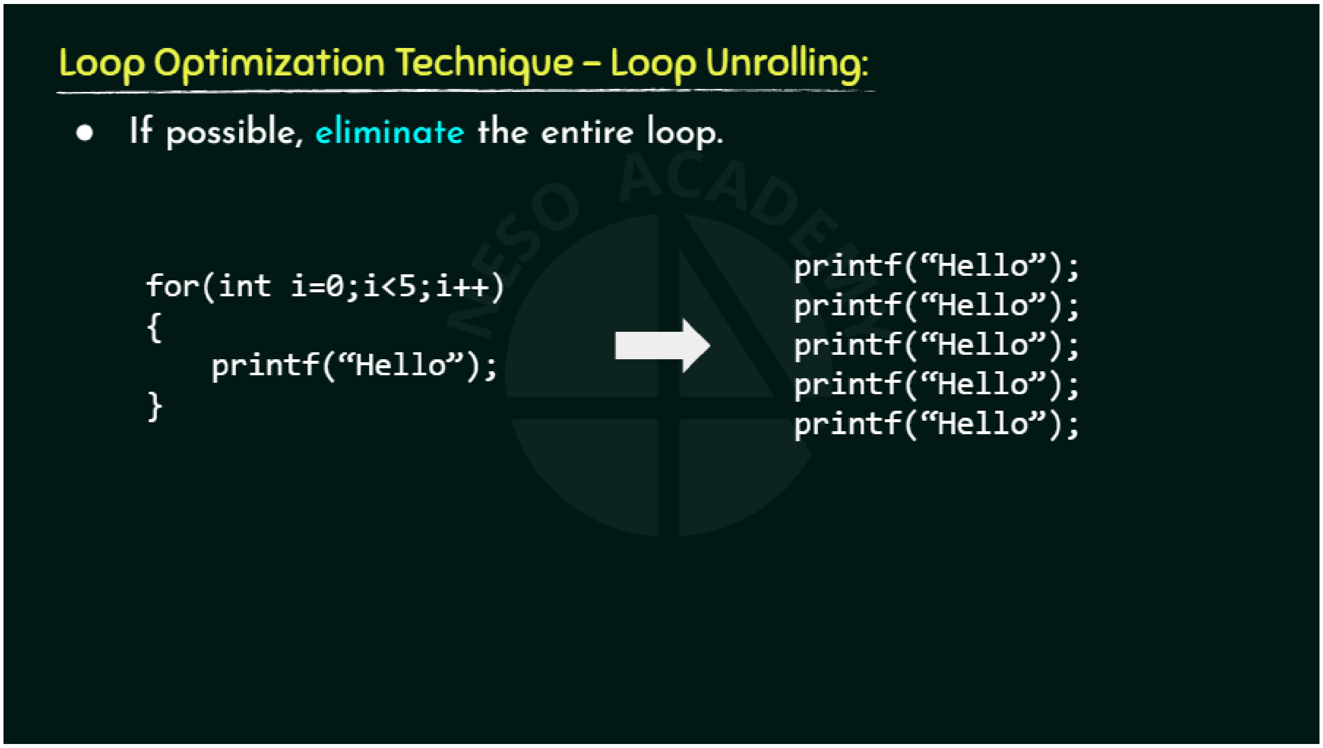




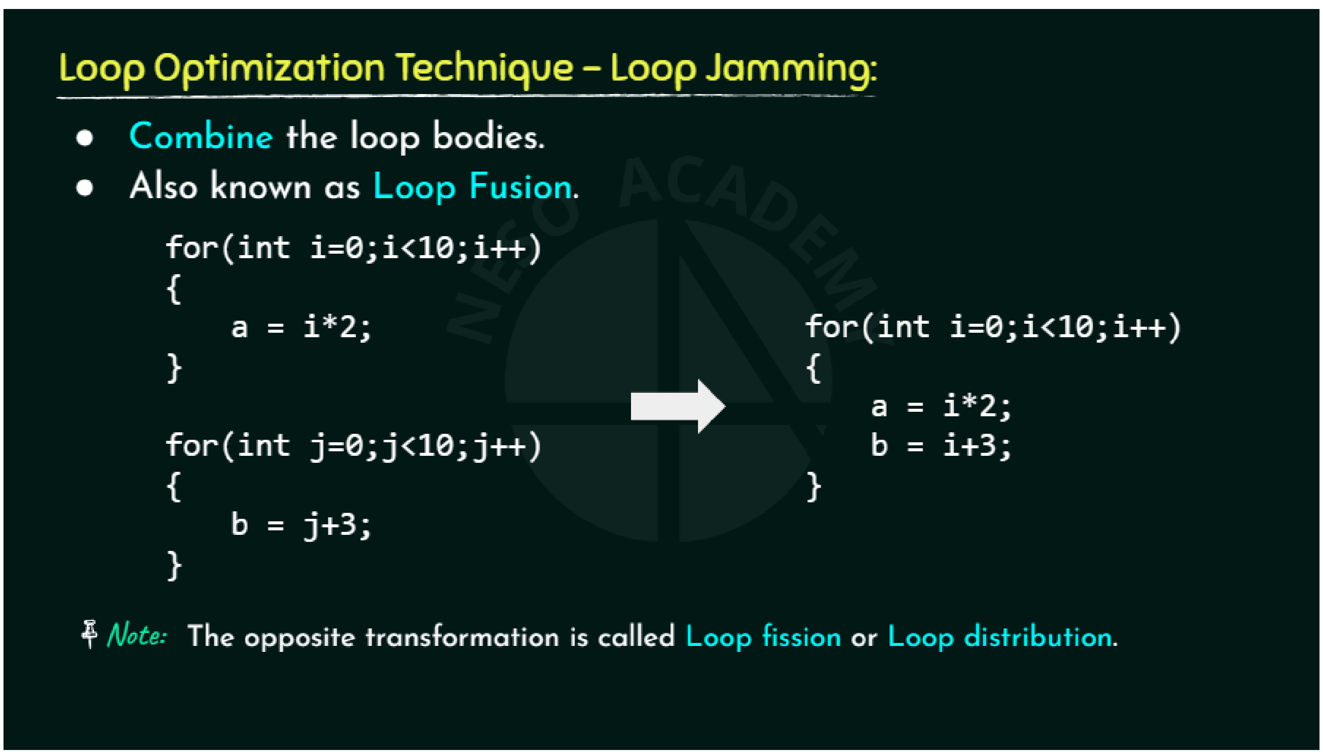
Outcome☆Different types of Loop Optimization Techniques.



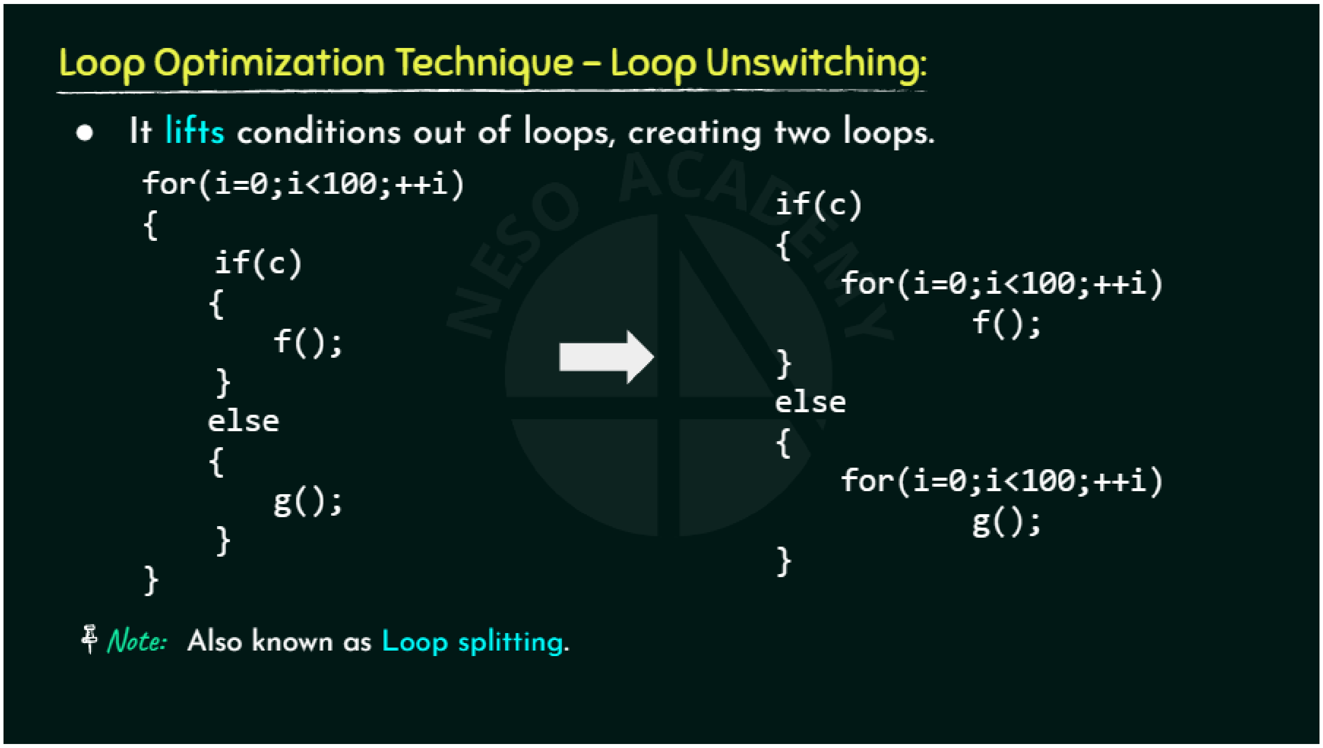
Loop Optimization Technique - Code Motion:●The number of statements within the loop is reduced.●Also known as Frequency reduction.while(i<1000){a = (sin(x)/cos(x)) * i;i++;}t = sin(x)/cos(x);Note:Moving the expression with computation outside, is a.k.a. Loop Invariant Method.while(i<1000){a = t * i;i++;}



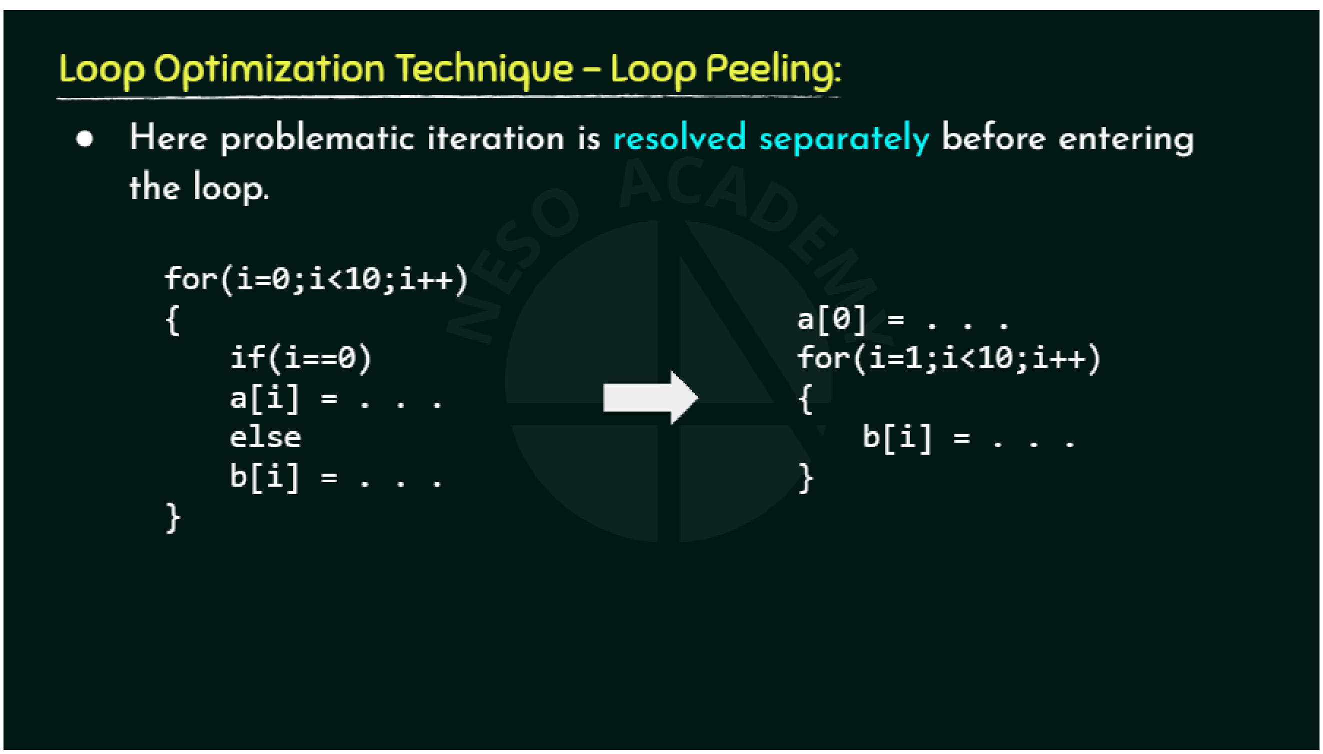
Loop Optimization Technique - Loop Unrolling:●If possible, eliminate the entire loop.for(int i=0;i<5;i++){ printf(“Hello”);}printf(“Hello”);printf(“Hello”);printf(“Hello”);printf(“Hello”);printf(“Hello”);



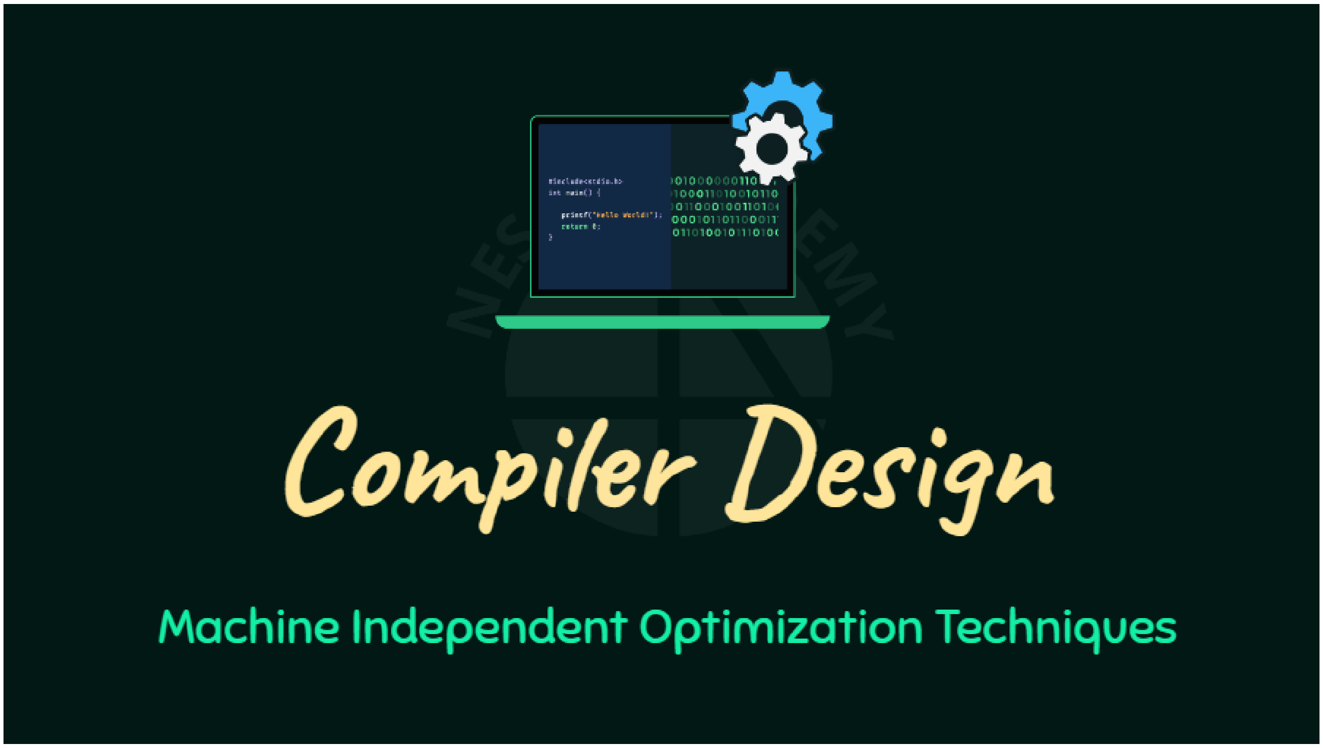
Loop Optimization Technique - Loop Jamming:●Combine the loop bodies.●Also known as Loop Fusion.for(int i=0;i<10;i++){ a = i*2;} for(int j=0;j<10;j++){ b = j+3;}for(int i=0;i<10;i++){ a = i*2;b = i+3;}Note:The opposite transformation is called Loop fission or Loop distribution.



Loop Optimization Technique - Loop Unswitching:●It lifts conditions out of loops, creating two loops.for(i=0;i<100;++i){ if(c){f(); } else { g(); }}if(c) {for(i=0;i<100;++i) f();} else {for(i=0;i<100;++i) g();}Note:Also known as Loop splitting.

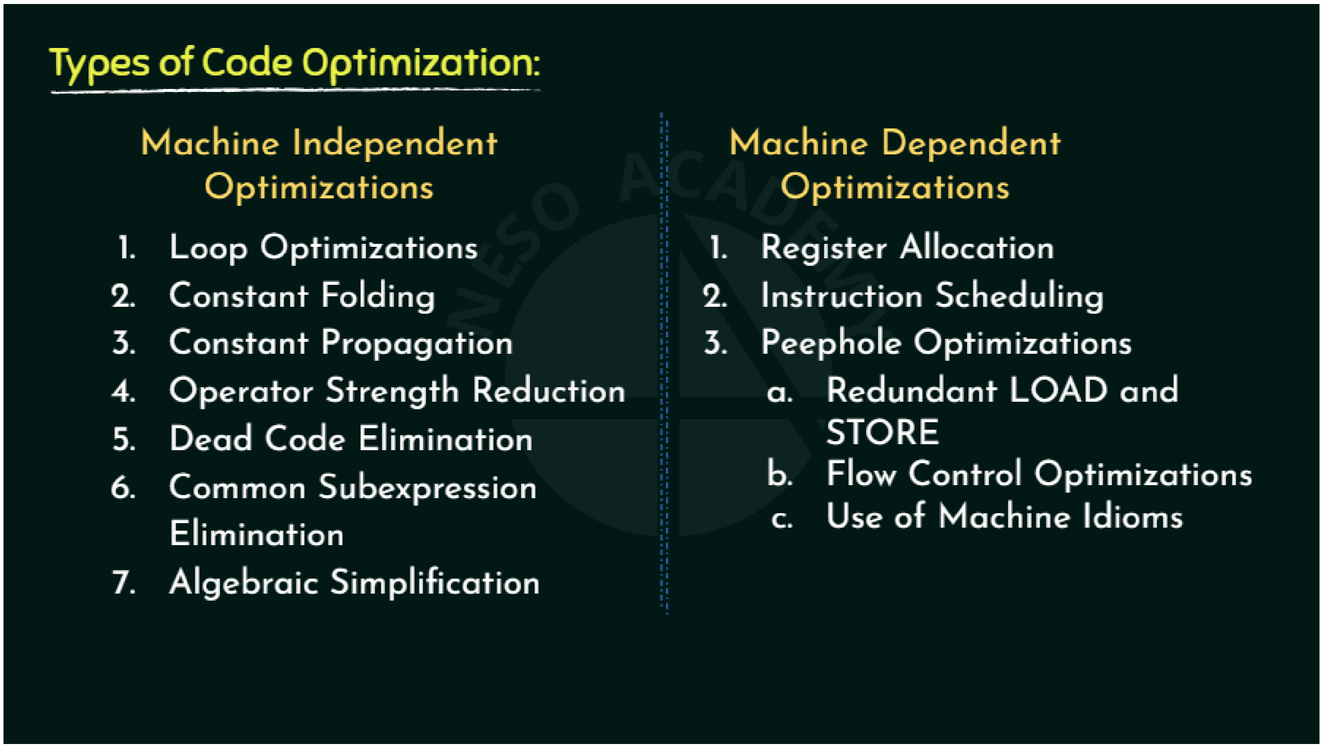


Loop Optimization Technique - Loop Peeling:●Here problematic iteration is resolved separately before entering the loop.for(i=0;i<10;i++){if(i==0)a[i] = . . .elseb[i] = . . .}a[0] = . . .for(i=1;i<10;i++){b[i] = . . .}

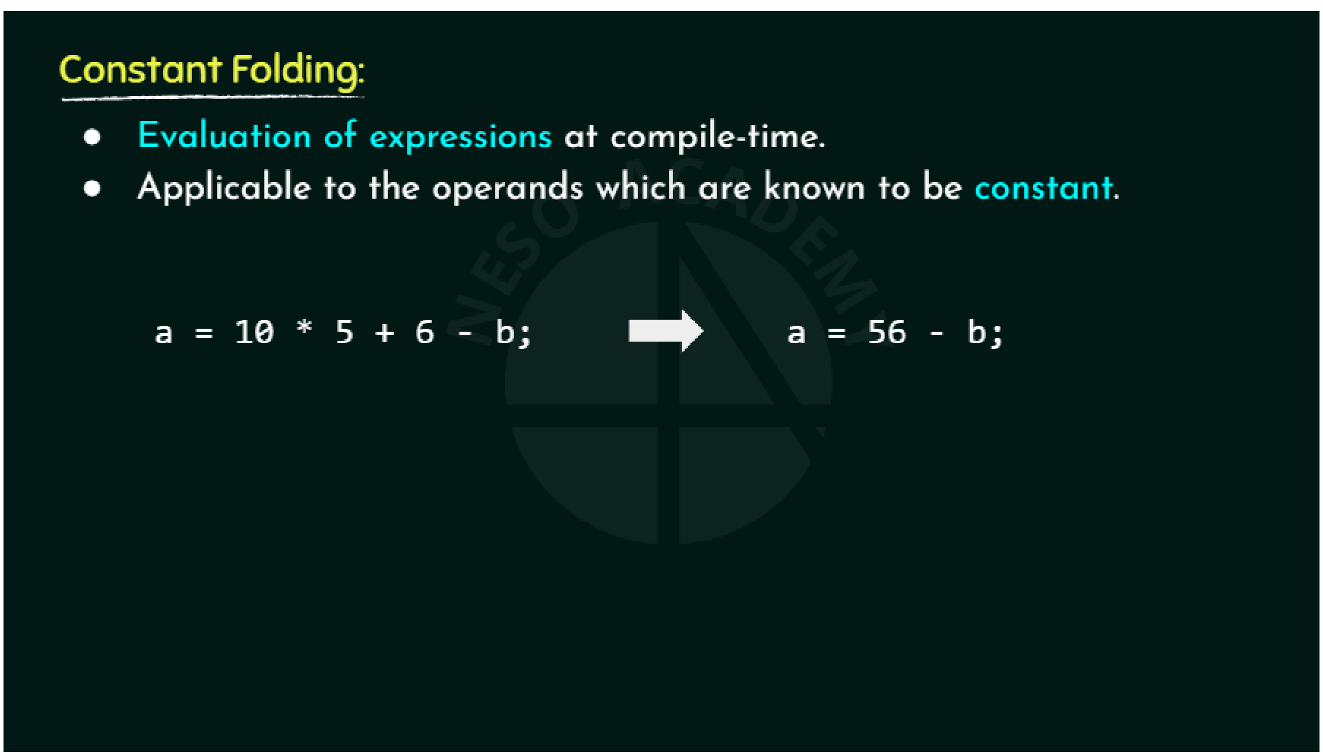


Compiler Design Machine Independent Optimization Techniques

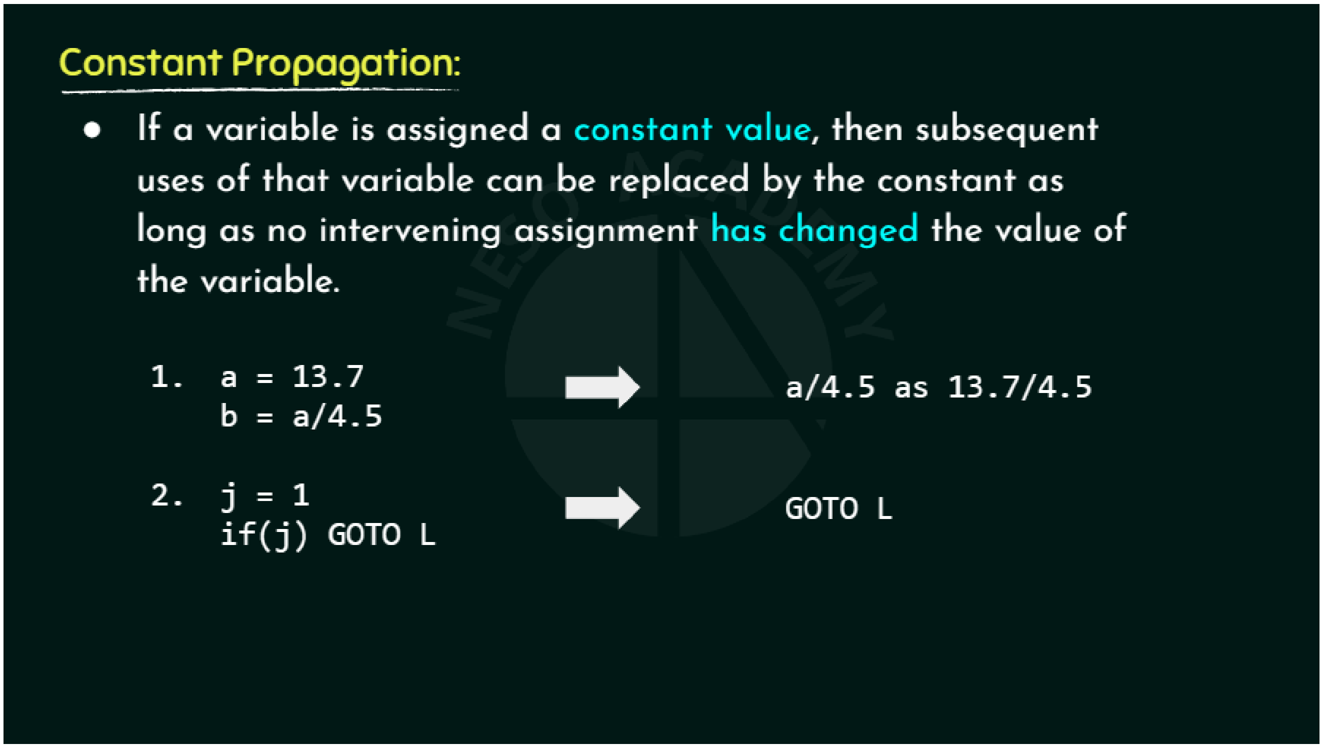
Outcome☆Different Machine Independent Optimization Techniques.



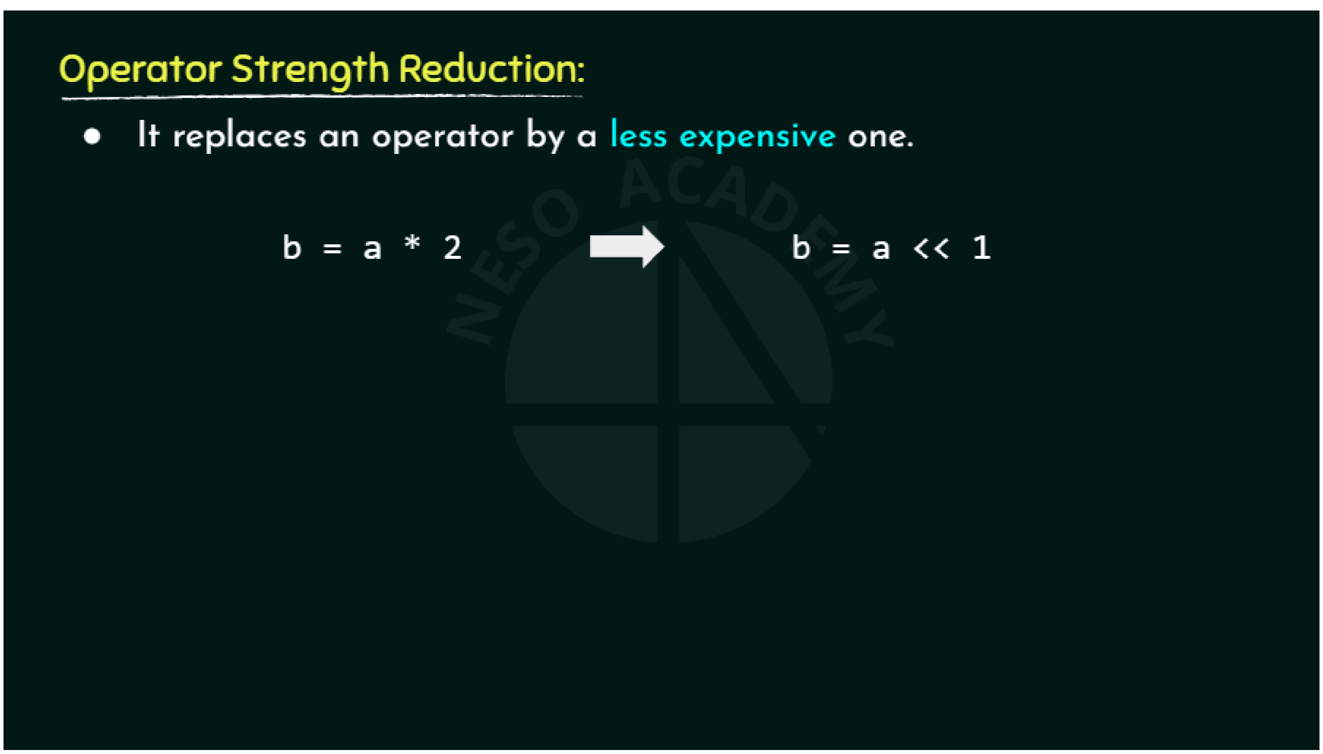
Types of Code Optimization:Machine Independent OptimizationsMachine Dependent Optimizations1.Register Allocation2.Instruction Scheduling 3.Peephole Optimizations a.Redundant LOAD and STOREb.Flow Control Optimizationsc.Use of Machine Idioms1.Loop Optimizations2.Constant Folding3.Constant Propagation4.Operator Strength Reduction5.Dead Code Elimination 6.Common Subexpression Elimination7.Algebraic Simplification



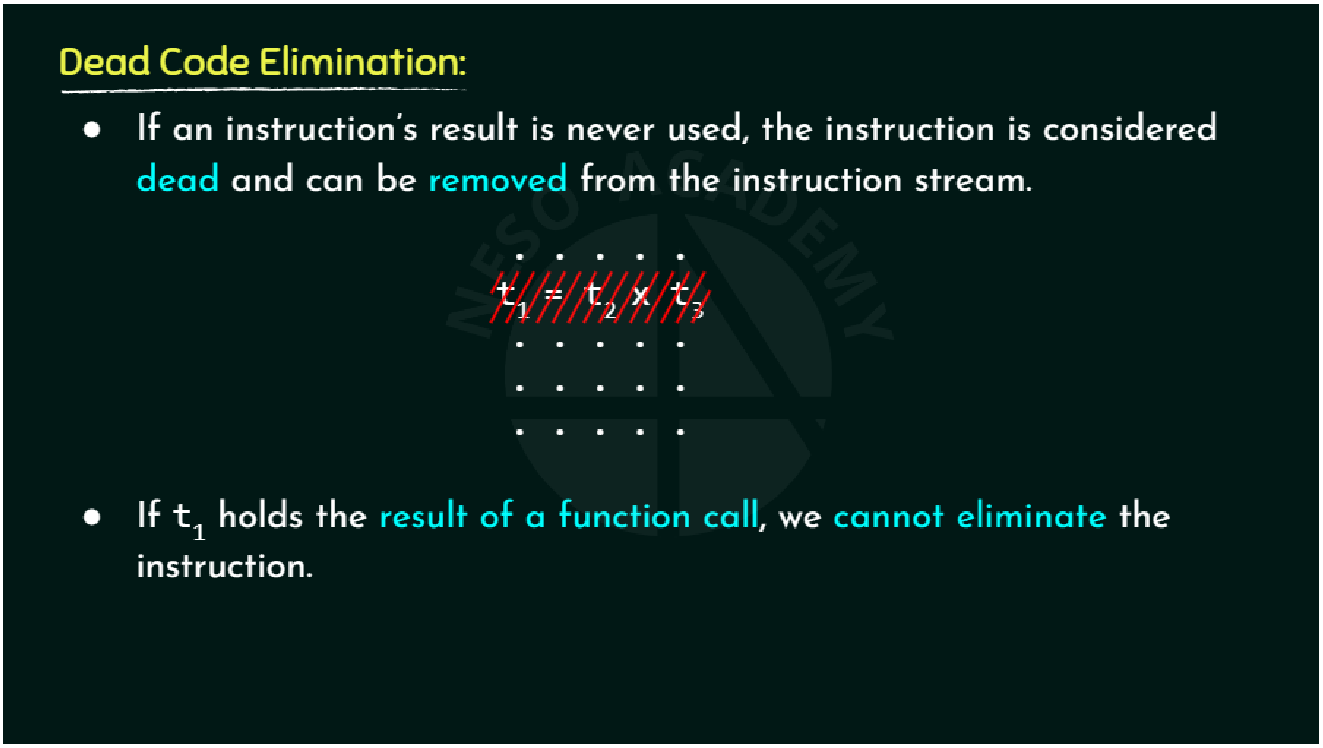
Constant Folding:●Evaluation of expressions at compile-time.●Applicable to the operands which are known to be constant. a = 10 * 5 + 6 - b; a = 56 - b;



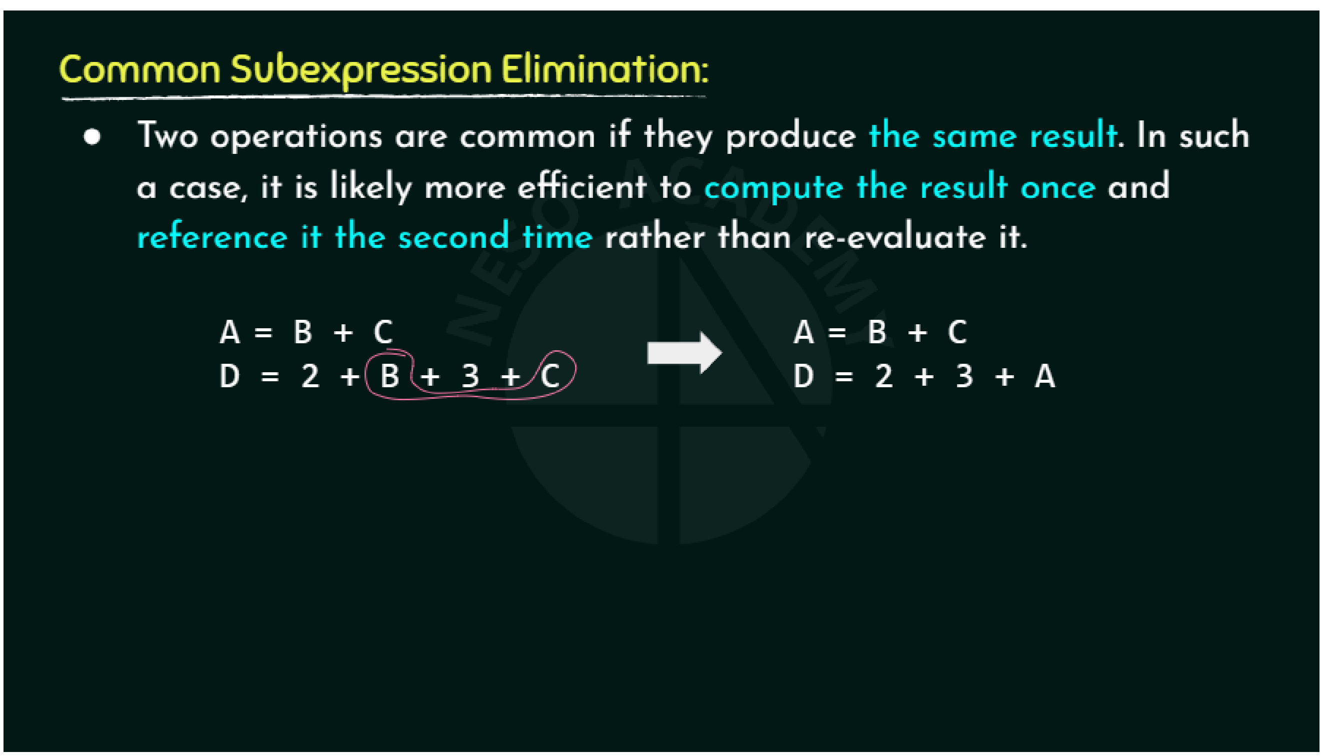
Constant Propagation:●If a variable is assigned a constant value, then subsequent uses of that variable can be replaced by the constant as long as no intervening assignment has changed the value of the variable. 1.a = 13.7b = a/4.52.j = 1if(j) GOTO La/4.5 as 13.7/4.5GOTO L



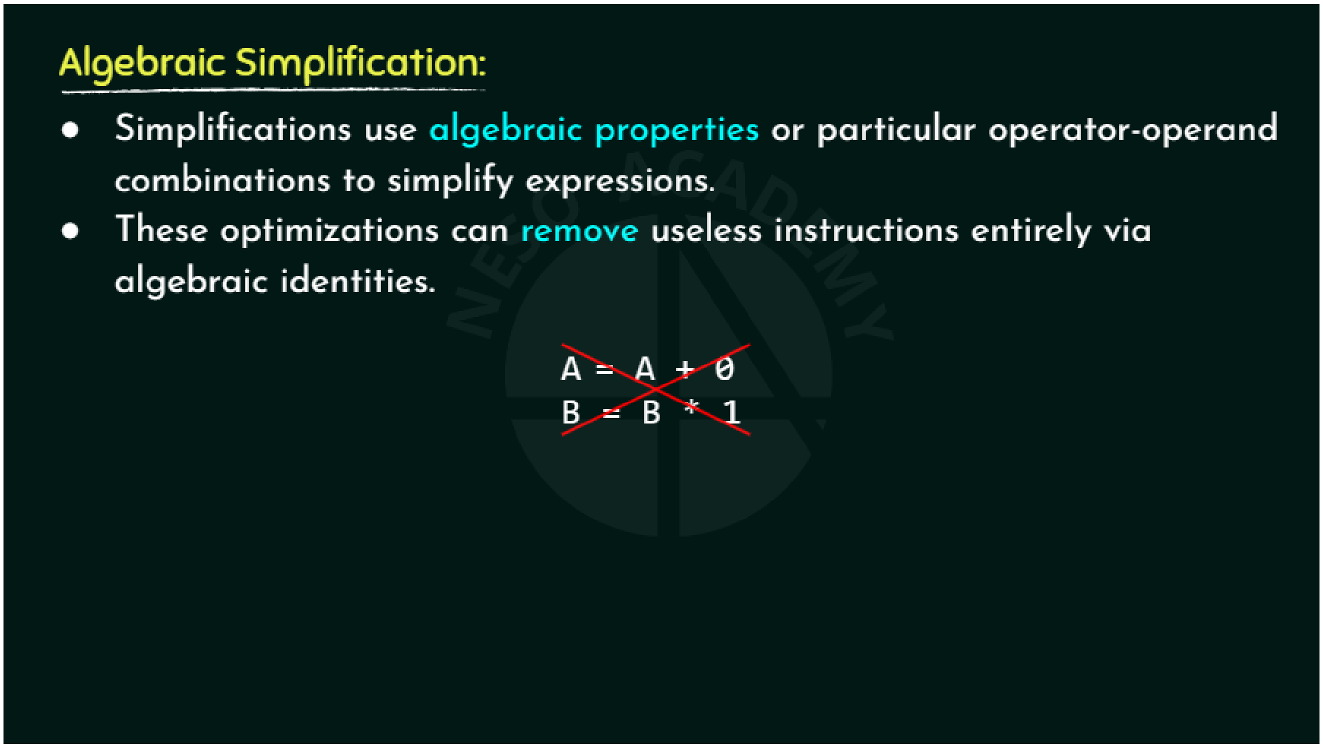
Operator Strength Reduction:●It replaces an operator by a less expensive one. b = a * 2b = a << 1



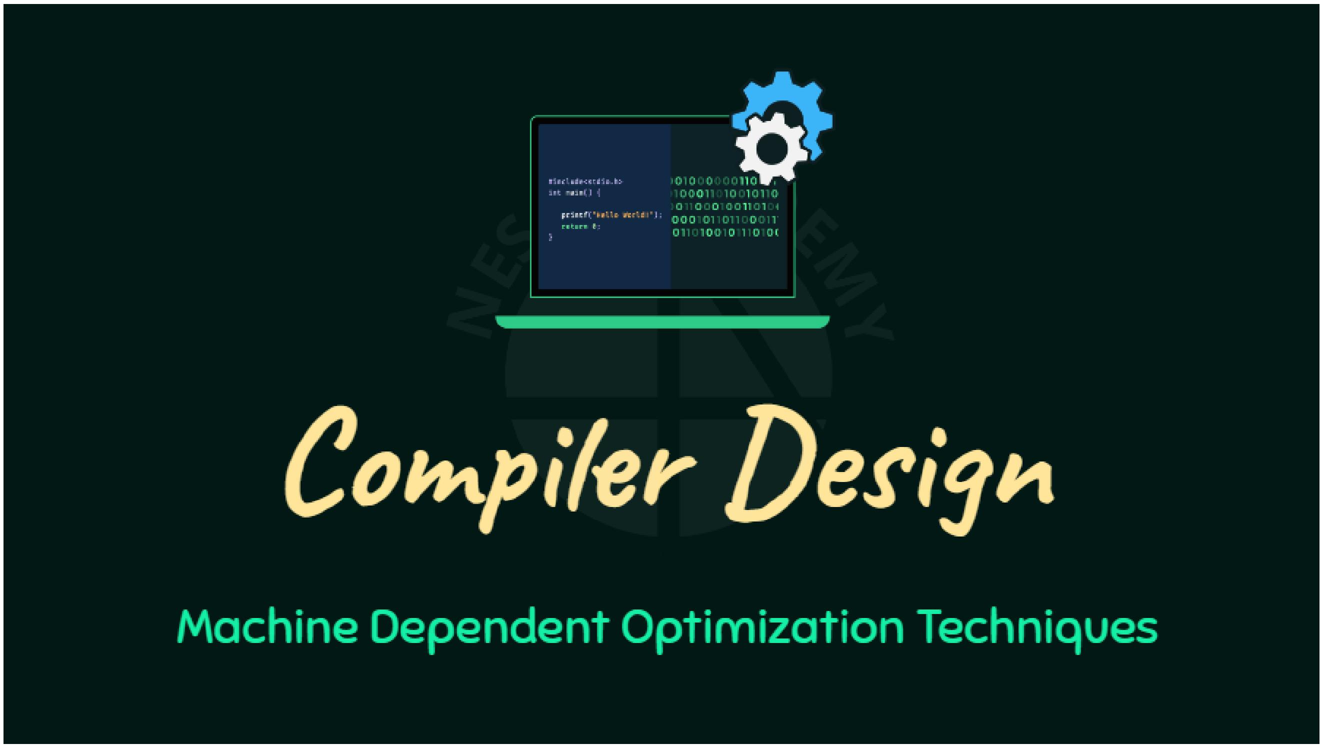
t1 = t2 x t3. . . . . .Dead Code Elimination:●If an instruction's result is never used, the instruction is considered dead and can be removed from the instruction stream. ●If t1 holds the result of a function call, we cannot eliminate the instruction.



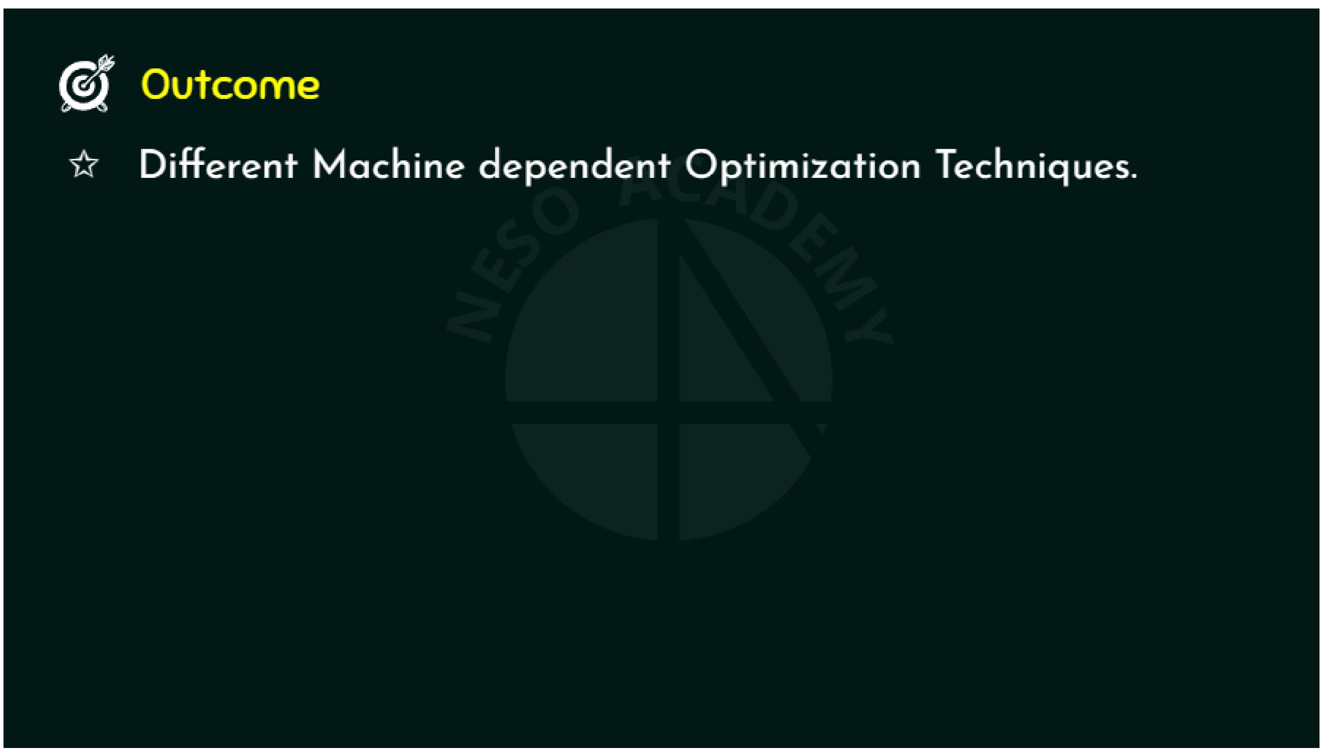
A = B + CD = 2 + B + 3 + CCommon Subexpression Elimination:●Two operations are common if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it. A = B + CD = 2 + 3 + A



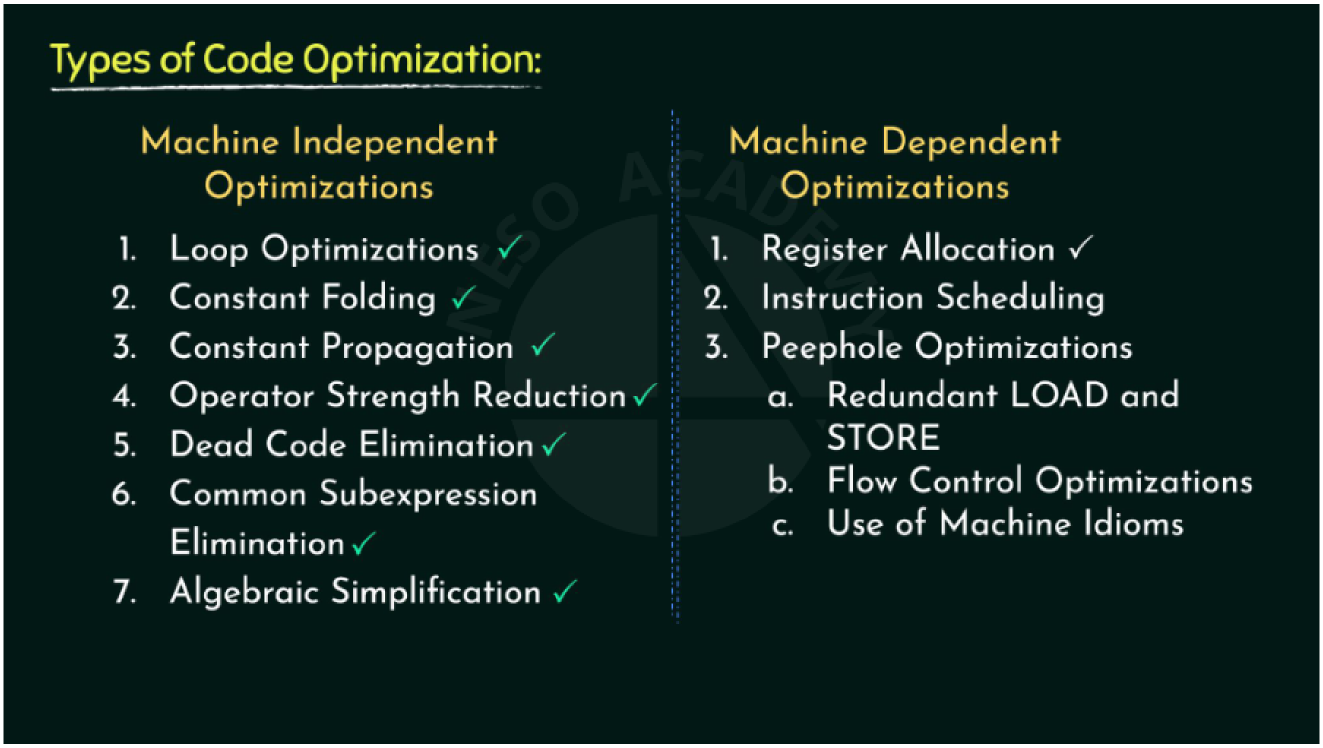
●Simplifications use algebraic properties or particular operator-operand combinations to simplify expressions. ●These optimizations can remove useless instructions entirely via algebraic identities.A = A + 0B = B * 1Algebraic Simplification:

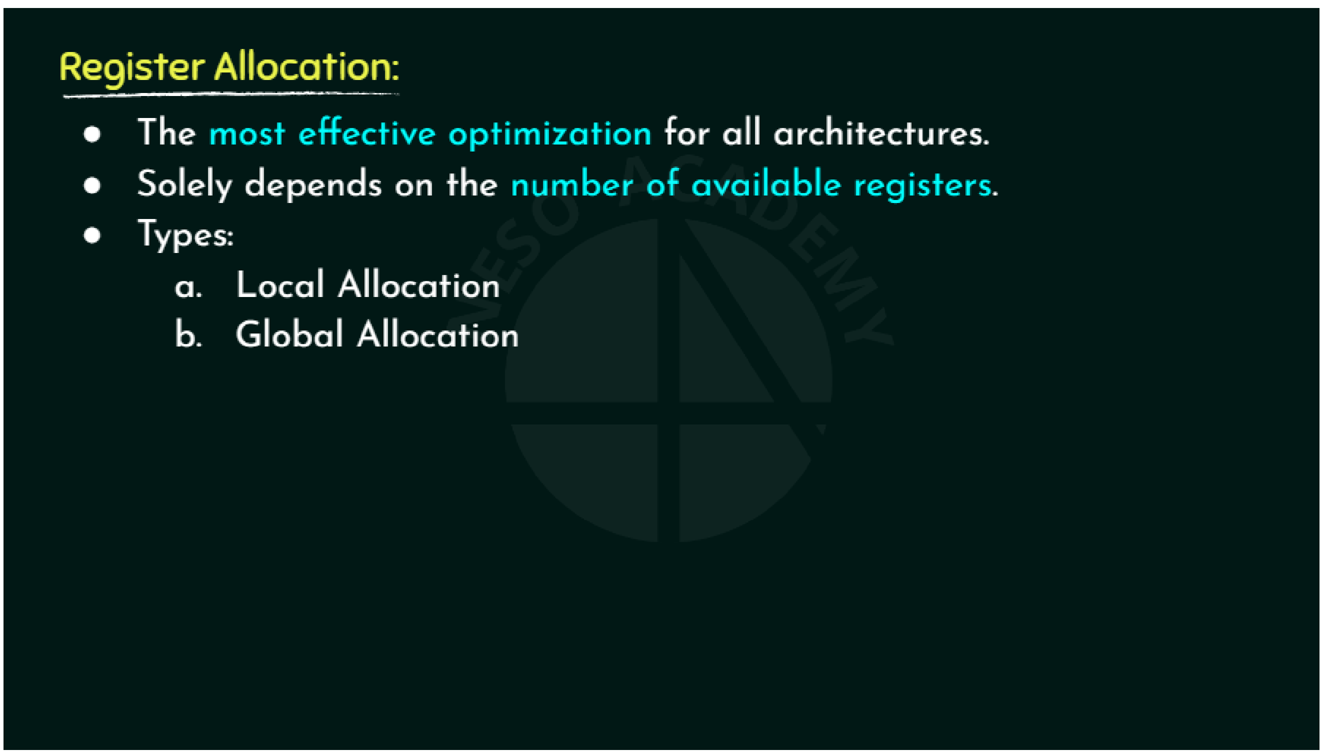


Compiler Design Machine Dependent Optimization Techniques

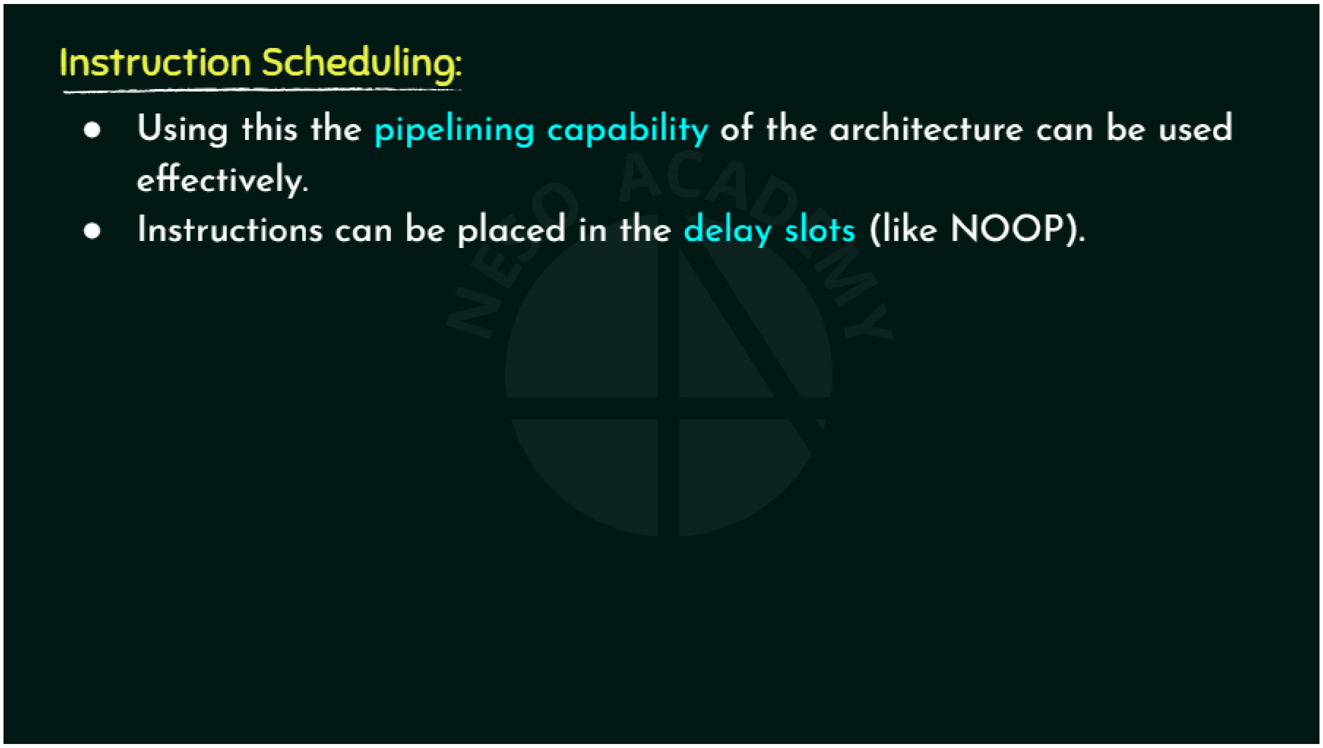


Outcome☆Different Machine dependent Optimization Techniques.

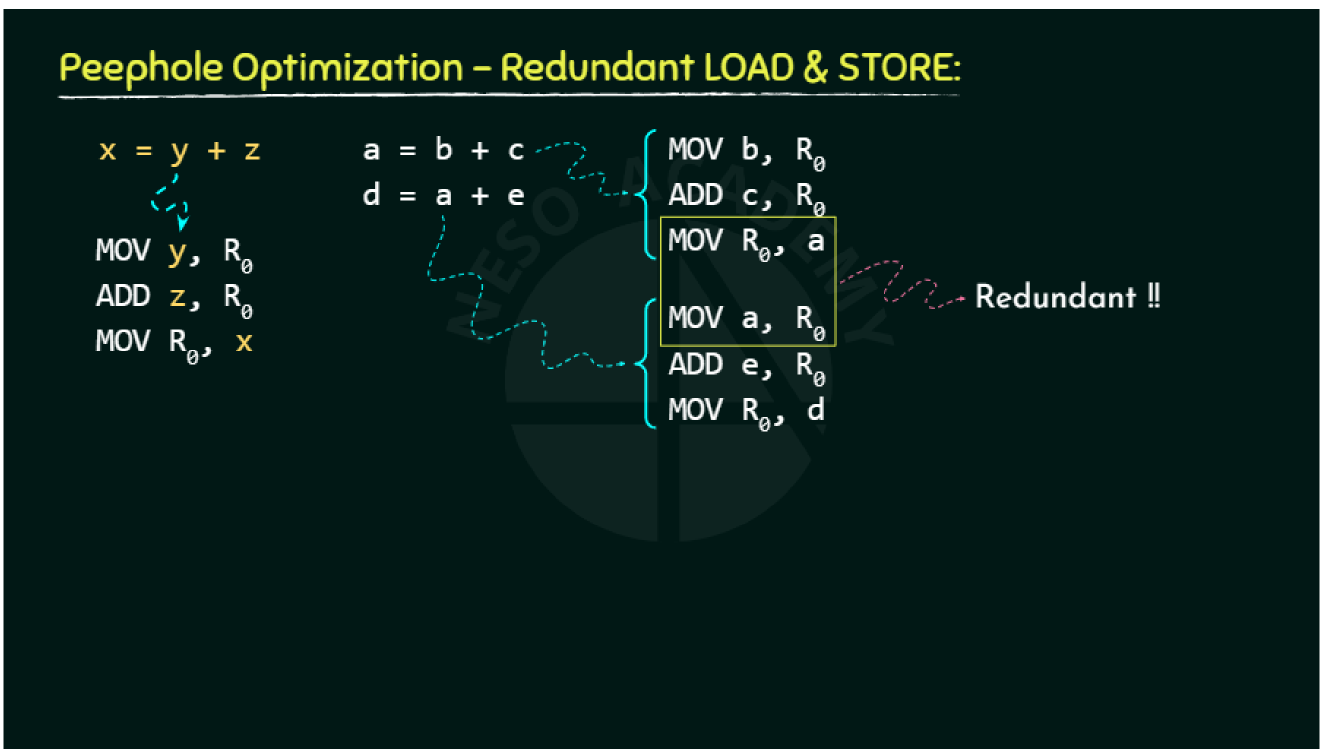




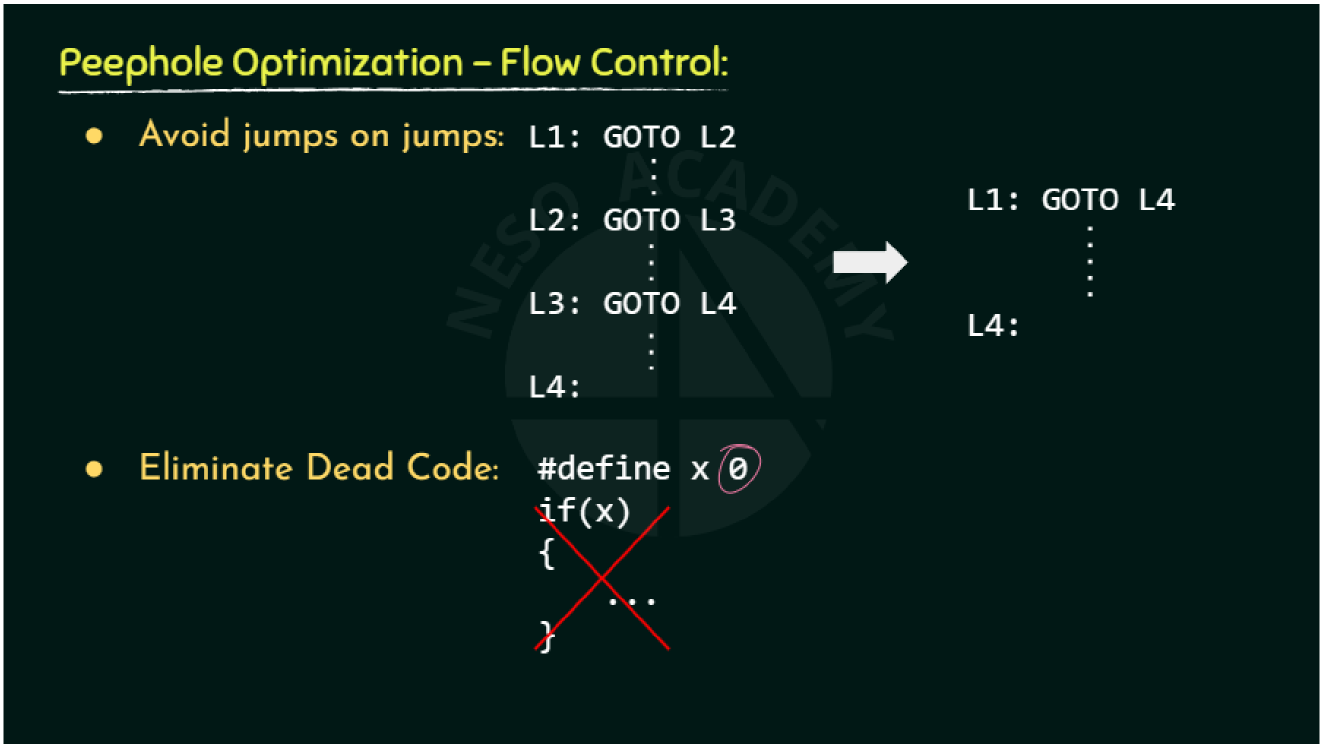
Register Allocation:●The most effective optimization for all architectures.●Solely depends on the number of available registers.●Types:a.Local Allocationb.Global Allocation



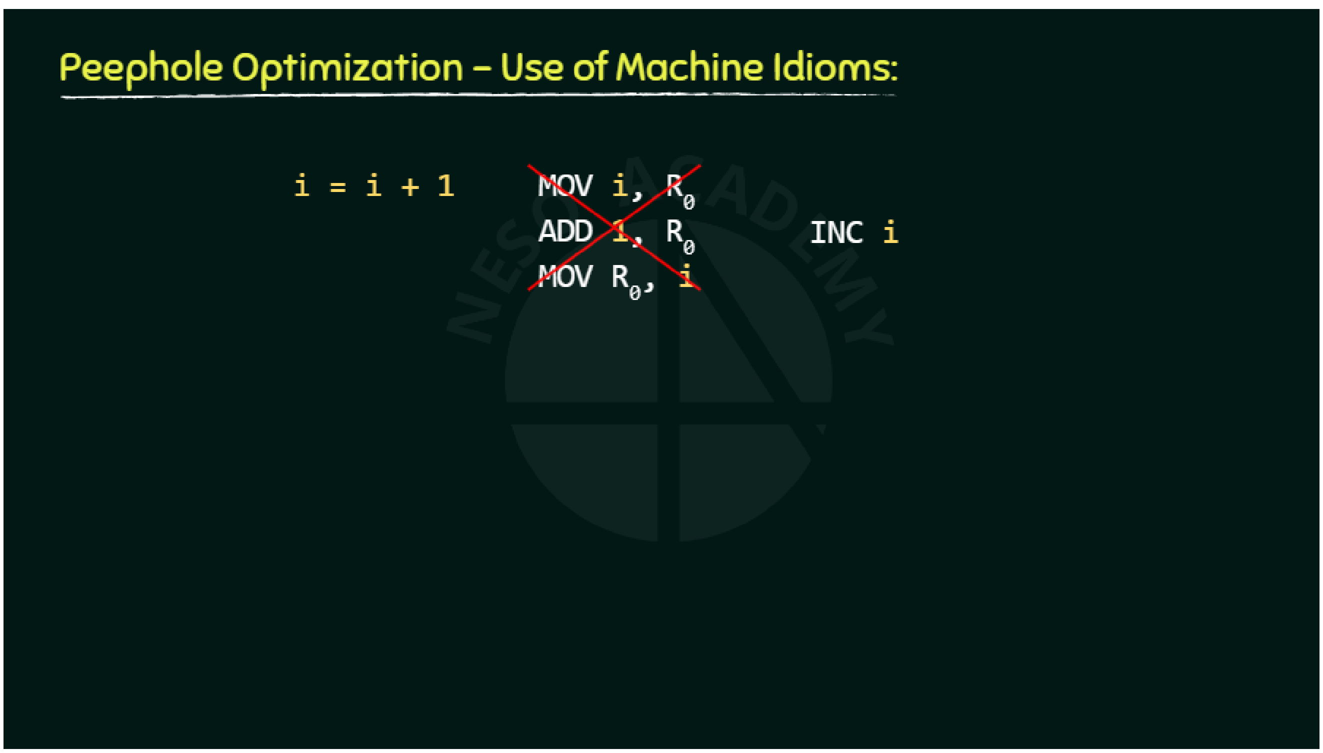
Instruction Scheduling:●Using this the pipelining capability of the architecture can be used effectively. ●Instructions can be placed in the delay slots (like NOOP).



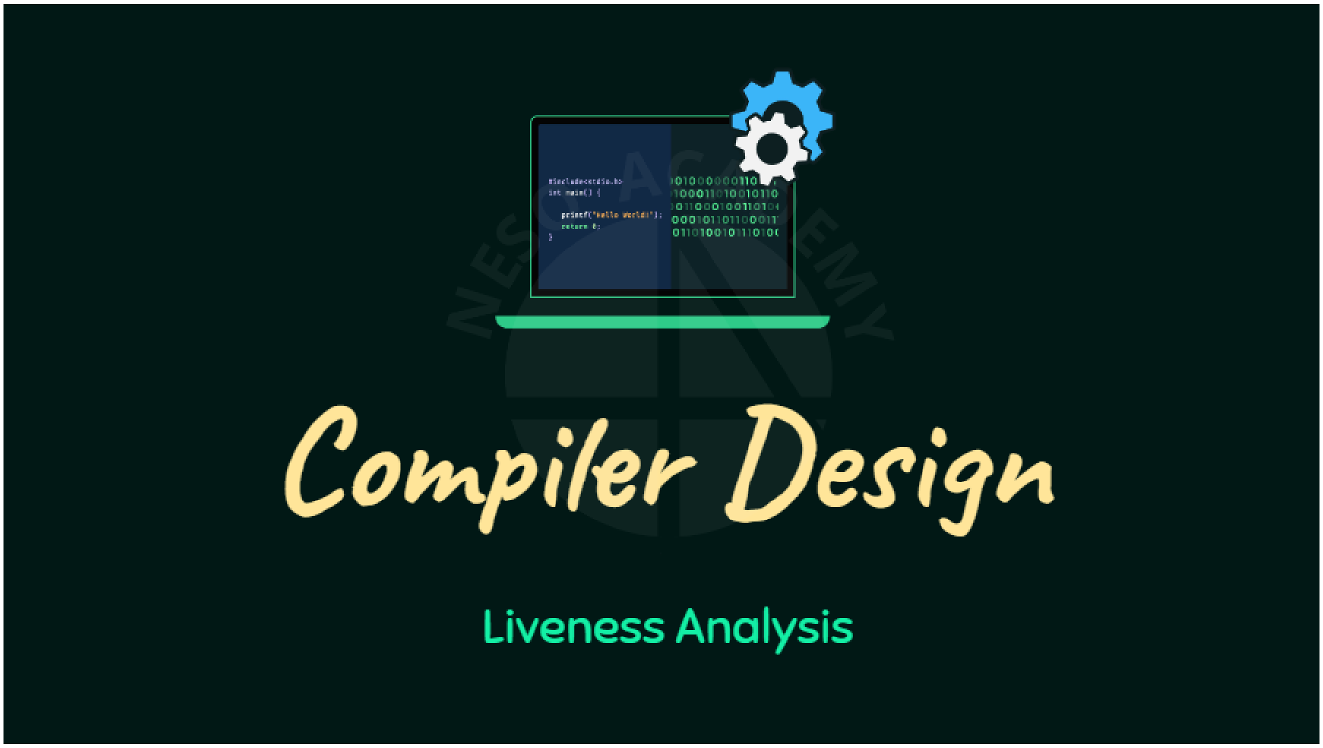
Peephole Optimization - Redundant LOAD & STORE:x = y + zMOV y, R0ADD z, R0MOV R0, xa = b + cd = a + eMOV b, R0ADD c, R0MOV R0, aMOV a, R0ADD e, R0MOV R0, dRedundant !!



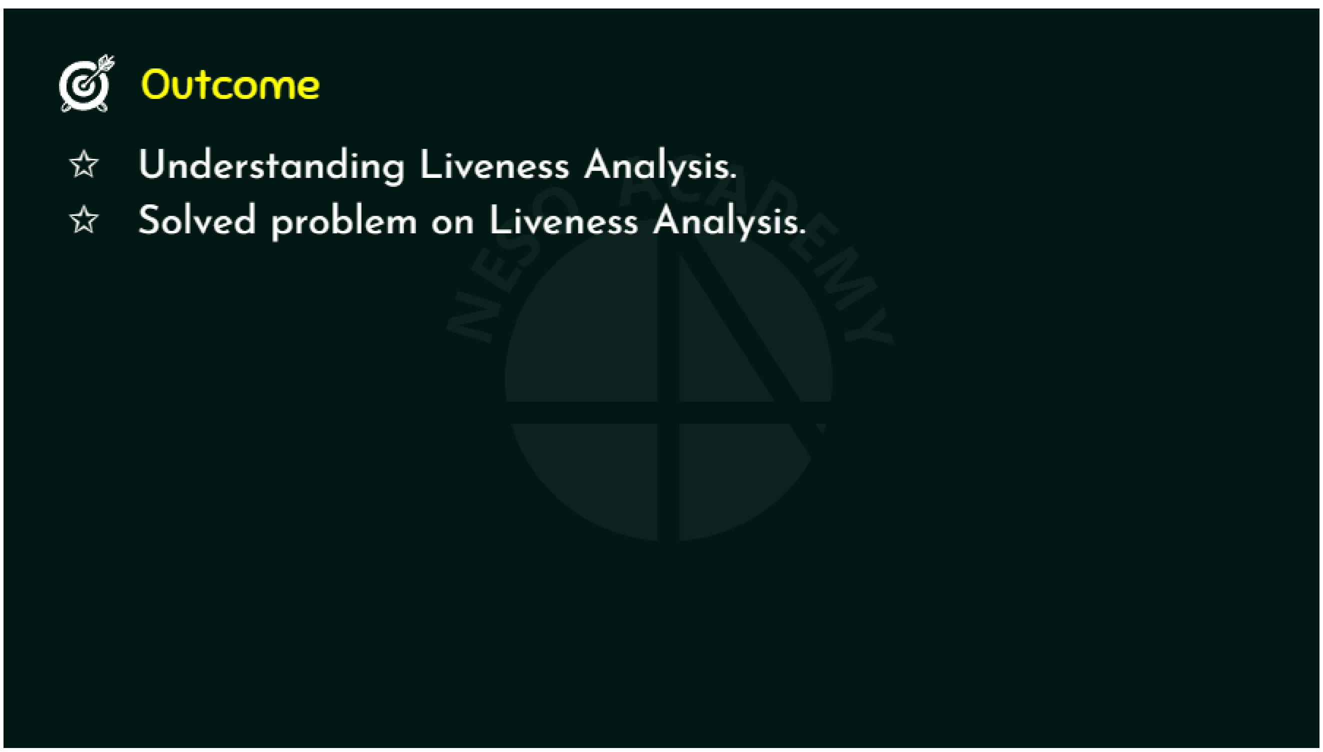
Peephole Optimization - Flow Control:●Avoid jumps on jumps:L1: GOTO L2L2: GOTO L3L3: GOTO L4L4:L1: GOTO L4L4:●Eliminate Dead Code:#define x 0if(x){...}



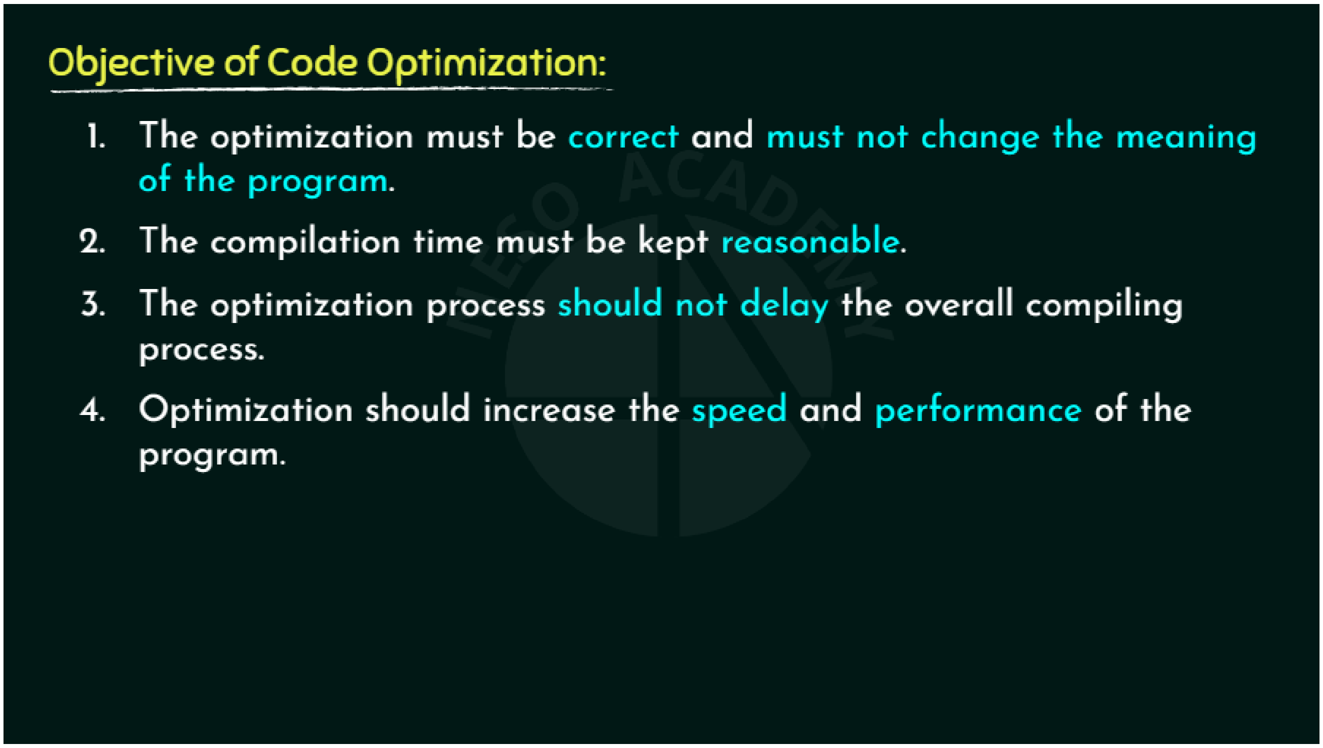
Peephole Optimization - Use of Machine Idioms:i = i + 1MOV i, R0ADD 1, R0MOV R0, iINC i



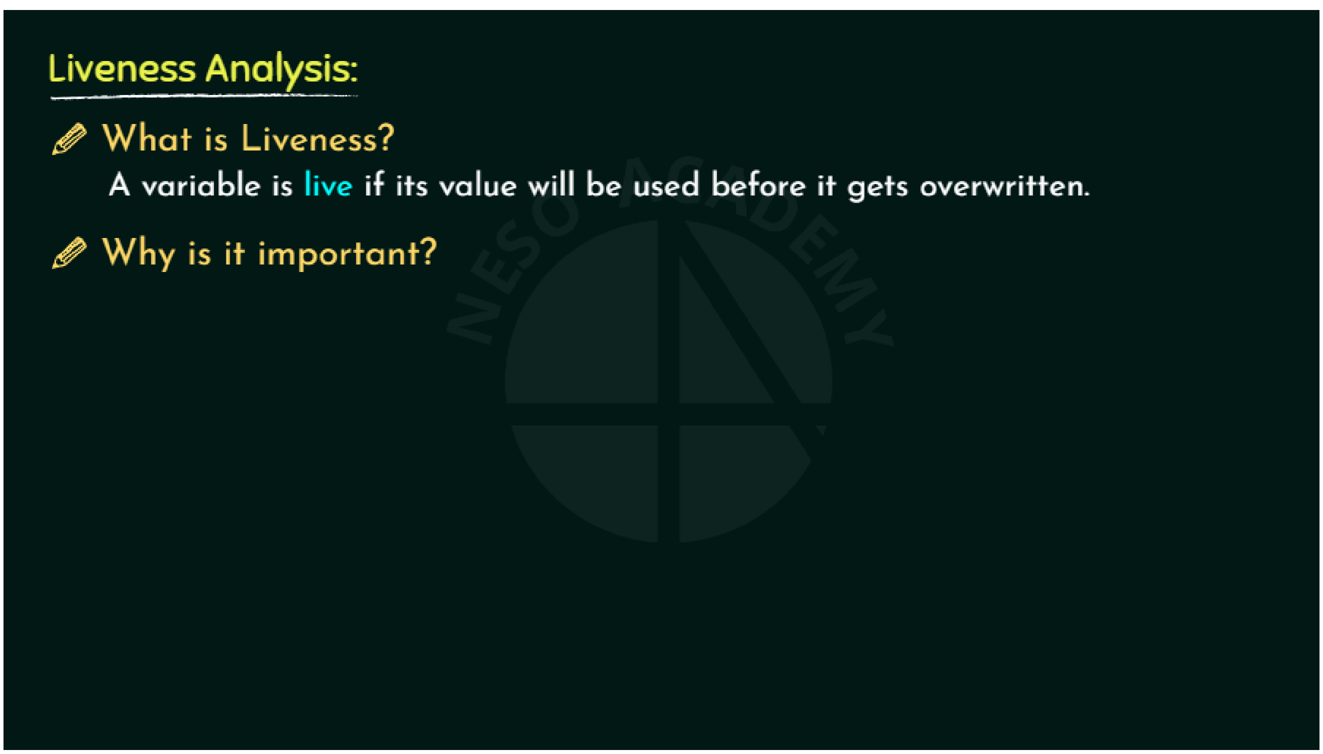
Compiler Design Liveness Analysis



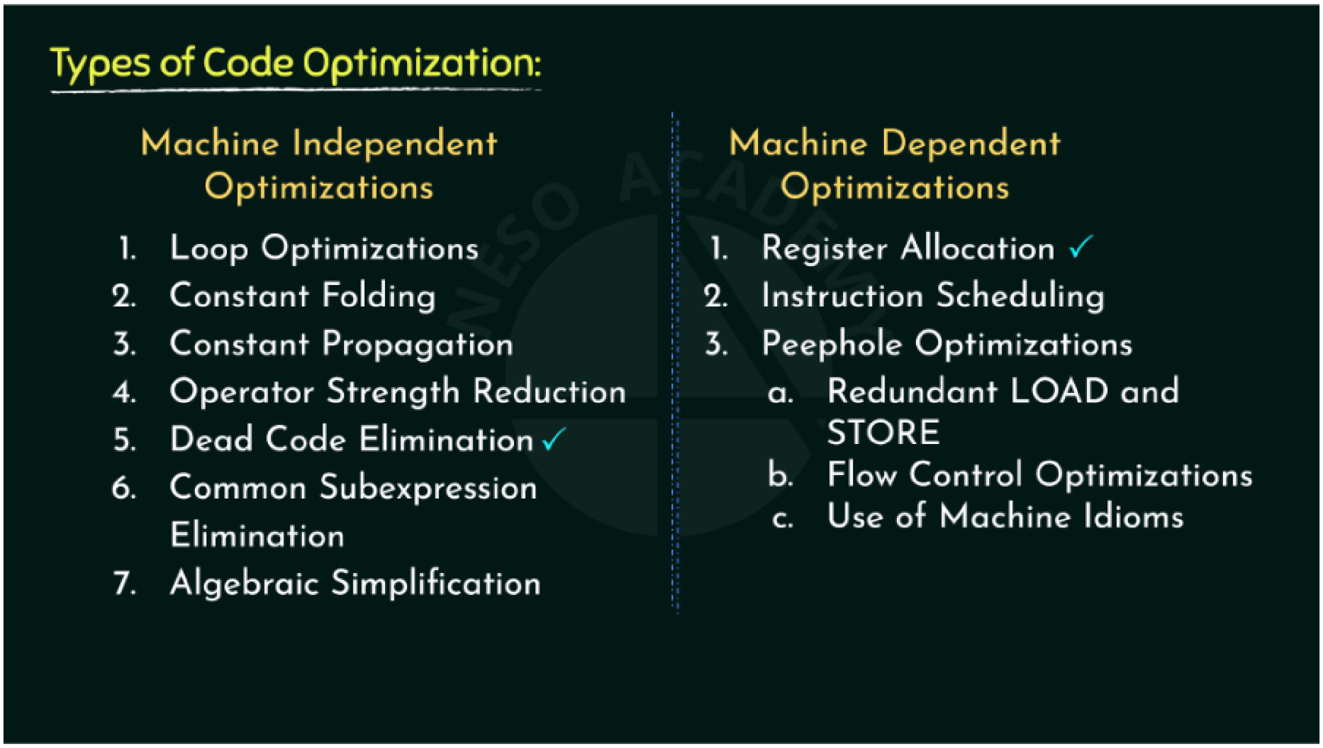
Outcome☆Understanding Liveness Analysis.☆Solved problem on Liveness Analysis.

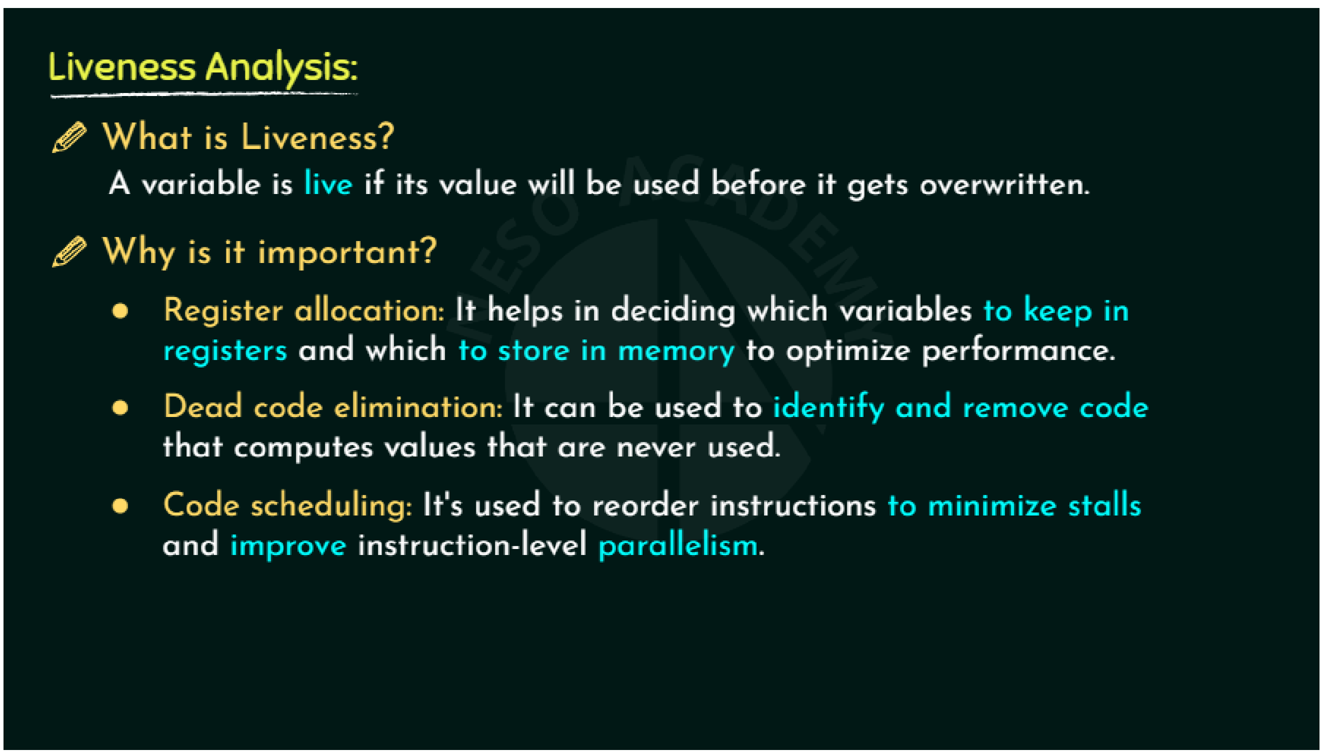


Objective of Code Optimization:1.The optimization must be correct and must not change the meaning of the program.2.The compilation time must be kept reasonable.3.The optimization process should not delay the overall compiling process.4.Optimization should increase the speed and performance of the program.

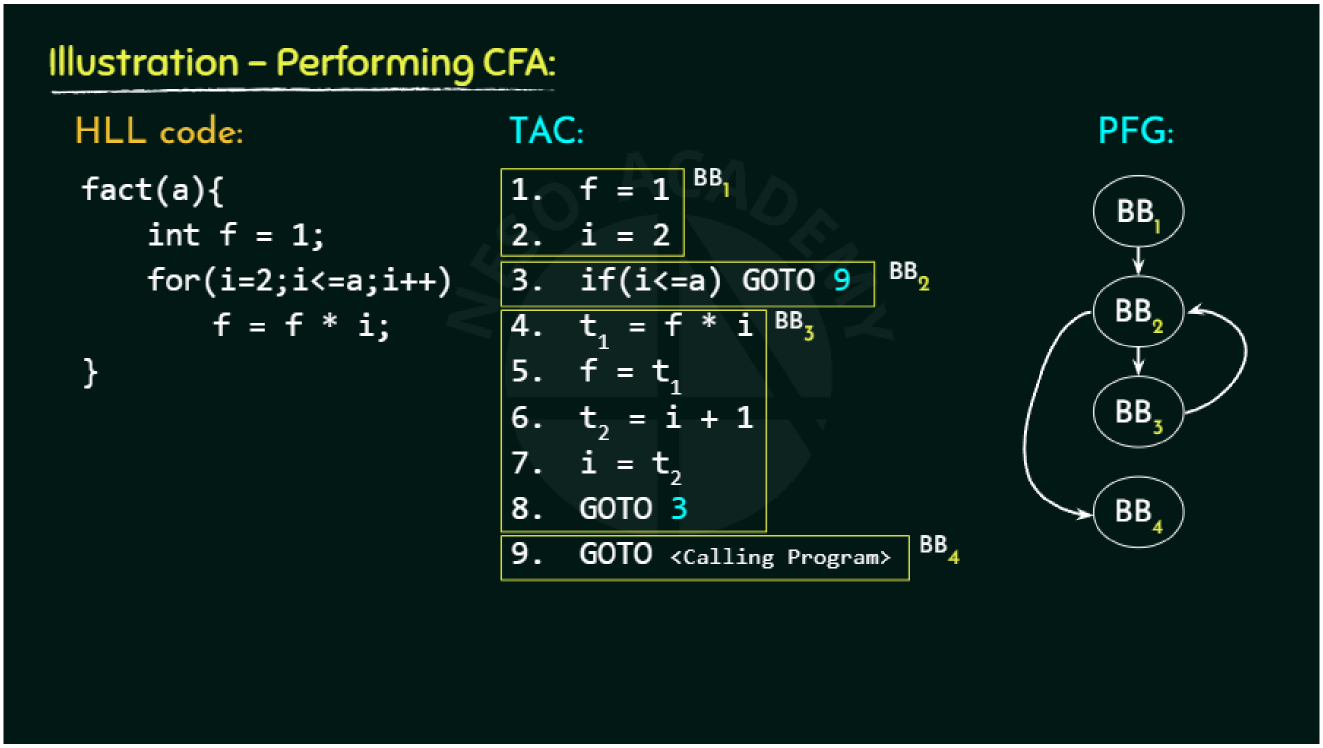


Liveness Analysis:What is Liveness? A variable is live if its value will be used before it gets overwritten.Why is it important?

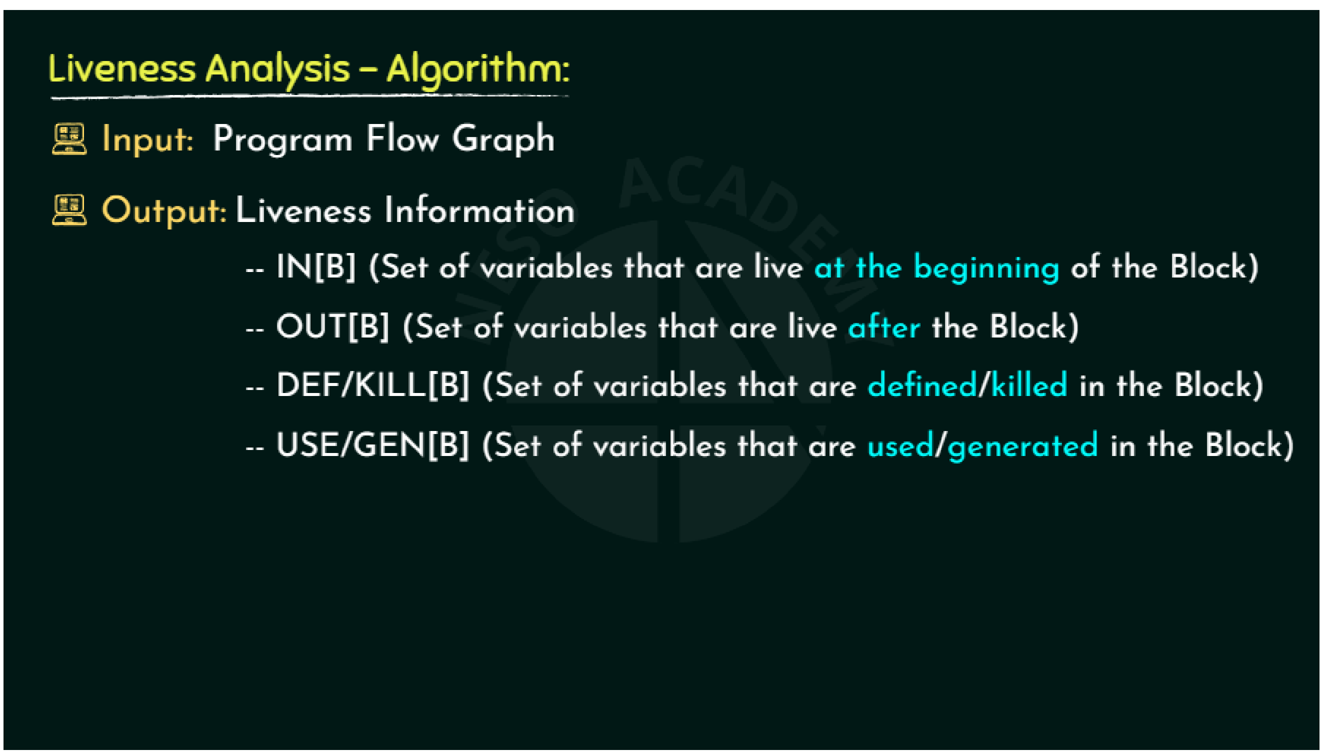




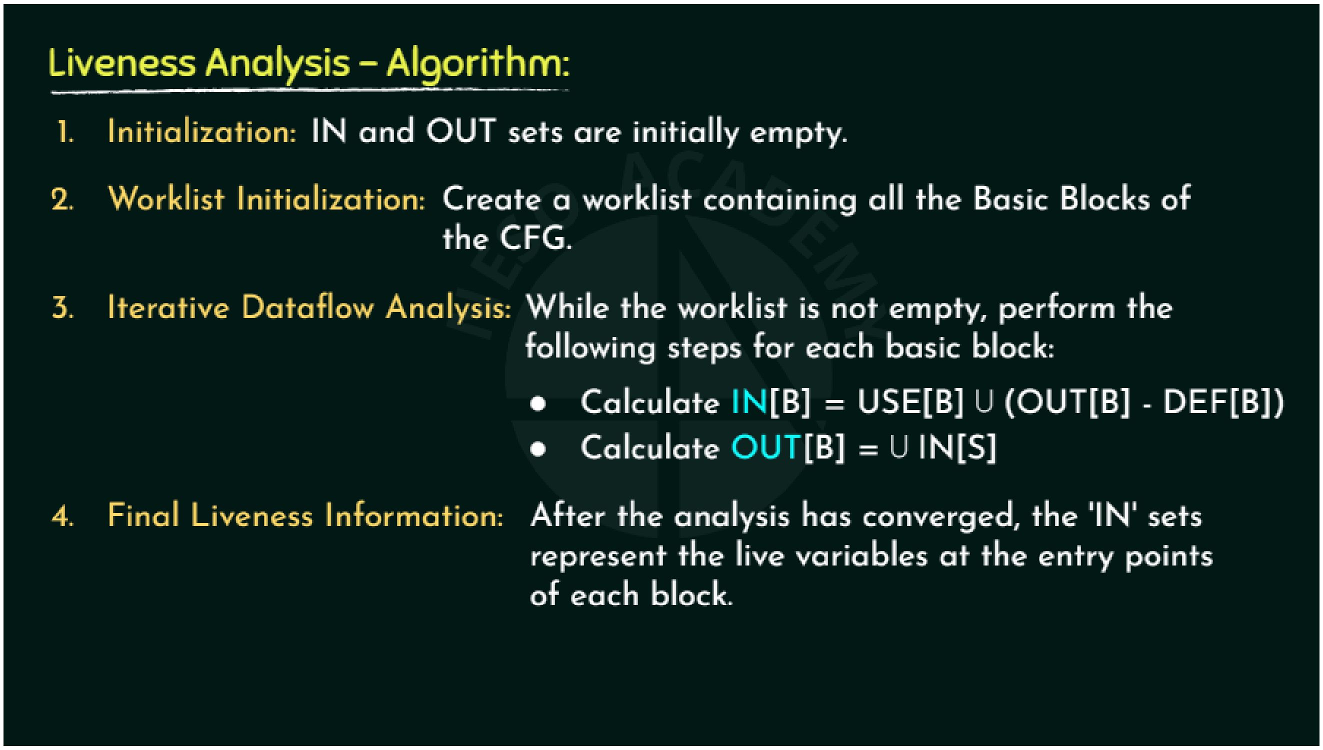
Liveness Analysis:What is Liveness? A variable is live if its value will be used before it gets overwritten.Why is it important?●Code scheduling:●Register allocation: It helps in deciding which variables to keep inregisters and which to store in memory to optimize performance.●Dead code elimination: It can be used to identify and remove codethat computes values that are never used.It's used to reorder instructions to minimize stalls and improve instruction-level parallelism.



TAC:1.f = 12.i = 23.if(i<=a) GOTO 94.t1 = f * i5.f = t1 6.t2 = i + 17.i = t2 8.GOTO 39.GOTO <Calling Program>BB1BB2BB3BB4HLL code:fact(a){int f = 1;for(i=2;i<=a;i++)f = f * i;}BB1BB2BB3BB4PFG:Illustration - Performing CFA:



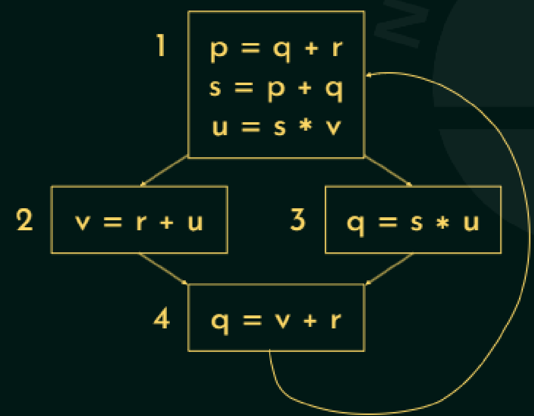
Liveness Analysis - Algorithm:Program Flow GraphInput: Liveness InformationOutput: --IN[B]-- OUT[B]-- DEF/KILL[B]-- USE/GEN[B](Set of variables that are used/generated in the Block)(Set of variables that are live at the beginning of the Block)(Set of variables that are live after the Block)(Set of variables that are defined/killed in the Block)



Liveness Analysis - Algorithm:IN and OUT sets are initially empty.1.Initialization: Create a worklist containing all the Basic Blocks of the CFG. 2.Worklist Initialization: 4.Final Liveness Information: After the analysis has converged, the 'IN' sets represent the live variables at the entry points of each block.3.Iterative Dataflow Analysis: While the worklist is not empty, perform the following steps for each basic block:●Calculate IN[B] = USE[B] ∪ (OUT[B] - DEF[B]) ●Calculate OUT[B] = ∪ IN[S]

Q: A variable x is said to be live at a statement S_i in a program if the following three conditions hold simultaneously:

1. There exists a statement S_j that uses x
2. There is a path from S_i to S_j in the flow graph corresponding to the program
3. The path has no intervening assignment to x including at S_i and S_j



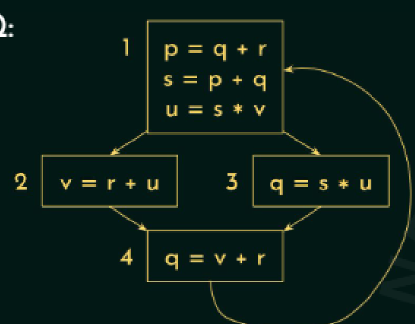
The variables which are live both at the statement in basic block 2 and at the statement in basic block 3 of the above control flow graph are

a. p, s, u
b. r, s, u
c. r, u
d. q, v

GATE 2015

Q:A variable x is said to be live at a statement Si in a program if the following three conditions hold simultaneously:1.There exists a statement Sj that uses x2.There is a path from Si to Sj in the flow graph corresponding to the program3.The path has no intervening assignment to x including at Si and Sjp = q + rs = p + qu = s + v*q = s + u*v = r + uq = v + r1234The variables which are live both at the statement in basic block 2 and at the statement in basic block 3 of the above control flow graph area.p, s, ub.r, s, uc.r, ud.q, vGATE 2015

Q:



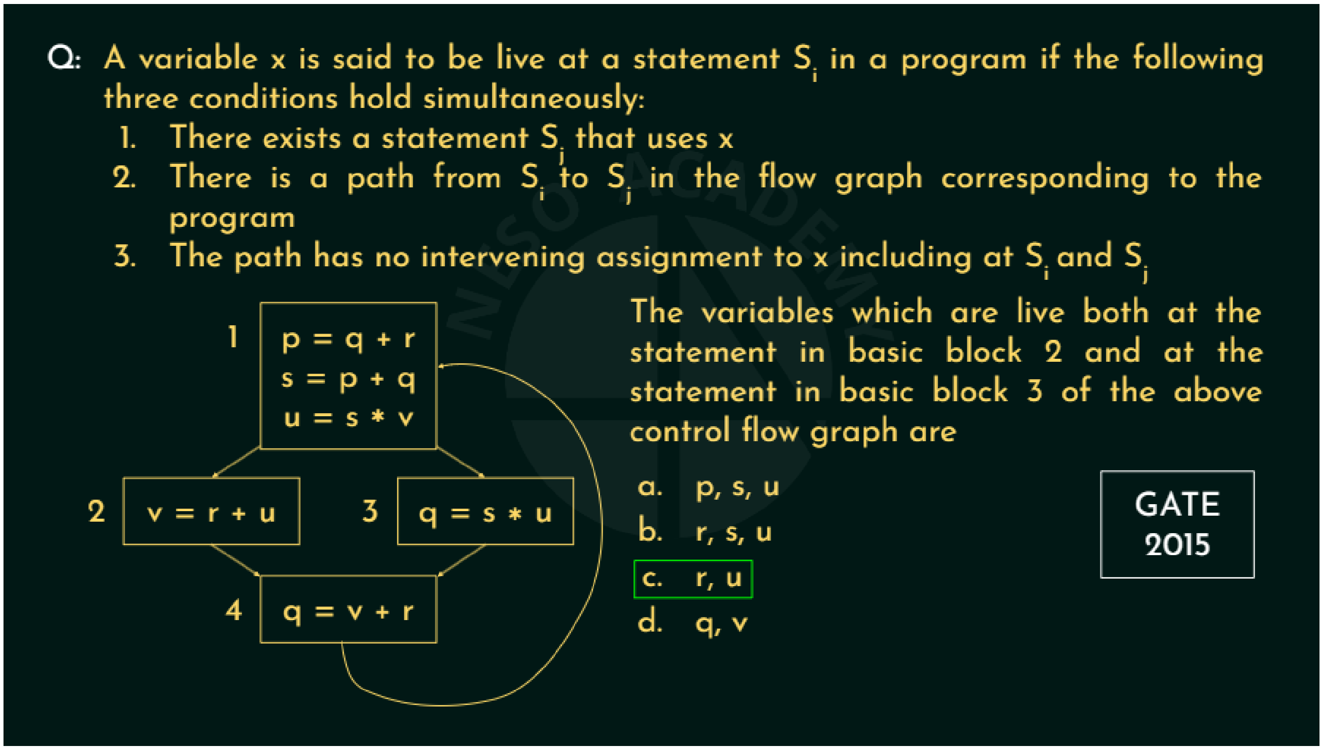
Liveness Analysis – Algorithm:

1. Initialization: IN and OUT sets are initially empty.
2. Worklist Initialization: Create a worklist containing all the Basic Blocks of the CFG.
3. Iterative Dataflow Analysis: While the worklist is not empty, perform the following steps for each basic block:
   - Calculate IN[B] = USE[B] U (OUT[B] - DEF[B])
   - Calculate OUT[B] = U IN[S]
4. Final Liveness Information: After the analysis has converged, the 'IN' sets represent the live variables at the entry points of each block.

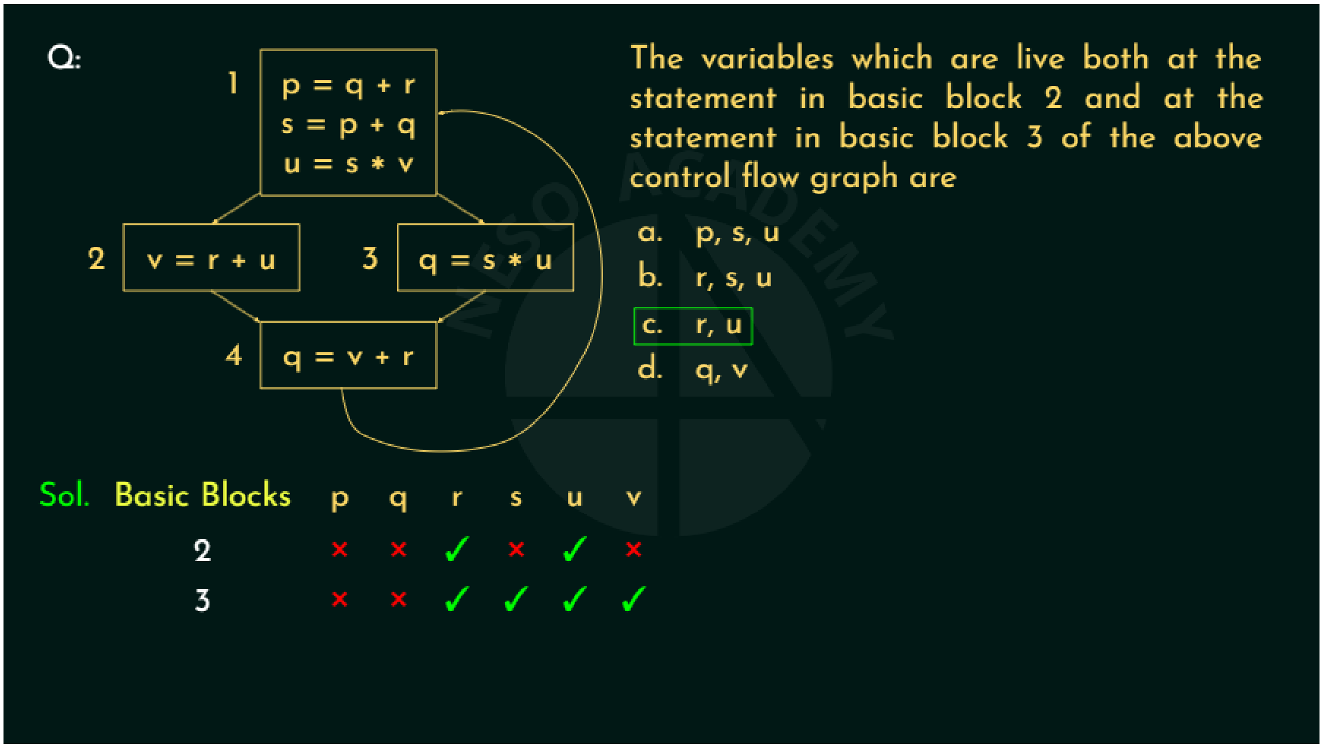
Sol.

Basic Block	USE	DEF	IN	OUT	IN	OUT	IN	OUT
1	q, r, v	p, s, u	q, r, v	r, u, s	r, q, v	r, u, s, v	r, v, q	r, u, s, v
2	r, u	v	r, u	v, r	r, u	r, v	r, u	r, v
3	s, u	q	s, u	v, r	v, r, s, u	r, v	r, v, s, u	r, v
4	v, r	q	v, r	q, r, v	r, v	r, q, v	r, v	r, v, q

, qQ:Sol.q, r, vr, us, uv, rp, s, uvqqq, r, vr, us, uv, rv, rv, rq, r, vr, vr, u, s, vr, vr, vr, q, vr, vr, u, s, vr, vr, vr, v, qINOUTINOUT1234Basic BlockUSEDEFINOUTr, u, sr, q, vr, uv, r, s, ur, vr, ur, v, s, u



Q:A variable x is said to be live at a statement Si in a program if the following three conditions hold simultaneously:1.There exists a statement Sj that uses x2.There is a path from Si to Sj in the flow graph corresponding to the program3.The path has no intervening assignment to x including at Si and Sjp = q + rs = p + qu = s + v*q = s + u*v = r + uq = v + r1234The variables which are live both at the statement in basic block 2 and at the statement in basic block 3 of the above control flow graph area.p, s, ub.r, s, uc.r, ud.q, vGATE 2015



Q:p = q + rs = p + qu = s + v*q = s + u*v = r + uq = v + r1234The variables which are live both at the statement in basic block 2 and at the statement in basic block 3 of the above control flow graph area.p, s, ub.r, s, uc.r, ud.q, vSol.Basic Blockspqrsv23 ✓✓ ✓u✓✓ ✓