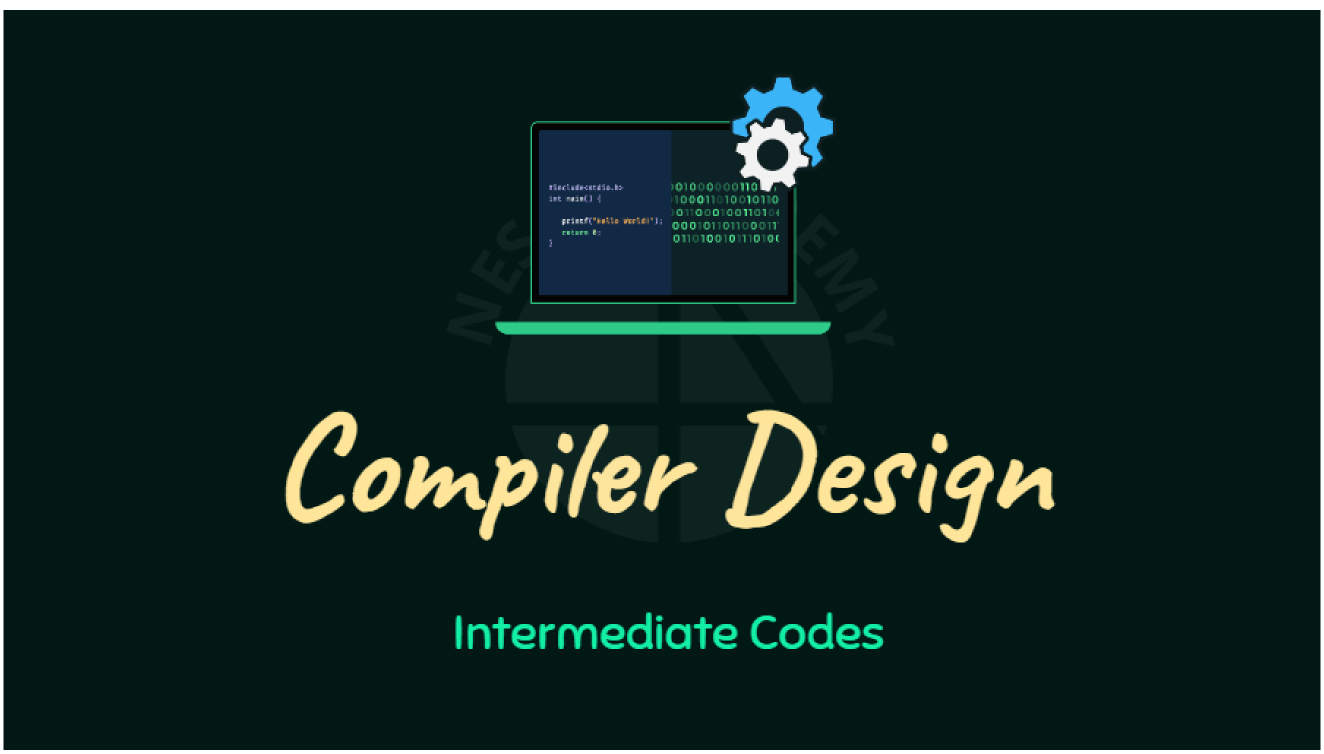
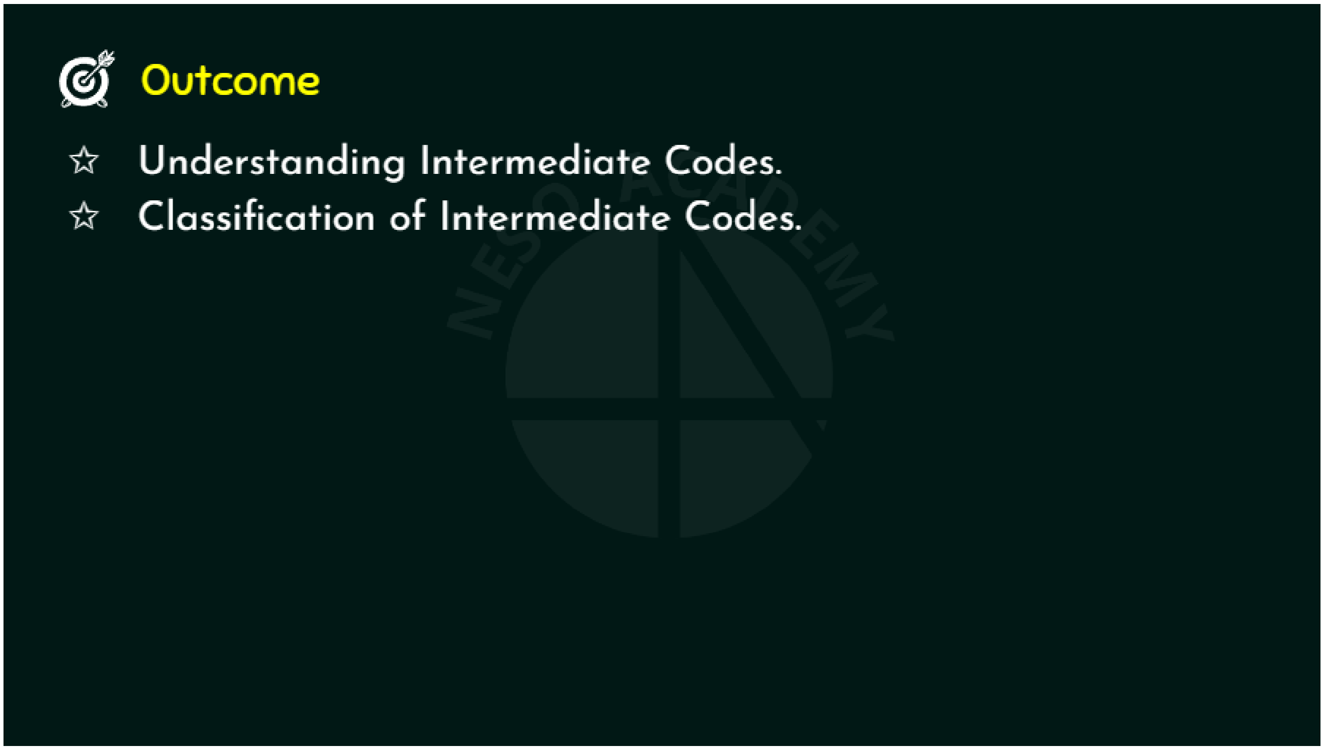
Intermediate Code Generation | Neso Academy

nesoacademy.org/cs/12-compiler-design/ppts/06-intermediatecodegeneration

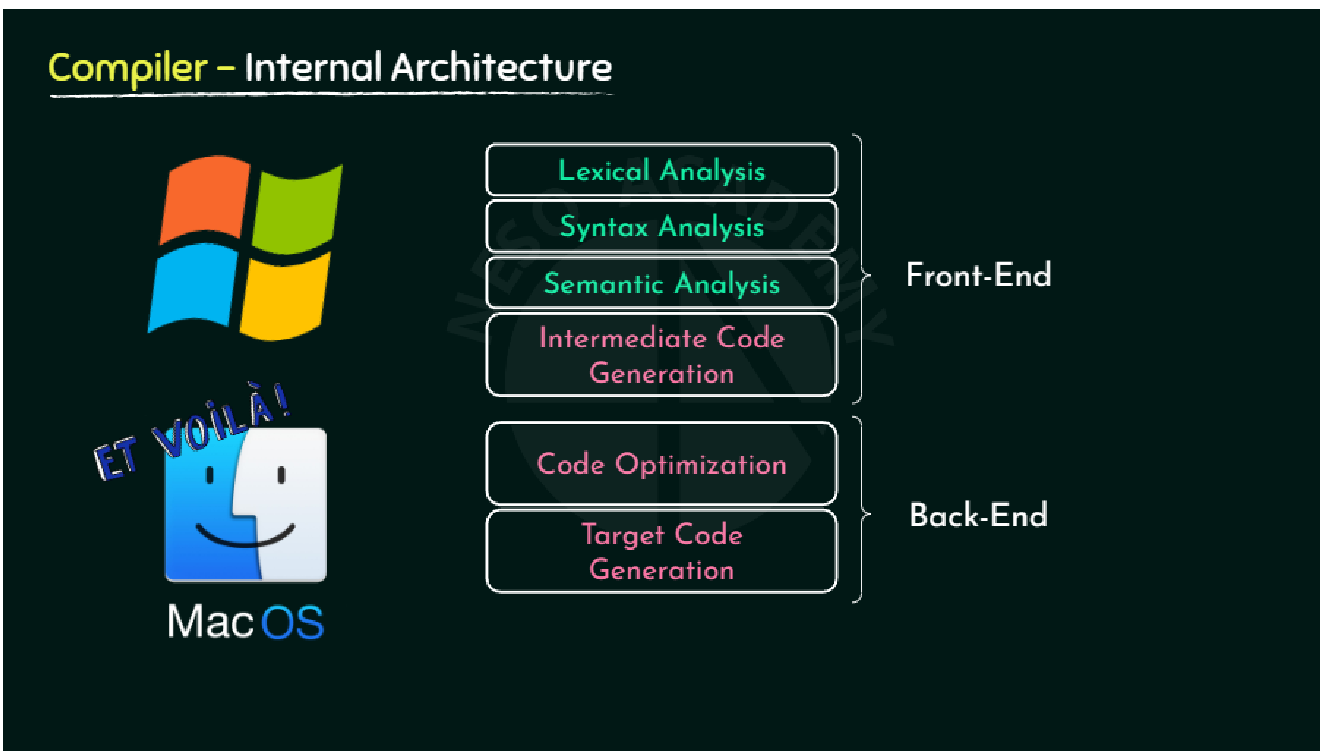
Intermediate Code GenerationNeso AcademyCHAPTER-6



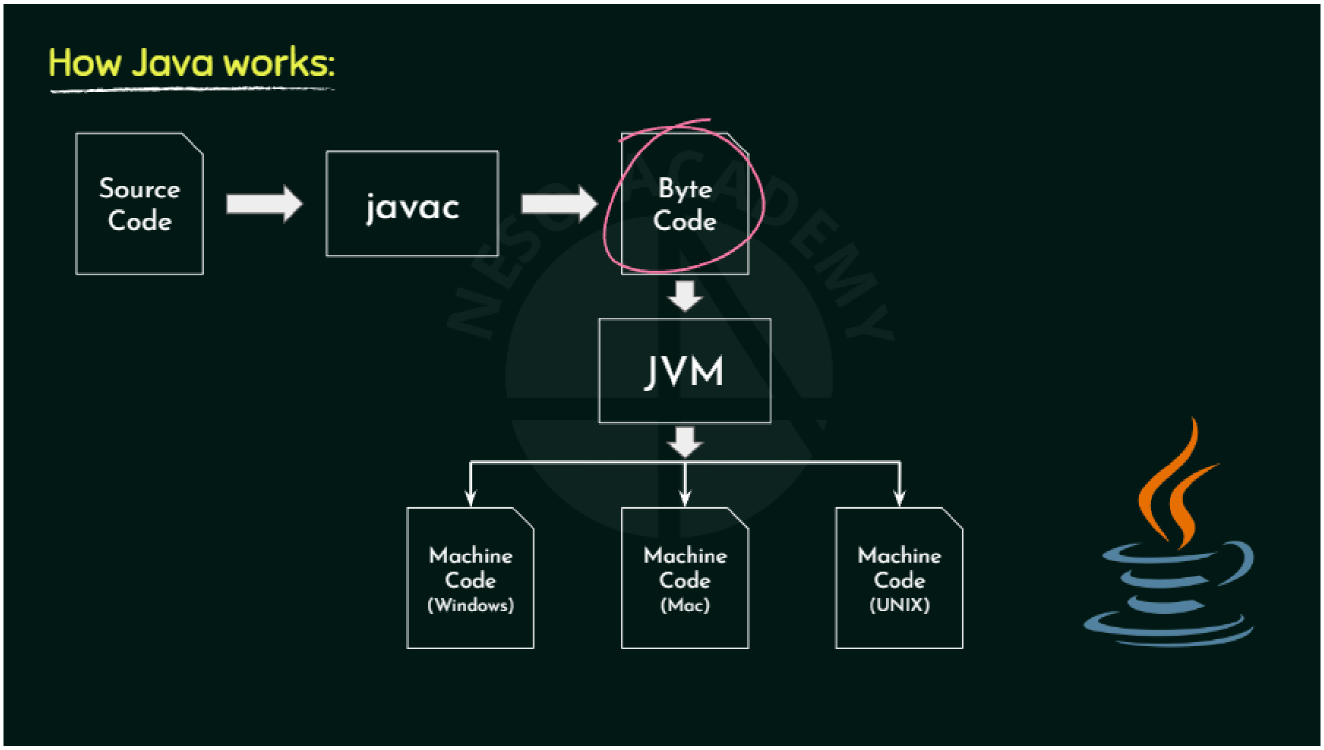
Compiler Design Intermediate Codes



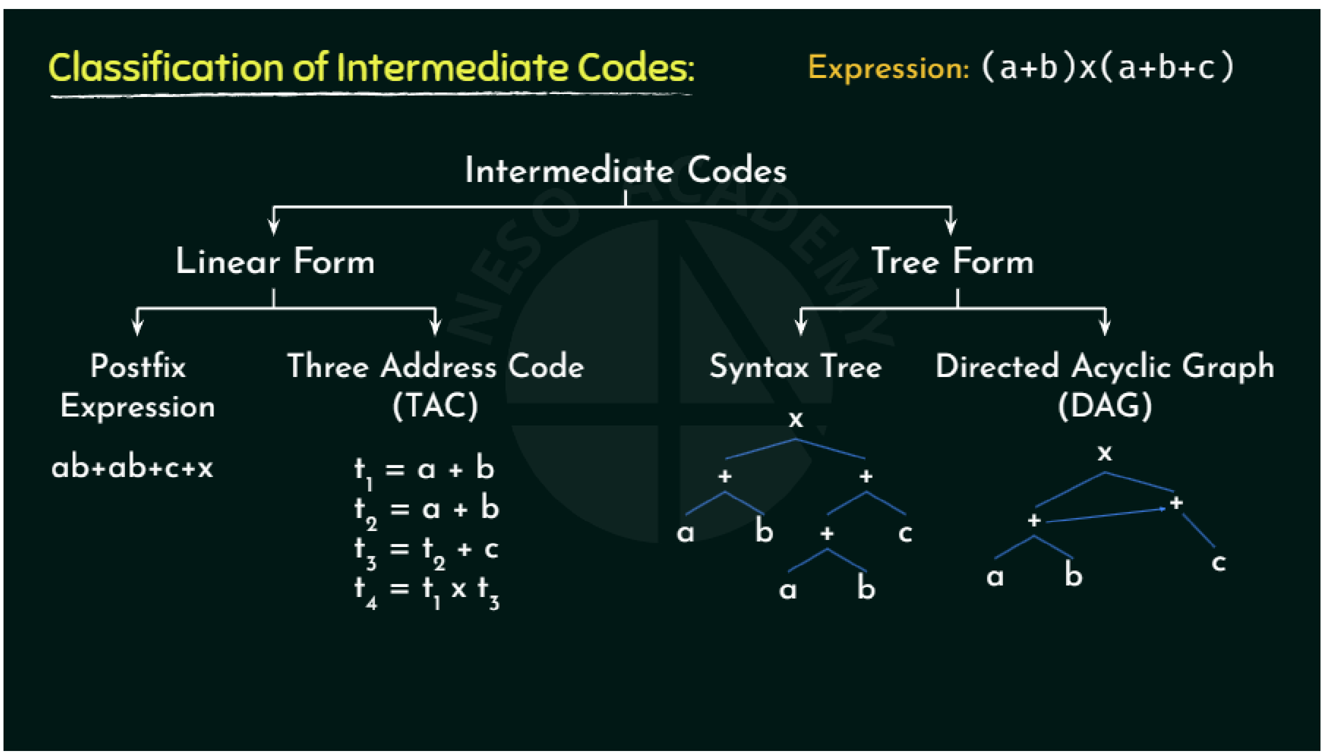
Outcome☆Understanding Intermediate Codes.☆Classification of Intermediate Codes.



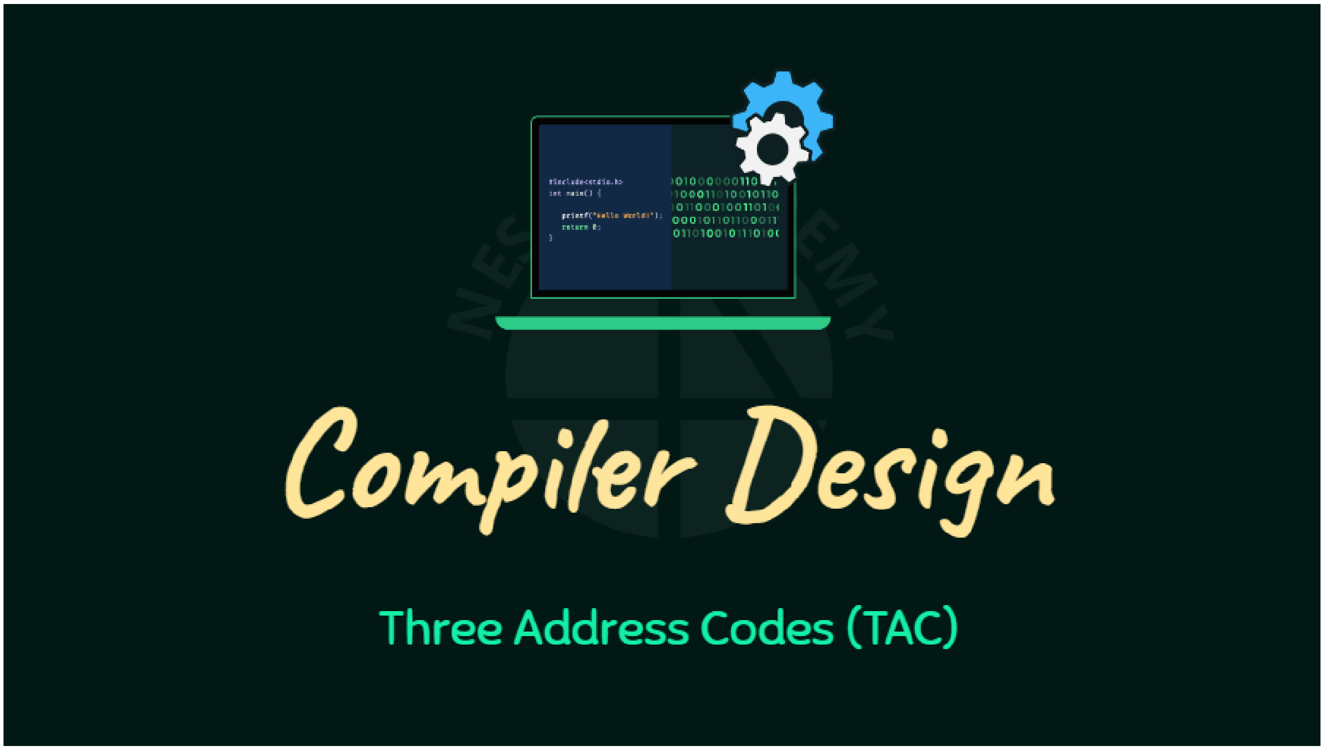
Lexical AnalysisSyntax AnalysisSemantic AnalysisIntermediate Code GenerationCode OptimizationTarget Code GenerationFront-EndBack-EndCompiler - Internal Architecture



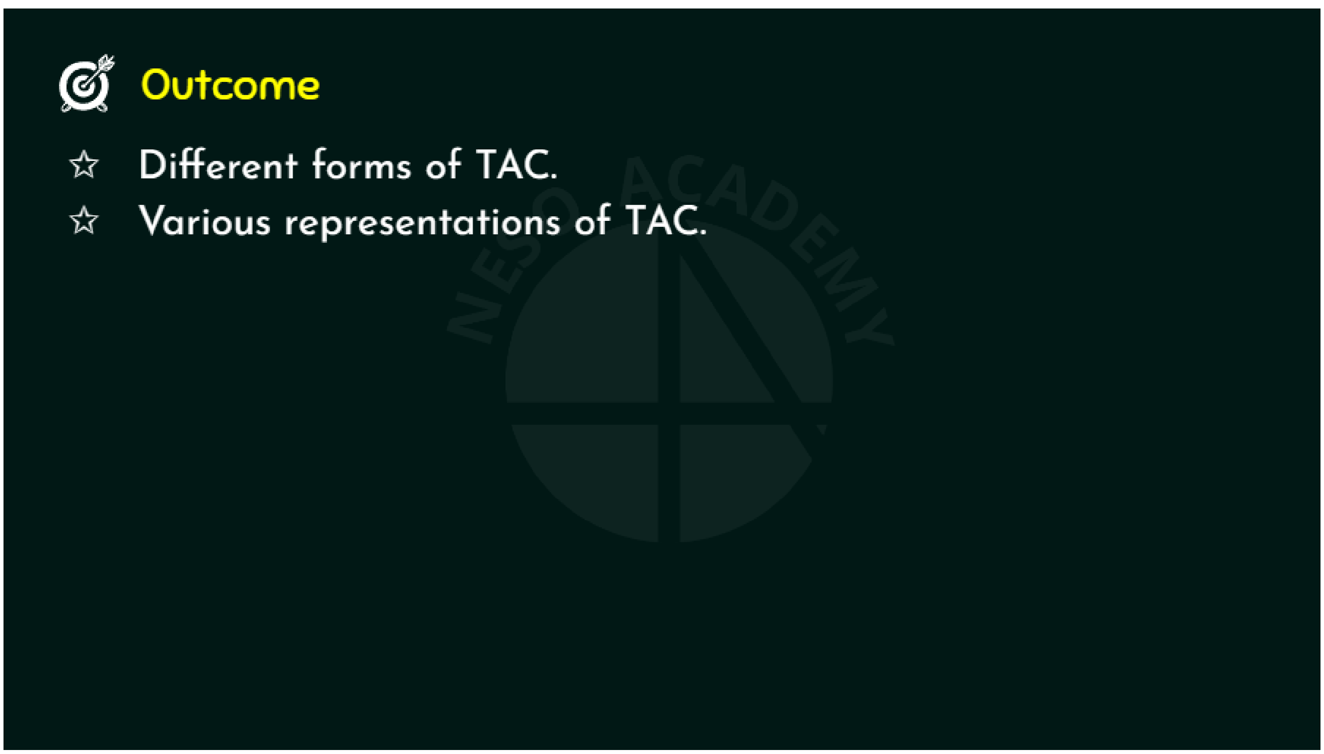
How Java works:javacSourceCodeByteCodeJVMMachineCode(Windows)MachineCode(Mac)MachineCode(UNIX)



xClassification of Intermediate Codes:Intermediate CodesTree FormLinear FormPostfixExpressionThree Address Code(TAC)Syntax TreeDirected Acyclic Graph(DAG)Expression: (a+b)x(a+b+c) ab+ab+c+t1 = a + bt2 = a + bt3 = t2 + ct4 = t1 x t3 x++ab+cabx++abc



Compiler Design Three Address Codes (TAC)



Outcome☆Different forms of TAC.☆Various representations of TAC.

Different forms of TAC:

The HLL expressions are converted into the intermediate code using any of the following TAC forms.

TAC Form	Usage
1. x = y op z	For Binary Operation & then assignment.
2. x = op z	For Unary Operation & then assignment.
3. x = y	For simple assignment.
4. if (x <rel op> y) GOTO L	Conditional GOTO.
5. GOTO L	Unconditional GOTO.
6. A[i] = x y = A[i]	Used for arrays.
7. x = *p y = &x	'x' is a value pointed by the pointer 'p'. Address of 'x' is stored in 'y'.

Different forms of TAC:The HLL expressions are converted into the intermediate code using any of the following TAC forms.TAC Form Usage 1.x = y op z 2.x = op z 3.x = y4.if (x <rel op> y) GOTO L 5.GOTO L6.A[i] = xy = A[i] 7.x = +py = &x * For Binary Operation & then assignment. For Unary Operation & then assignment. For simple assignment. Conditional GOTO. Unconditional GOTO. Used for arrays. 'x' is a value pointed by the pointer 'p'. Address of 'x' is stored in 'y'.

Representations of TAC:

Traditionally, three types of representations of TAC are popularly used.

1. Quadruples:
   These use temporary variables (in temporary registers) for each individual portion of the expression in order to store the result.
   Each instruction is splitted into the following 4 different fields:
   opr, *op1*, *op2*, *result*
2. Triples:
   For these temporary variables are not dedicated rather temporary registers are used on demand. The flow of evaluation is numbered and cannot be altered.
3. Indirect Triples:
   It uses pointers to the listing of all references to computations which is made separately and stored in memory.
   Temporaries are implicit and its rearrangeable like quadruples.

Traditionally, three types of representations of TAC are popularly used.1.Quadruples:These use temporary variables (in temporary registers) for each individual portion of the expression in order to store the result.Each instruction is splitted into the following 4 different fields:opr, op1, op2, result2.Triples:For these temporary variables are not dedicated rather temporary registers are used on demand. The flow of evaluation is numbered and cannot be altered.3.Indirect Triples:It uses pointers to the listing of all references to computations which is made separately and stored in memory.Temporaries are implicit and its rearrangeable like quadruples.Representations of TAC:

Representations of TAC:

Expression: -(a+b) x (c+d) + (a+b+c)

Three Address Code:

1. $t_1 = a + b$
2. $t_2 = -t_1$
3. $t_3 = c + d$
4. $t_4 = t_2 \times t_3$
5. $t_5 = a + b$
6. $t_6 = t_5 + c$
7. $t_7 = t_4 + t_6$

	Quadruples					Triples			Indirect Triples	
	opr	op1	op2	result		opr	op1	op2		Reference
1.	+	a	b	t_1	1.	+	a	b	i.	(1)
2.	-	t_1		t_2	2.	-	(1)		ii.	(2)
3.	+	c	d	t_3	3.	+	c	d	iii.	(3)
4.	x	t_2	t_3	t_4	4.	x	(2)	(3)	iv.	(4)
5.	+	a	b	t_5	5.	+	a	b	v.	(5)
6.	+	t_5	c	t_6	6.	+	(5)	c	vi.	(6)
7.	+	t_4	t_6	t_7	7.	+	(4)	(6)	vii.	(7)

Representations of TAC:Expression: -(a+b) x (c+d) + (a+b+c)Three Address Code:1.t1 = a + b2.t2 = -t13.t3 = c + d4.t4 = t2 x t3 5.t5 = a + b6.t6 = t5 + c7.t7 = t4 + t6resultQuadruplesopr op1 op2opr op1 op2TriplesReferenceIndirect Triples1.+abt11.+ab(1)2.-t1t22.-(1)(2)+cd3.t3+cd3.(3)4.t4xt2t3x4.(2)(3)(4)+ab5.t5+ab5.(5)+c6.t6t5+c6.(5)(6)+7.t7t4t6+7.(4)(6)(7)i.ii.iii.iv.v.vi.vii.

Representations of TAC:

Expression: -(a+b) x (c+d) + (a+b+c)

Three Address Code:

1. t_1 = a + b
2. t_2 = -t_1
3. t_3 = c + d
4. t_4 = t_2 x t_3
5. t_5 = a + b
6. t_6 = t_5 + c
7. t_7 = t_4 + t_6

	Quadruples			
	opr	op1	op2	result
1.	+	a	b	t_1
2.	-	t_1		t_2
3.	+	c	d	t_3
4.	x	t_2	t_3	t_4
5.	+	a	b	t_5
6.	+	t_5	c	t_6
7.	+	t_4	t_6	t_7

Pro:
Statements can be rearranged.

Con:
Too many registers are needed.

Representations of TAC:Expression: -(a+b) x (c+d) + (a+b+c)Three Address Code:1.t1 = a + b2.t2 = -t13.t3 = c + d4.t4 = t2 x t3 5.t5 = a + b6.t6 = t5 + c7.t7 = t4 + t6Quadruplesopr op1 op21.+abt12.-t1t2+cd3.t3x4.t4t2t3+ab5.t5+c6.t6t5+7.t7t4t6Pro: Statements can be rearranged.Con: Too many registers are needed.result

Representations of TAC:

Expression: -(a+b) x (c+d) + (a+b+c)

Three Address Code:

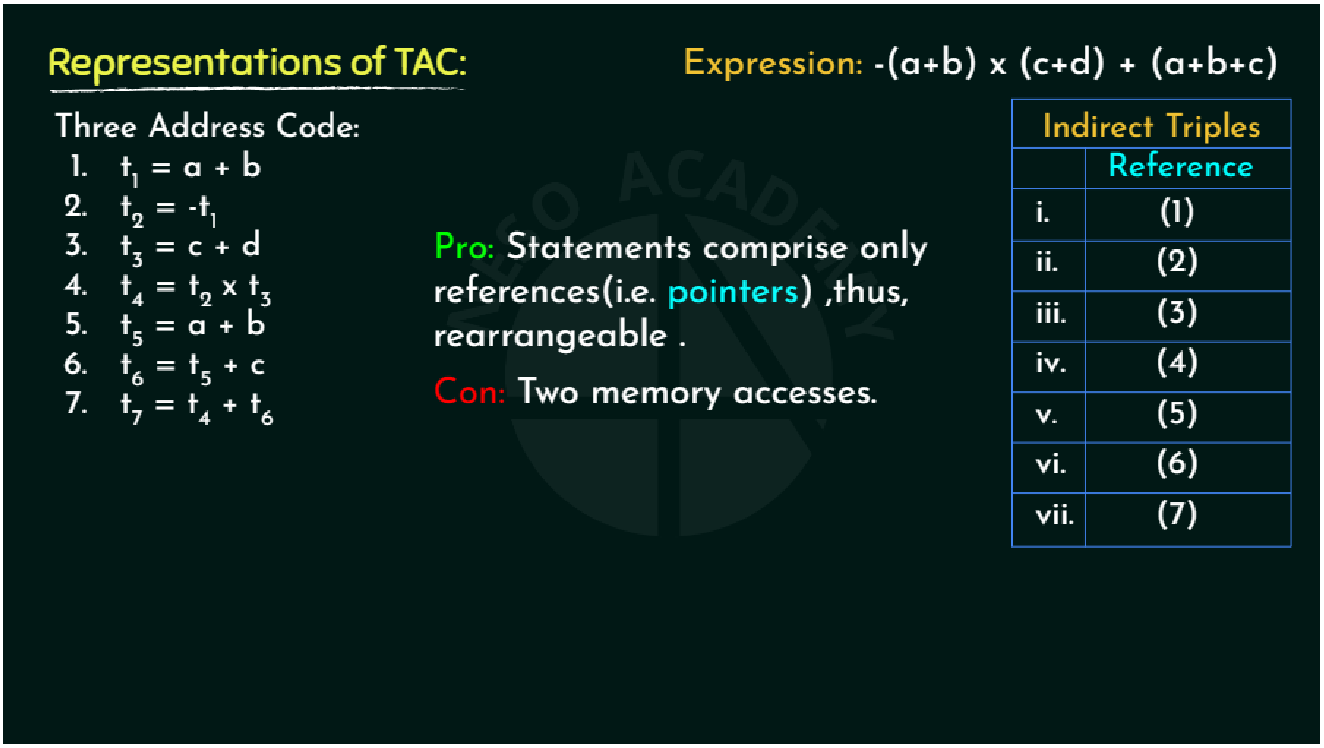
1. t_1 = a + b
2. t_2 = -t_1
3. t_3 = c + d
4. t_4 = t_2 x t_3
5. t_5 = a + b
6. t_6 = t_5 + c
7. t_7 = t_4 + t_6

Pro:
Temporary registers are used on demand.

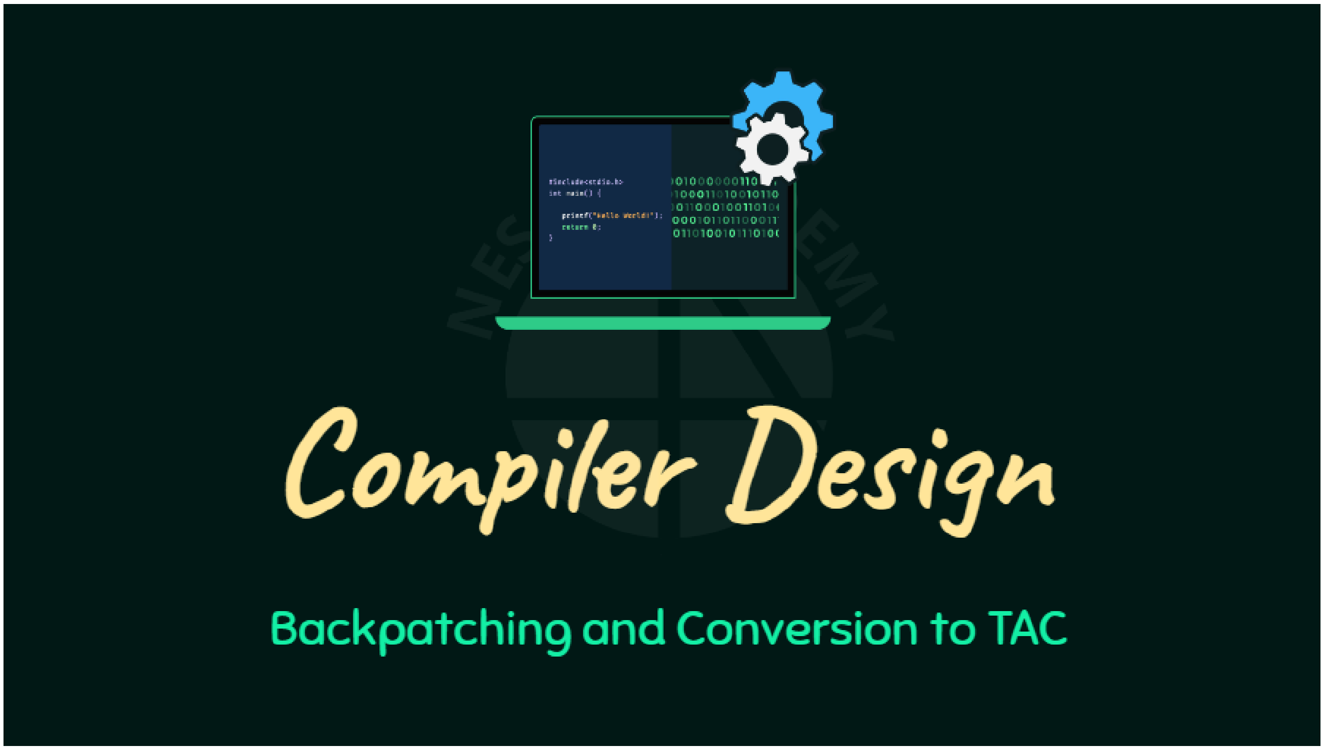
	Triples		
	opr	op1	op2
1.	+	a	b
2.	-	(1)	b
3.	+	c	d
4.	x	(2)	(3)
5.	+	a	b
6.	+	(5)	c
7.	+	(4)	(6)

Con:
Statements can't be rearranged.

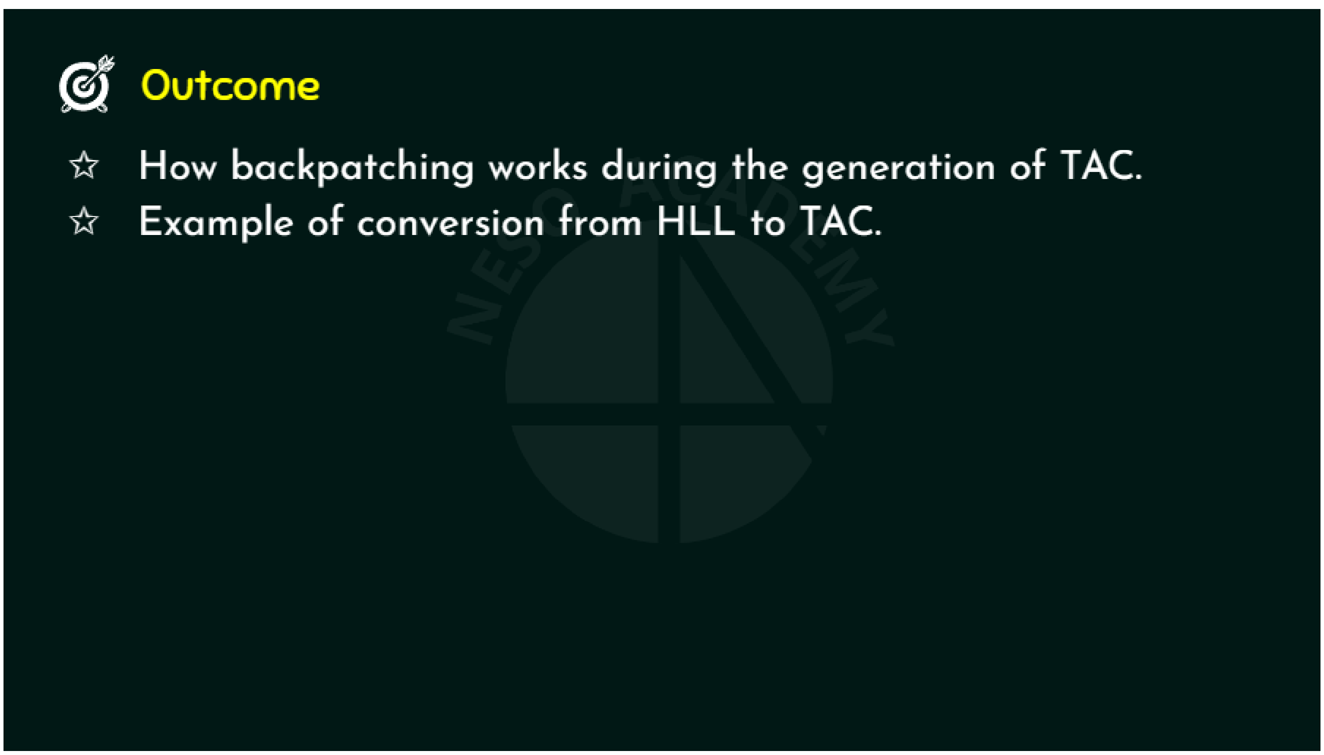
Representations of TAC:Expression: -(a+b) x (c+d) + (a+b+c)Three Address Code:1.t1 = a + b2.t2 = -t13.t3 = c + d4.t4 = t2 x t3 5.t5 = a + b6.t6 = t5 + c7.t7 = t4 + t6opr op1 op2Triples1.+ab2.-b+cd3.x4.+ab5.+c6.+7.Pro:Temporary registers are used on demand.Con:Statements can’t be rearranged.(1)(2)(3)(5)(4)(6)



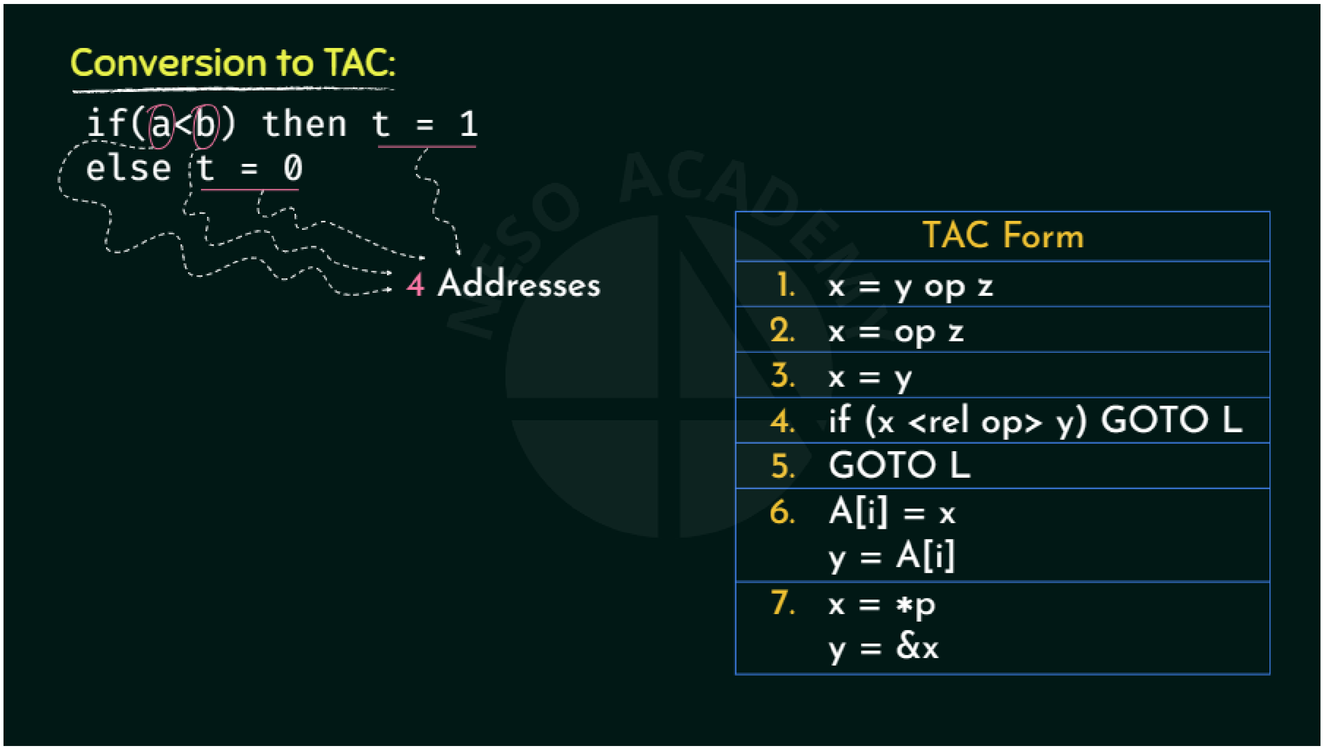
Representations of TAC:Expression: -(a+b) x (c+d) + (a+b+c)Three Address Code:1.t1 = a + b2.t2 = -t13.t3 = c + d4.t4 = t2 x t3 5.t5 = a + b6.t6 = t5 + c7.t7 = t4 + t6Indirect TriplesReference(1)i.(2)ii.(3)iii.(4)iv.(5)v.(6)vi.(7)vii.Pro: Statements comprise only references(i.e. pointers) ,thus, rearrangeable .Con: Two memory accesses.



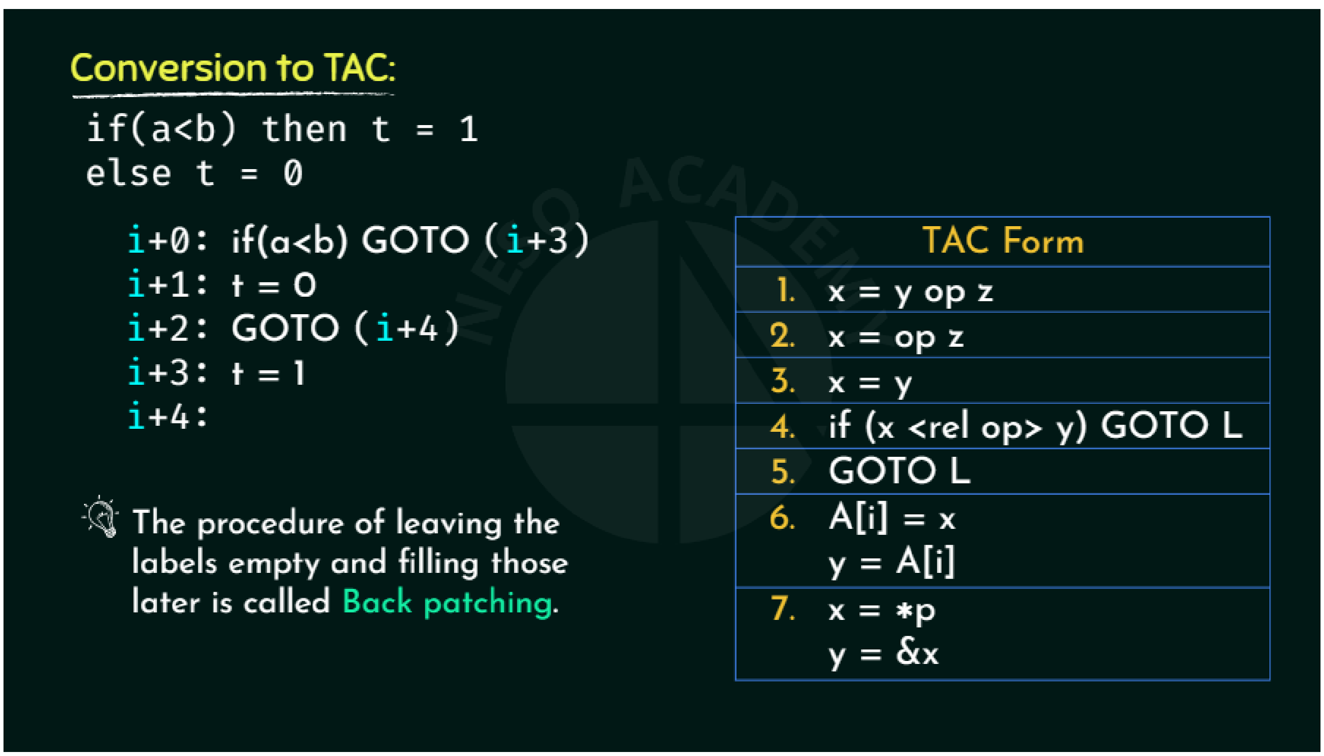
Compiler Design Backpatching and Conversion to TAC



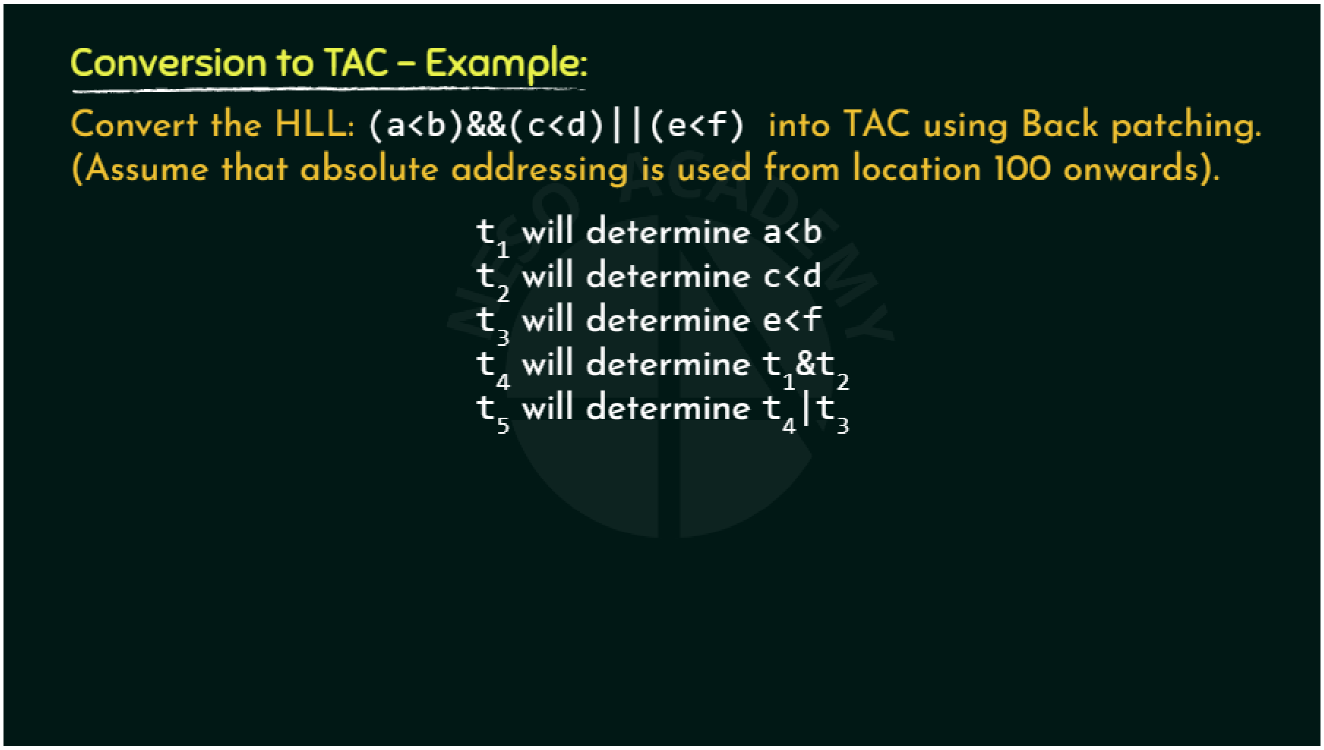
Outcome☆How backpatching works during the generation of TAC.☆Example of conversion from HLL to TAC.



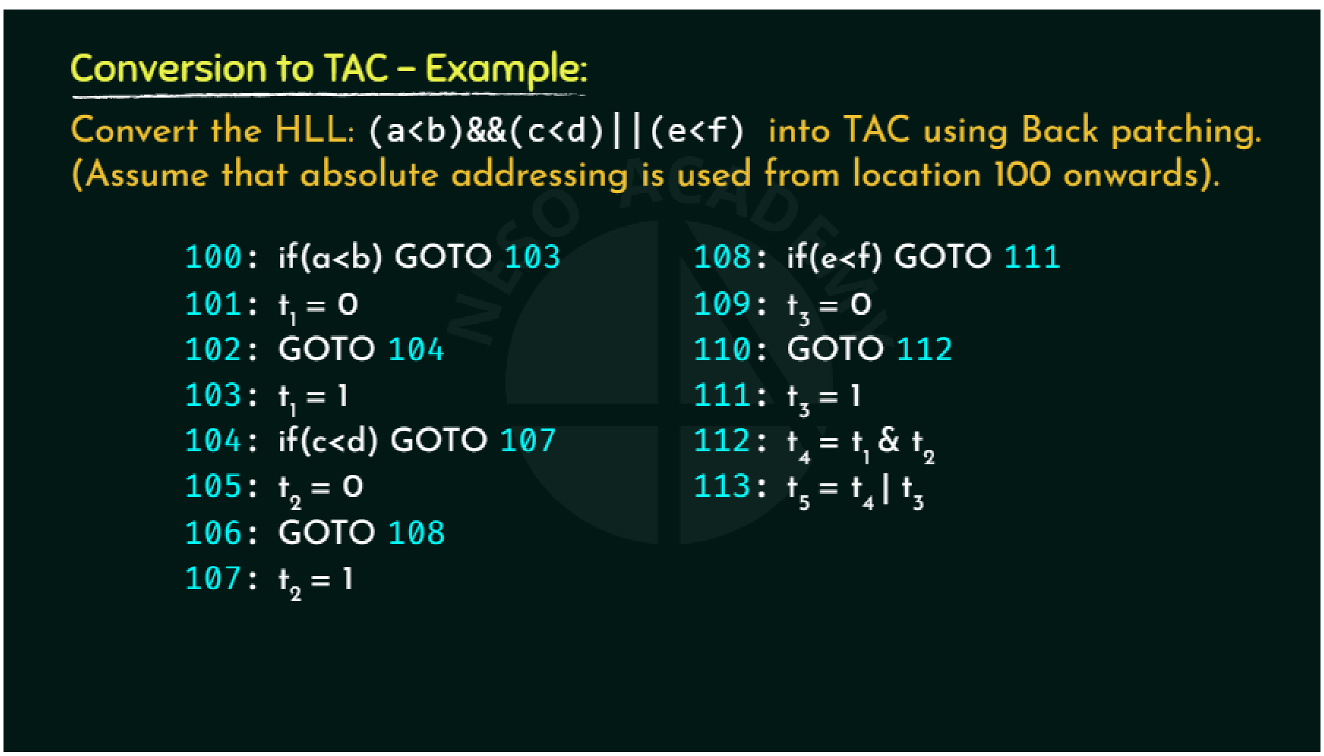
if(a<b) then t = 1else t = 0Conversion to TAC:4.if (x <rel op> y) GOTO L TAC Form 1.x = y op z 2.x = op z 3.x = y5.GOTO L6.A[i] = xy = A[i] 7.x = +py = &x * 4 Addresses



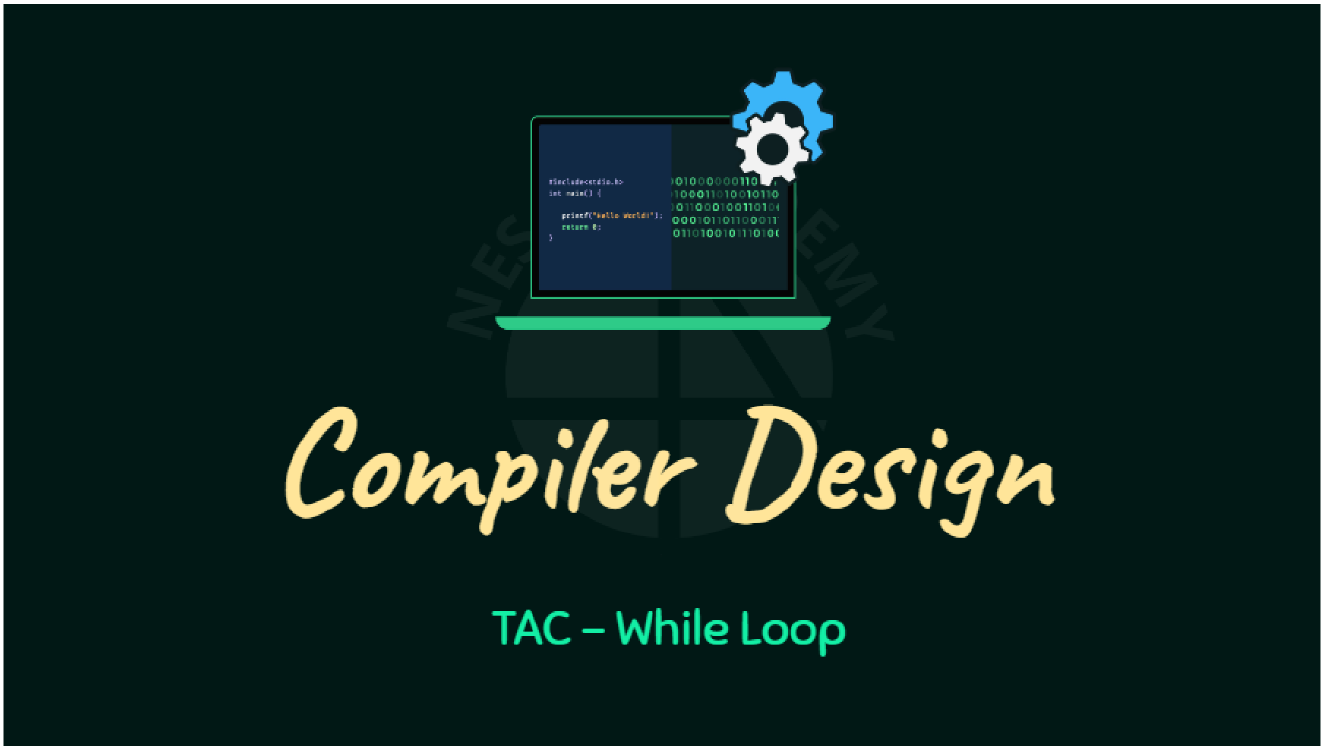
if(a<b) then t = 1else t = 0Conversion to TAC:4.if (x <rel op> y) GOTO L TAC Form 1.x = y op z 2.x = op z 3.x = y5.GOTO L6.A[i] = xy = A[i] 7.x = +py = &x * if(a<b) GOTO (i+3)GOTO (i+4).t = 0t = 1i+0:i+1:i+2:i+3:i+4:The procedure of leaving the labels empty and filling those later is called Back patching.



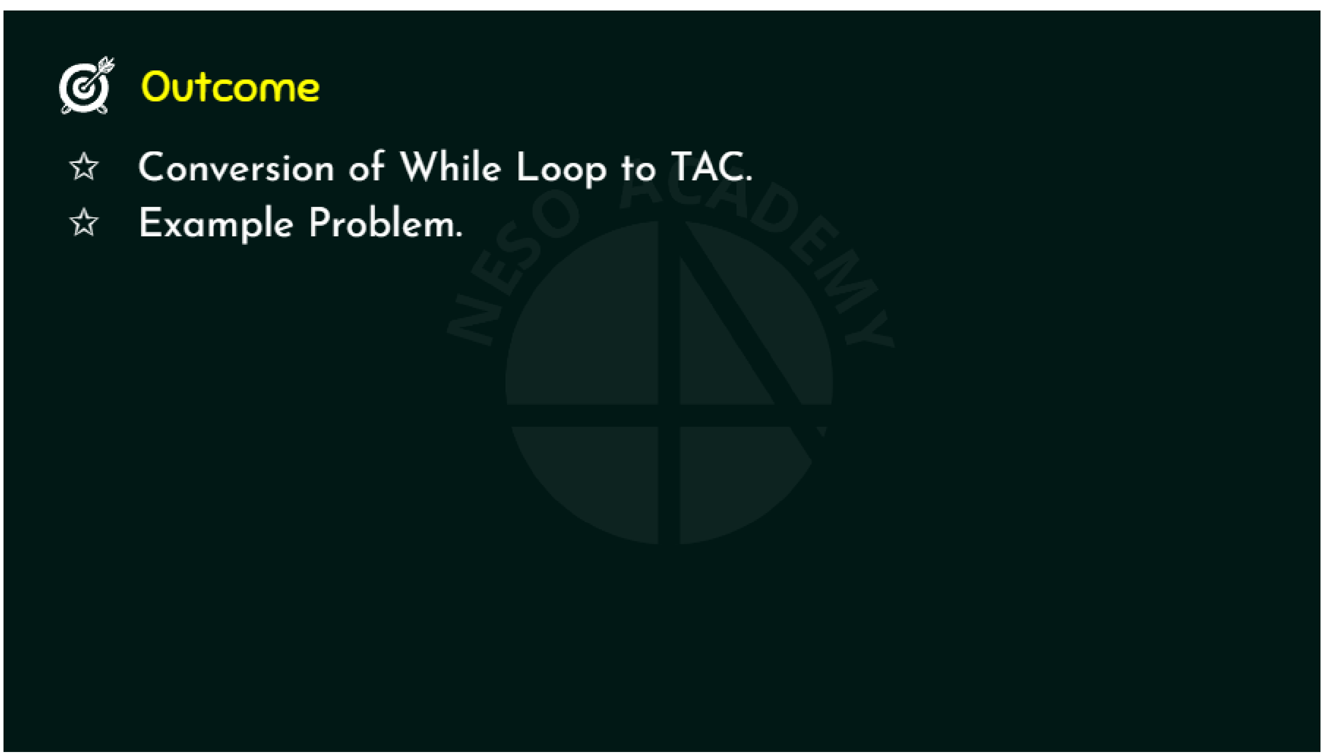
Convert the HLL: (a<b)&&(c<d)||(e<f) into TAC using Back patching. (Assume that absolute addressing is used from location 100 onwards). Conversion to TAC - Example:t1 will determine a<bt2 will determine c<dt3 will determine e<ft4 will determine t1&t2t5 will determine t4|t3



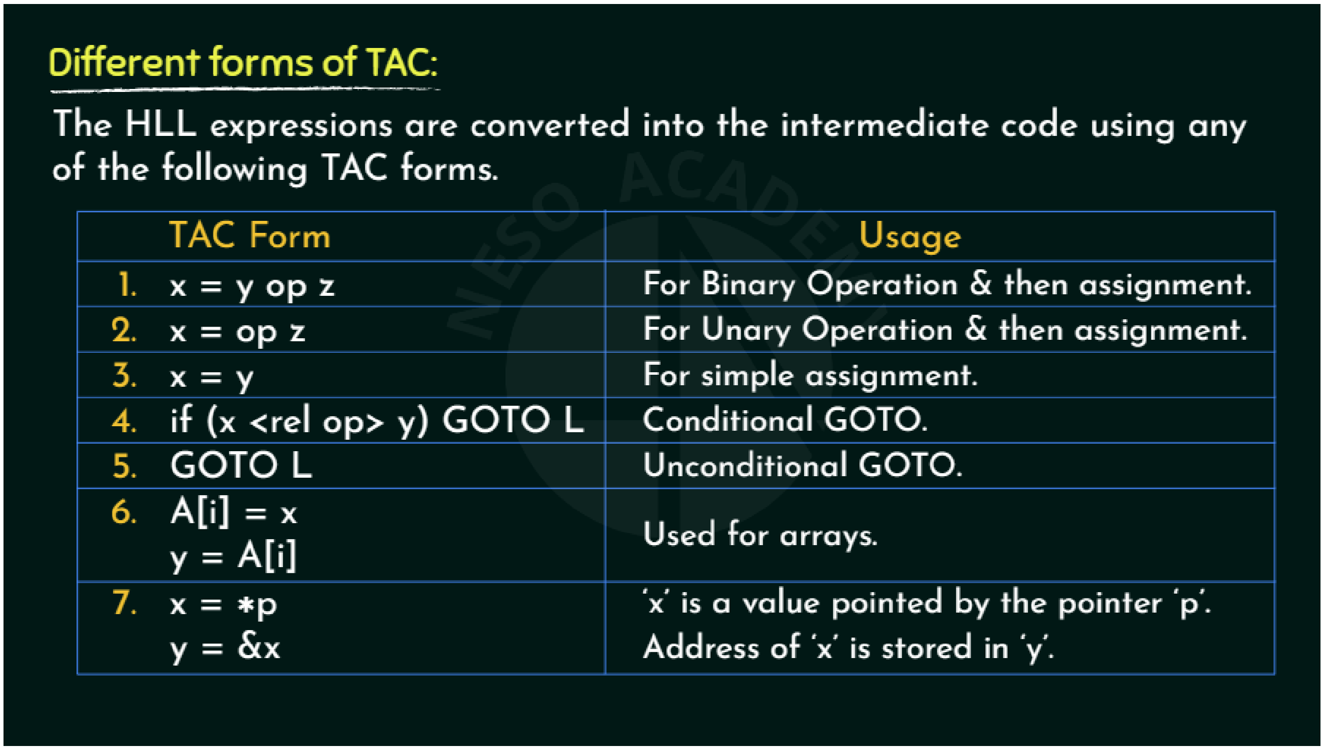
Convert the HLL: (a<b)&&(c<d)||(e<f) into TAC using Back patching. (Assume that absolute addressing is used from location 100 onwards). Conversion to TAC - Example:100:if(a<b) GOTO 103.t1 = 0101:GOTO 104102:t1 = 1103:if(c<d) GOTO 107104:t2 = 0105:GOTO 108106:t2 = 1107:108:if(e<f) GOTO 111.109:t3 = 0110:GOTO 112.111:t3 = 1112:t4 = t1 & t2 113:t5 = t4 | t3



Compiler Design TAC - While Loop



Outcome☆Conversion of While Loop to TAC.☆Example Problem.



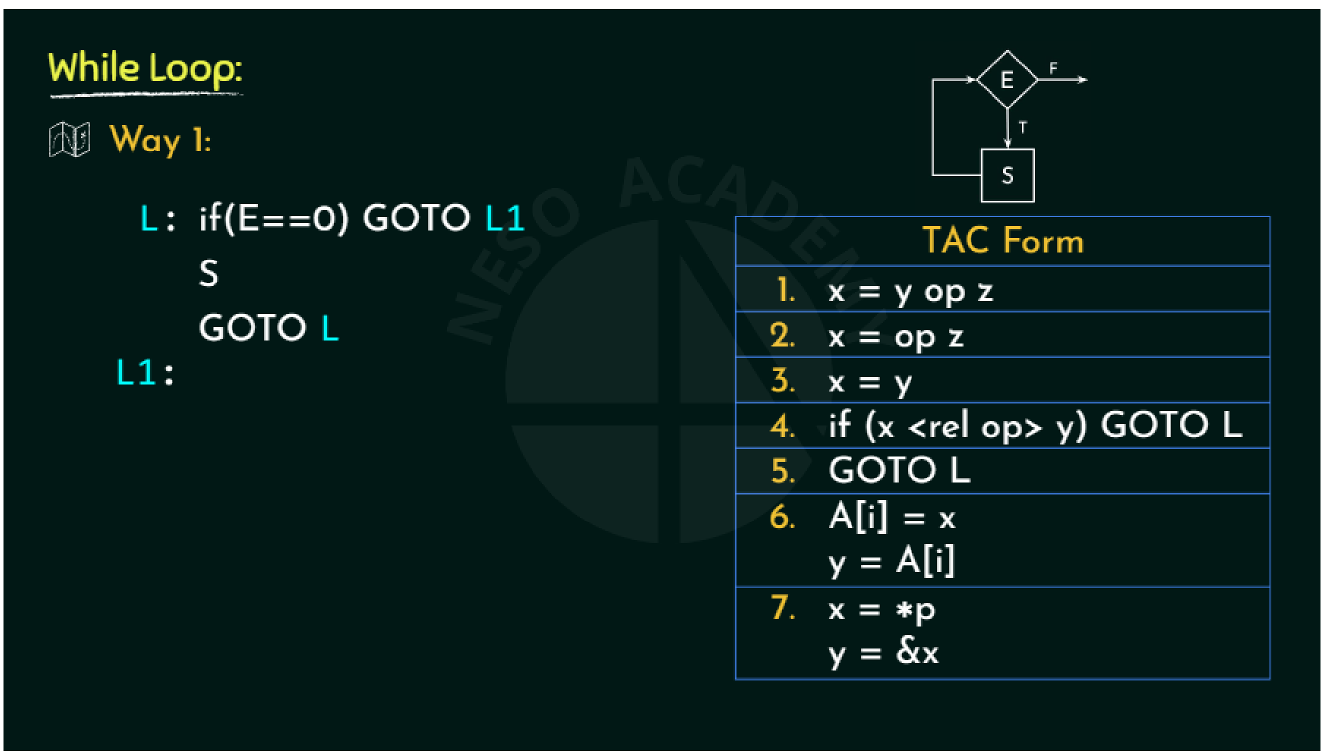
Different forms of TAC:

The HLL expressions are converted into the intermediate code using any of the following TAC forms.

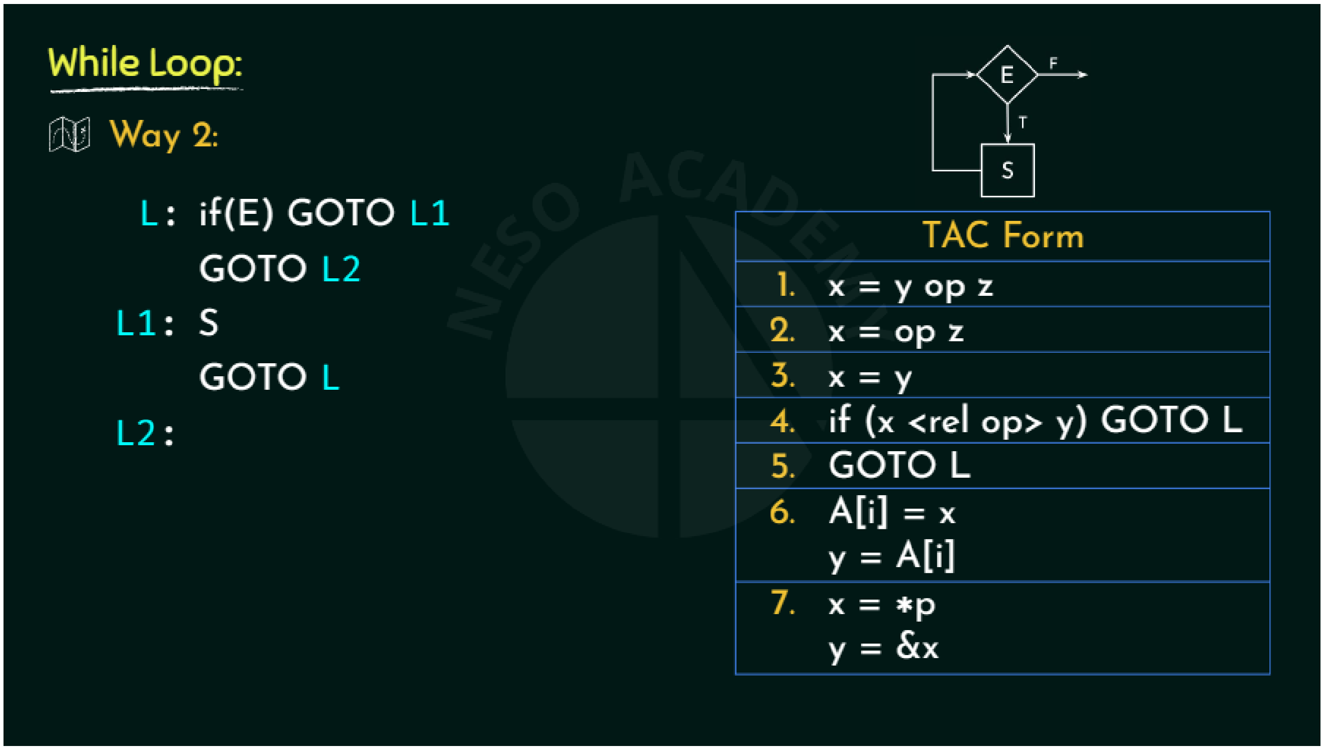
	TAC Form	Usage
1.	x = y op z	For Binary Operation & then assignment.
2.	x = op z	For Unary Operation & then assignment.
3.	x = y	For simple assignment.
4.	if (x <rel op> y) GOTO L	Conditional GOTO.
5.	GOTO L	Unconditional GOTO.
6.	A[i] = x y = A[i]	Used for arrays.
7.	x = *p y = &x	‘x’ is a value pointed by the pointer ‘p’. Address of ‘x’ is stored in ‘y’.

TAC Form Usage Different forms of TAC:The HLL expressions are converted into the intermediate code using any of the following TAC forms.1.x = y op z 2.x = op z 3.x = y4.if (x <rel op> y) GOTO L 5.GOTO L6.A[i] = xy = A[i] 7.x = +py = &x * For Binary Operation & then assignment. For Unary Operation & then assignment. For simple assignment. Conditional GOTO. Unconditional GOTO. Used for arrays. ‘x’ is a value pointed by the pointer ‘p’. Address of ‘x’ is stored in ‘y’.

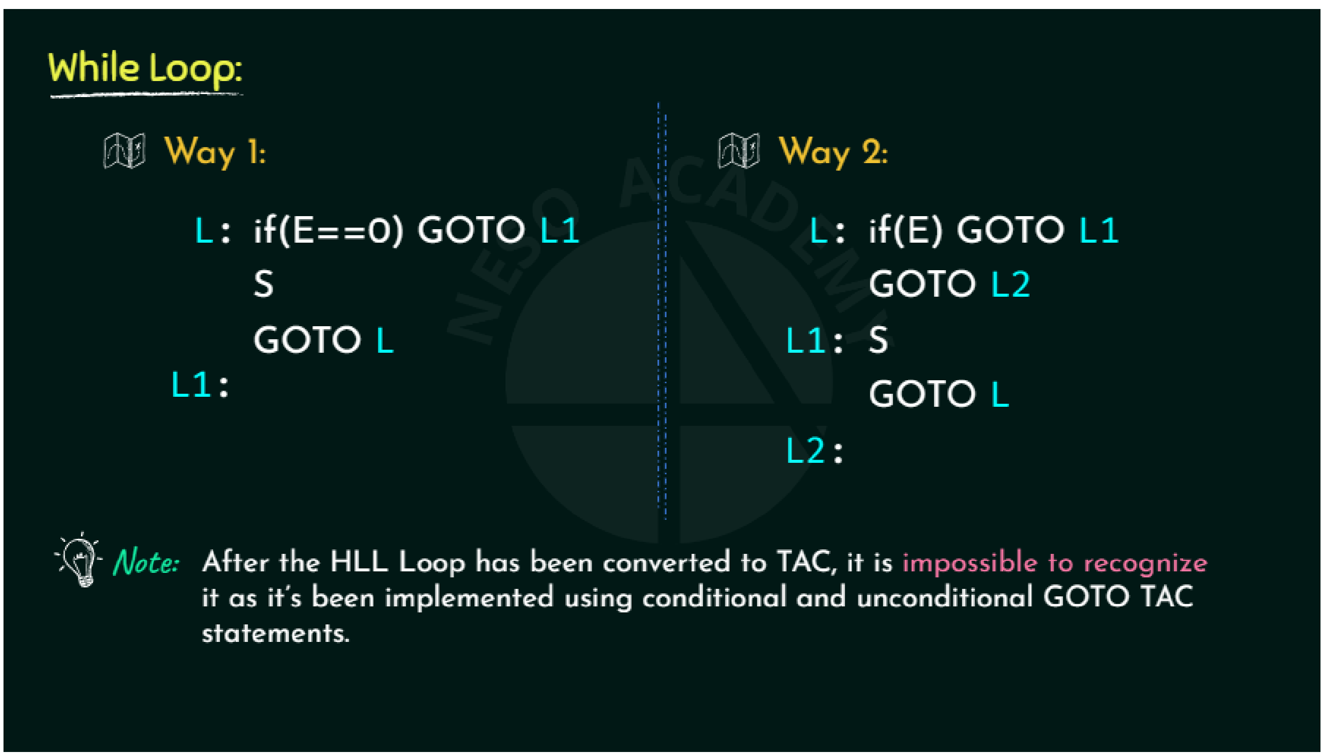
While Loop:while(E) do Swhile(E){ S}ESTF



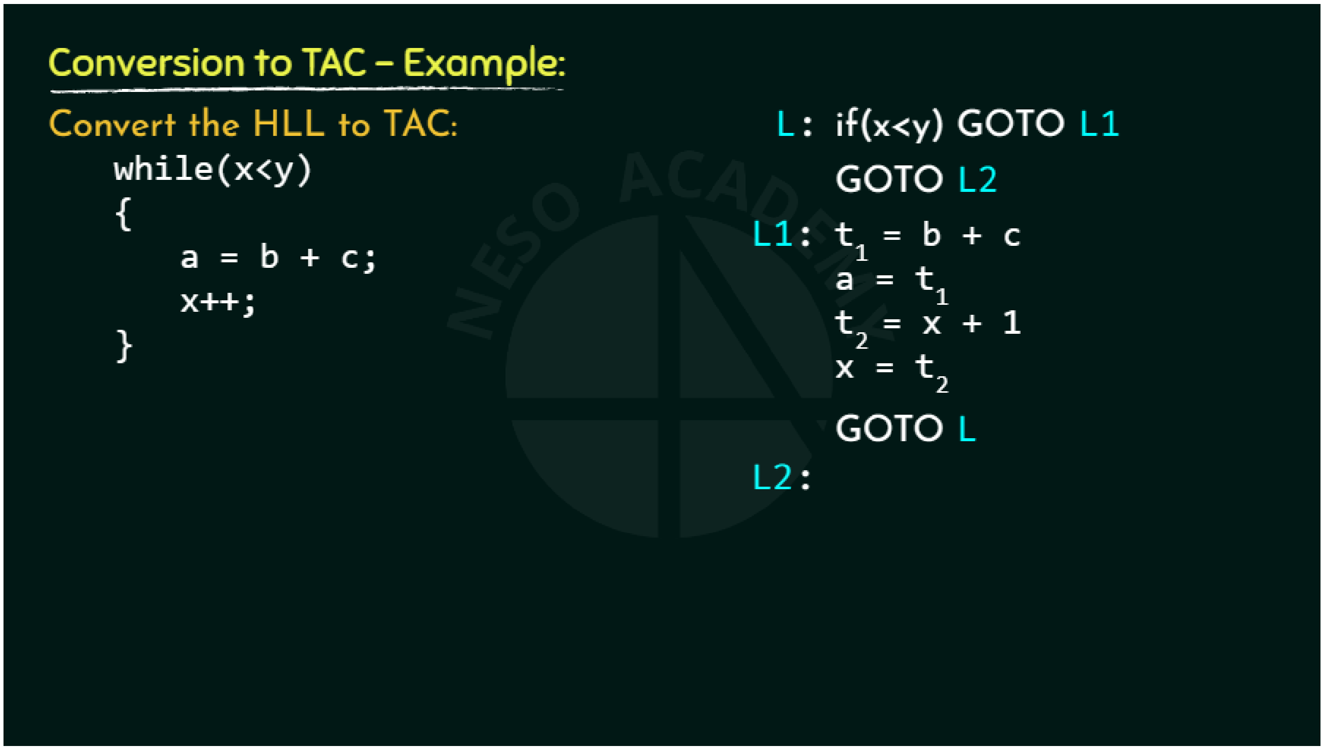
Way 1:While Loop:4.if (x <rel op> y) GOTO L TAC Form 1.x = y op z 2.x = op z 3.x = y5.GOTO L6.A[i] = xy = A[i] 7.x = +py = &x * if(E==0) GOTO L1GOTO LSL:L1:



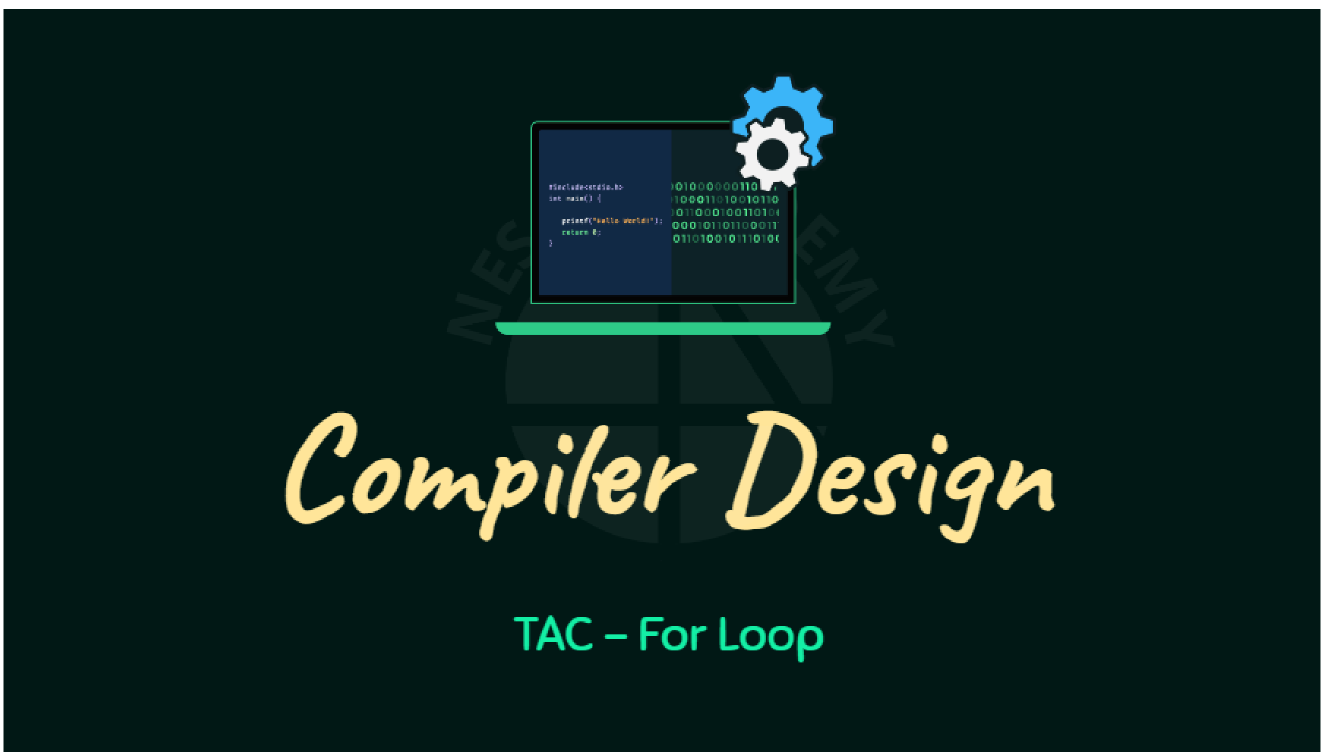
Way 2:While Loop:4.if (x <rel op> y) GOTO L TAC Form 1.x = y op z 2.x = op z 3.x = y5.GOTO L6.A[i] = xy = A[i] 7.x = +py = &x * if(E) GOTO L1SL:L1:GOTO L2GOTO LL2:



While Loop:Way 2:if(E) GOTO L1SL:L1:GOTO L2GOTO LL2:Way 1:if(E==0) GOTO L1GOTO LSL:L1:Note:After the HLL Loop has been converted to TAC, it is impossible to recognize it as it’s been implemented using conditional and unconditional GOTO TAC statements.



Convert the HLL to TAC: while(x<y){a = b + c;x++;}Conversion to TAC - Example:if(x<y) GOTO L1t1 = b + ca = t1t2 = x + 1x = t2L:L1:GOTO L2GOTO LL2:



Compiler Design TAC - For Loop

Outcome

☆ Conversion of For Loop to TAC.

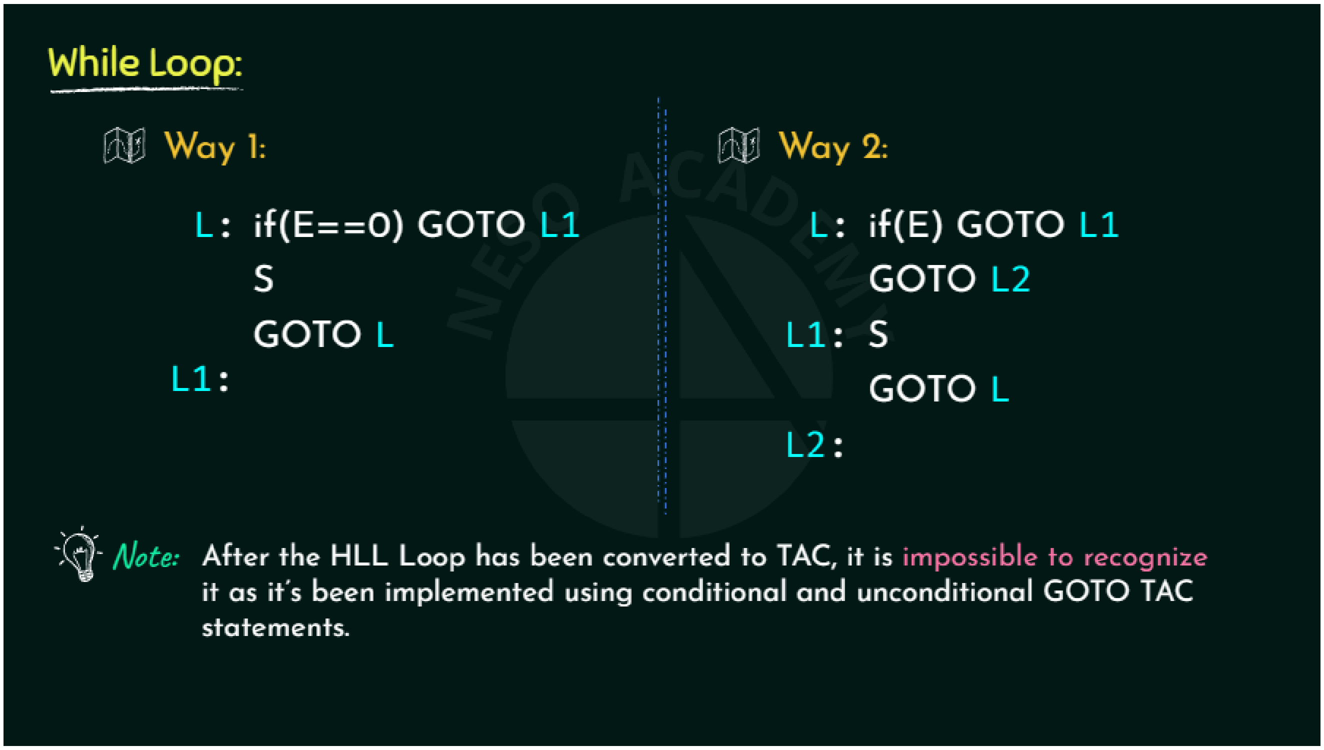
Outcome☆Conversion of For Loop to TAC.

Different forms of TAC:

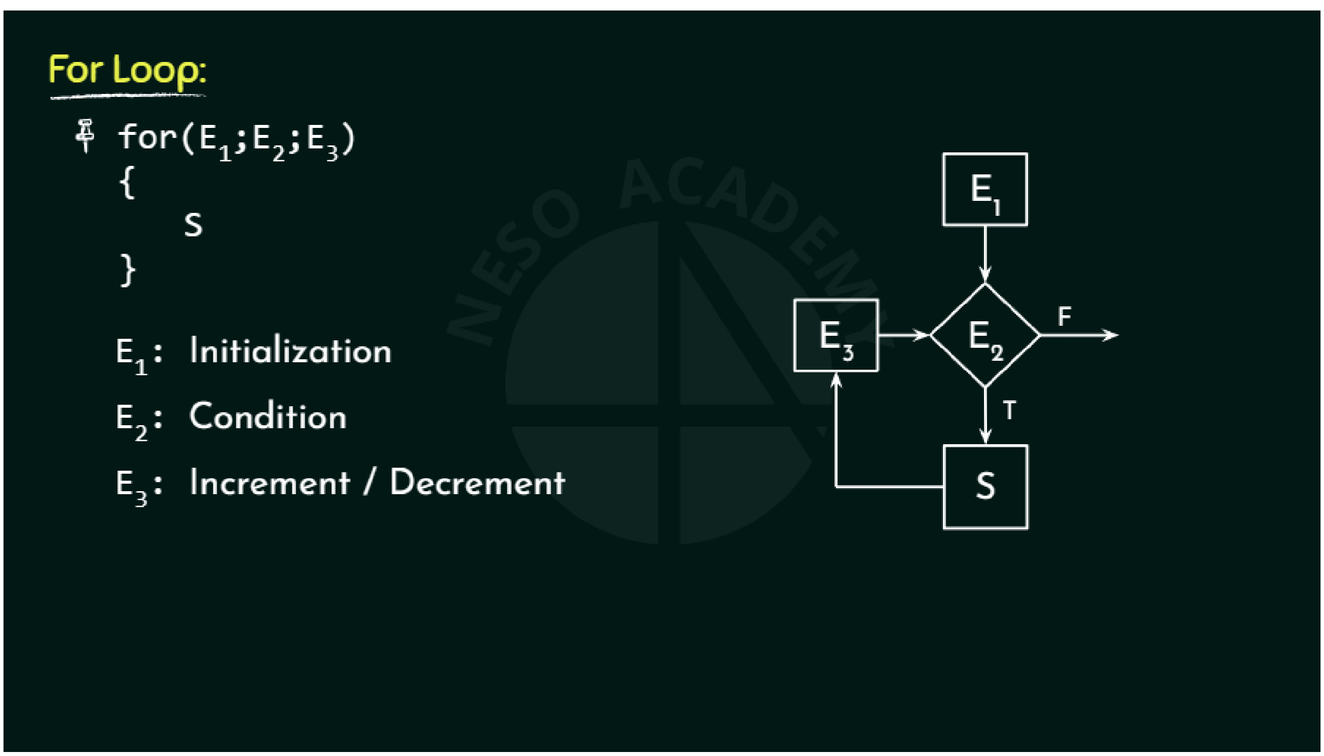
The HLL expressions are converted into the intermediate code using any of the following TAC forms.

	TAC Form	Usage
1.	x = y op z	For Binary Operation & then assignment.
2.	x = op z	For Unary Operation & then assignment.
3.	x = y	For simple assignment.
4.	if (x <rel op> y) GOTO L	Conditional GOTO.
5.	GOTO L	Unconditional GOTO.
6.	A[i] = x y = A[i]	Used for arrays.
7.	x = *p y = &x	'x' is a value pointed by the pointer 'p'. Address of 'x' is stored in 'y'.

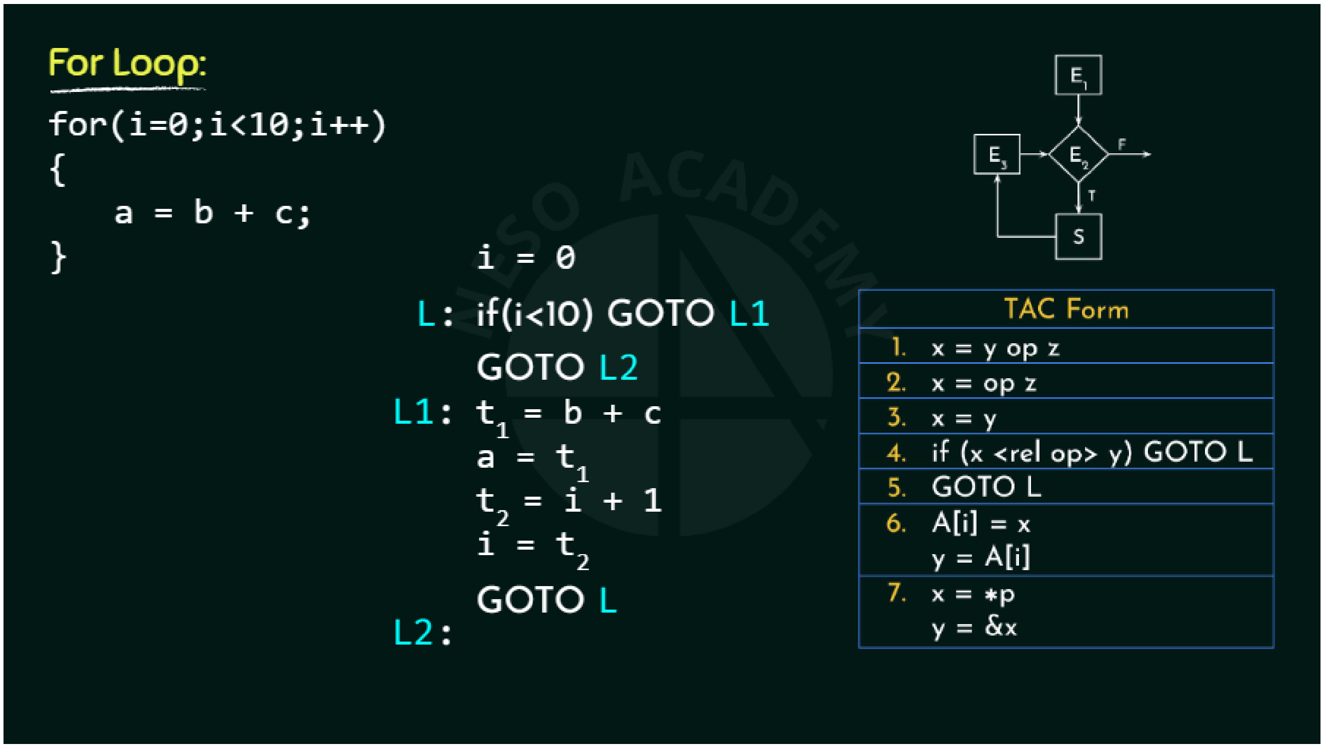
TAC Form Usage Different forms of TAC:The HLL expressions are converted into the intermediate code using any of the following TAC forms.1.x = y op z 2.x = op z 3.x = y4.if (x <rel op> y) GOTO L 5.GOTO L6.A[i] = xy = A[i] 7.x = +py = &x * For Binary Operation & then assignment. For Unary Operation & then assignment. For simple assignment. Conditional GOTO. Unconditional GOTO. Used for arrays. ‘x’ is a value pointed by the pointer ‘p’. Address of ‘x’ is stored in ‘y’.



While Loop:Way 2:if(E) GOTO L1SL:L1:GOTO L2GOTO LL2:Way 1:if(E==0) GOTO L1GOTO LSL:L1:Note:After the HLL Loop has been converted to TAC, it is impossible to recognize it as it’s been implemented using conditional and unconditional GOTO TAC statements.

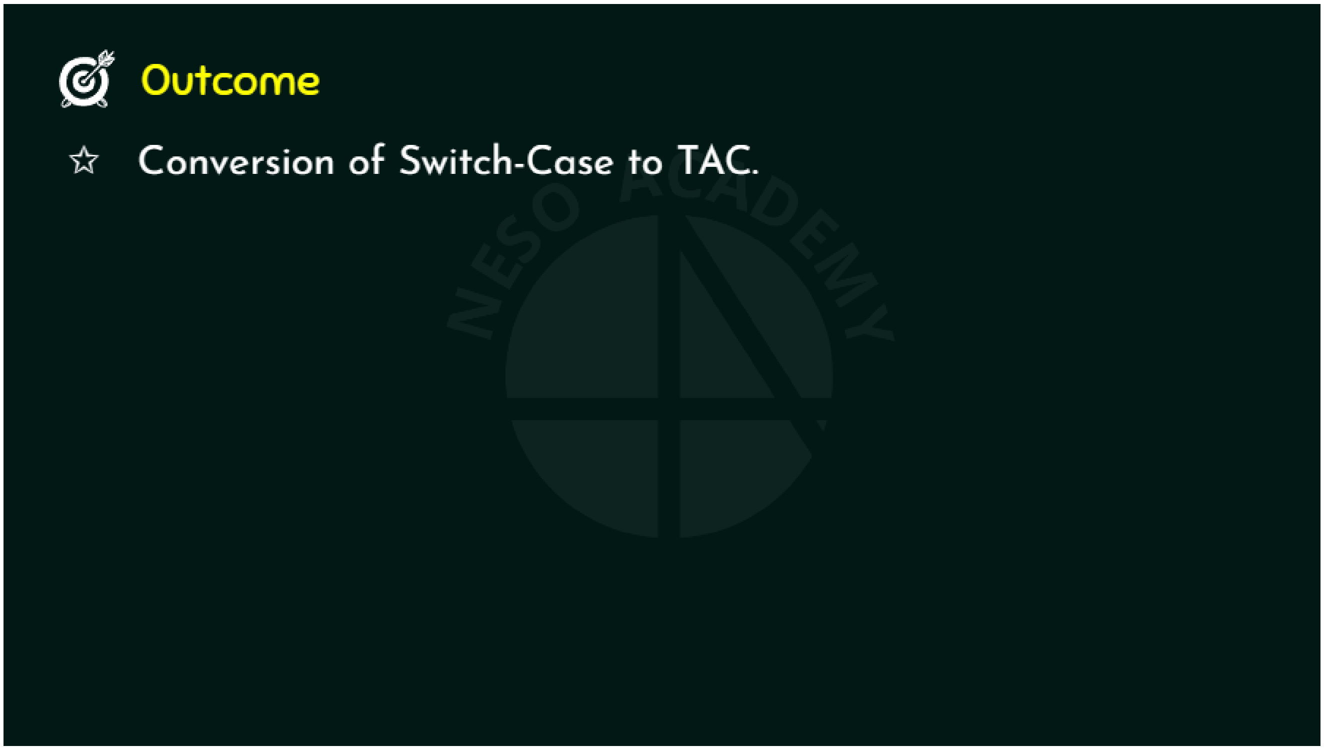


For Loop:for(E1;E2;E3){ S}E2STFE1: InitializationE2: Condition E3: Increment / DecrementE1E3

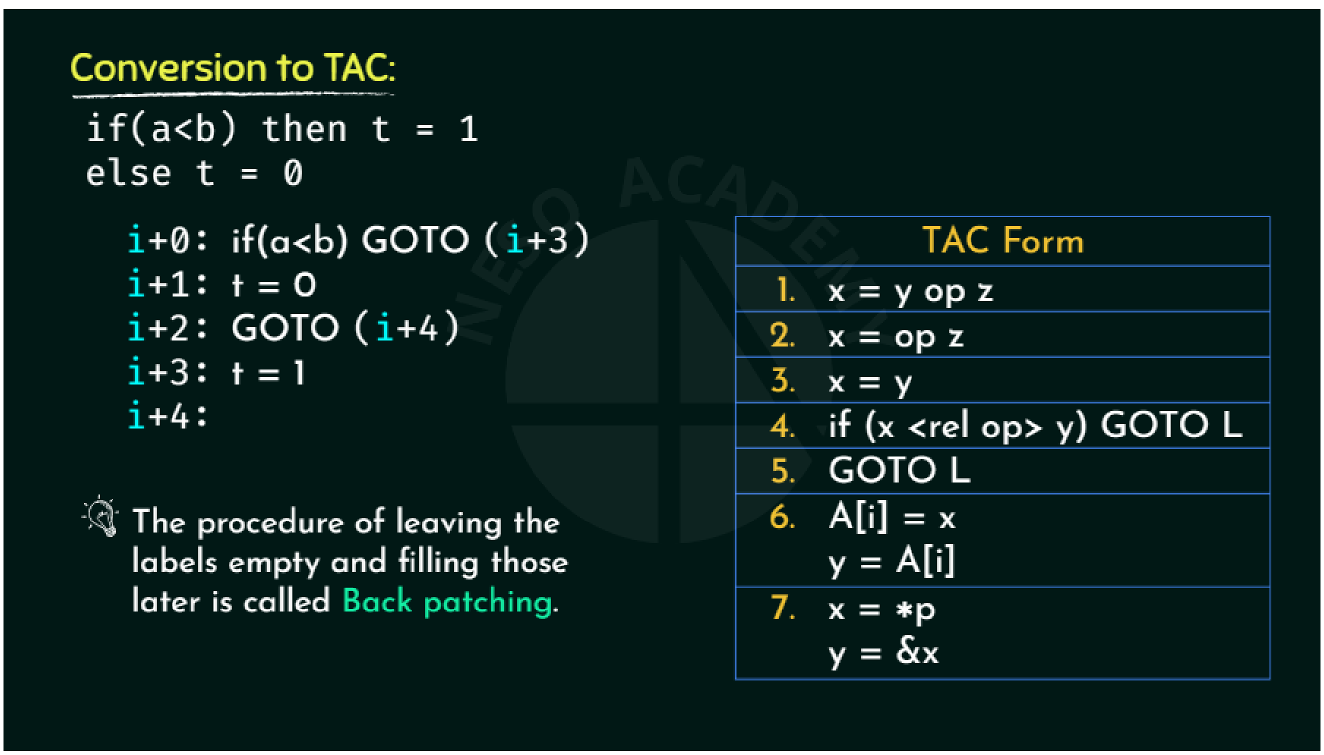


For Loop:for(i=0;i<10;i++){a = b + c;}if(i<10) GOTO L1L:GOTO L2.t1 = b + ca = t1t2 = i + 1i = t2L1:GOTO LL2:i = 0

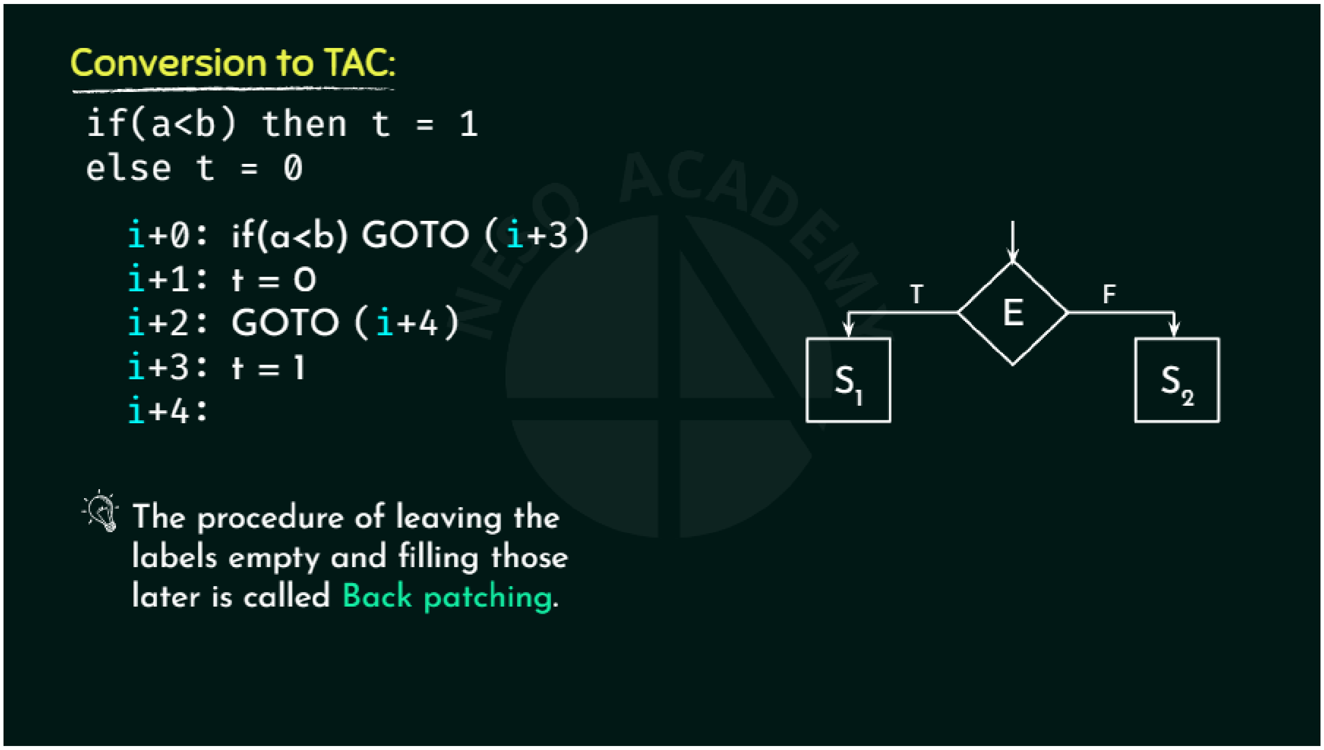
Compiler Design TAC - Multiway Branching



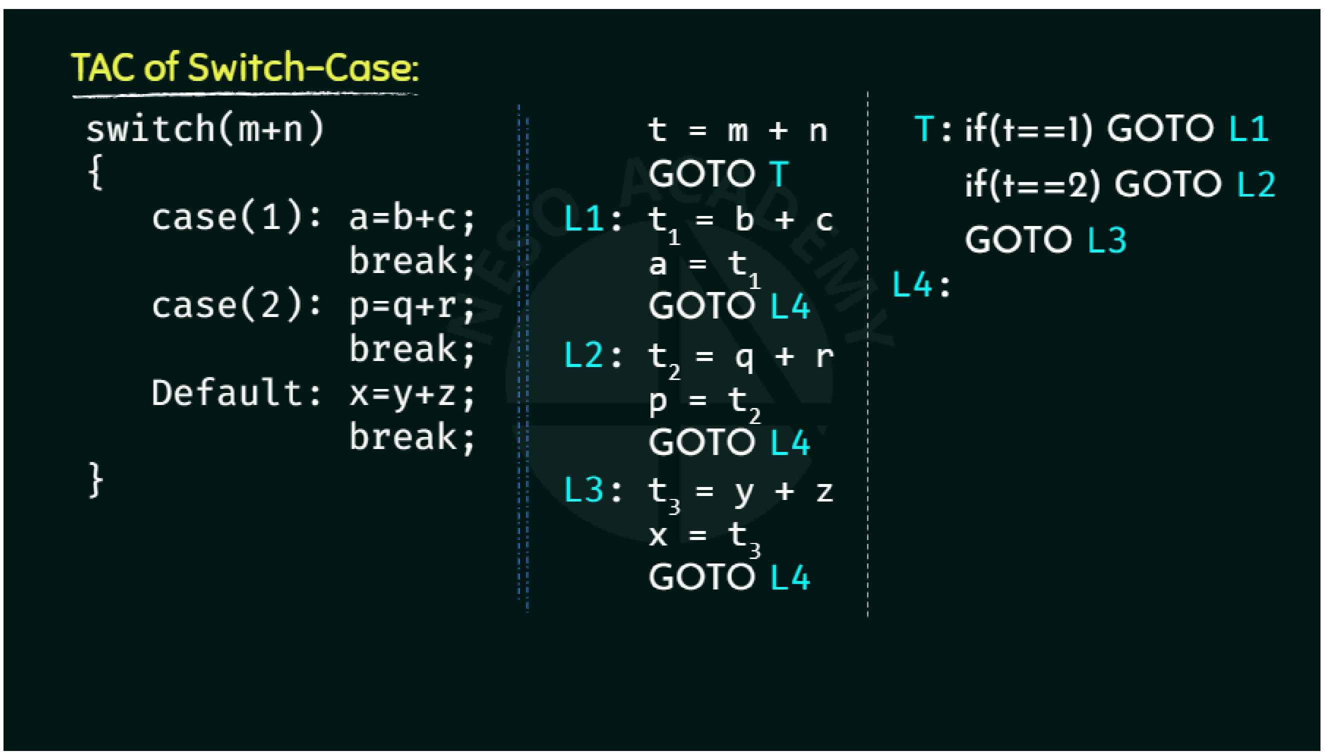
Outcome☆Conversion of Switch-Case to TAC.



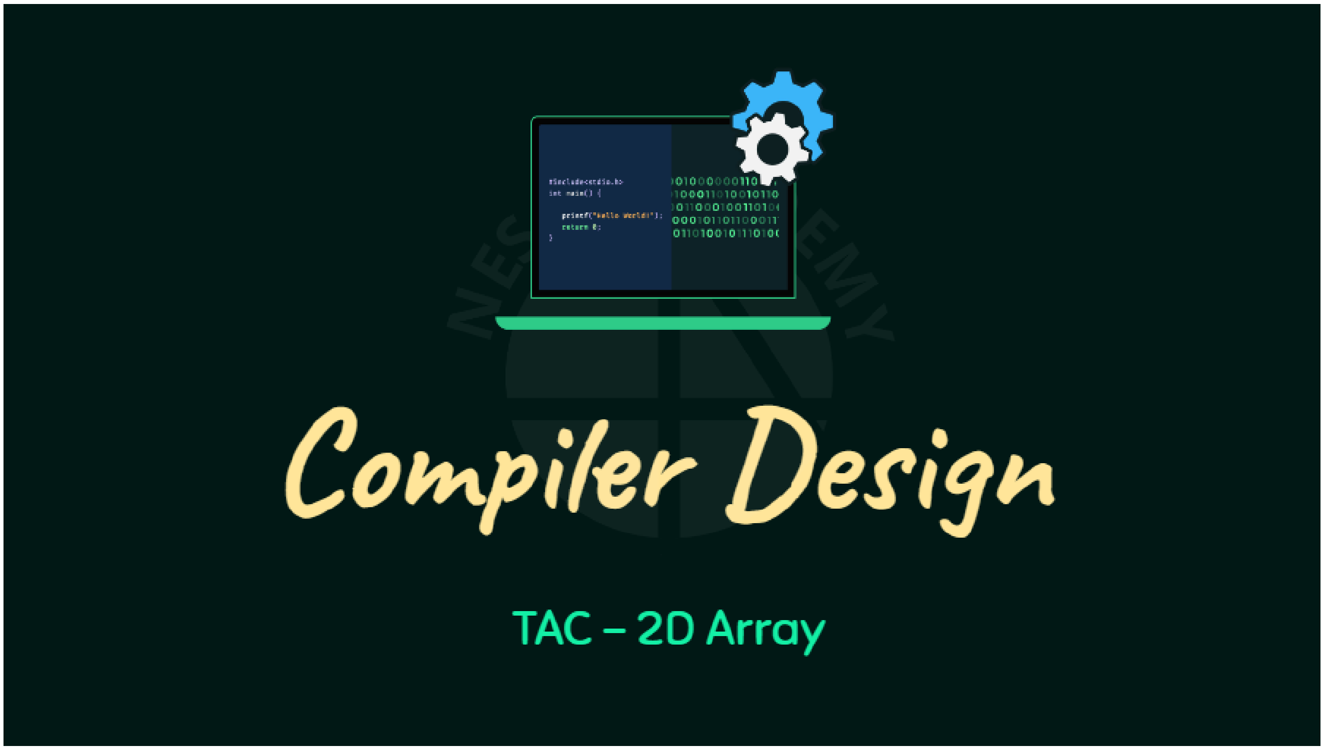
if(a<b) then t = 1else t = 0Conversion to TAC:4.if (x <rel op> y) GOTO L TAC Form 1.x = y op z 2.x = op z 3.x = y5.GOTO L6.A[i] = xy = A[i] 7.x = +py = &x * if(a<b) GOTO (i+3)GOTO (i+4).t = 0t = 1i+0:i+1:i+2:i+3:i+4:The procedure of leaving the labels empty and filling those later is called Back patching.



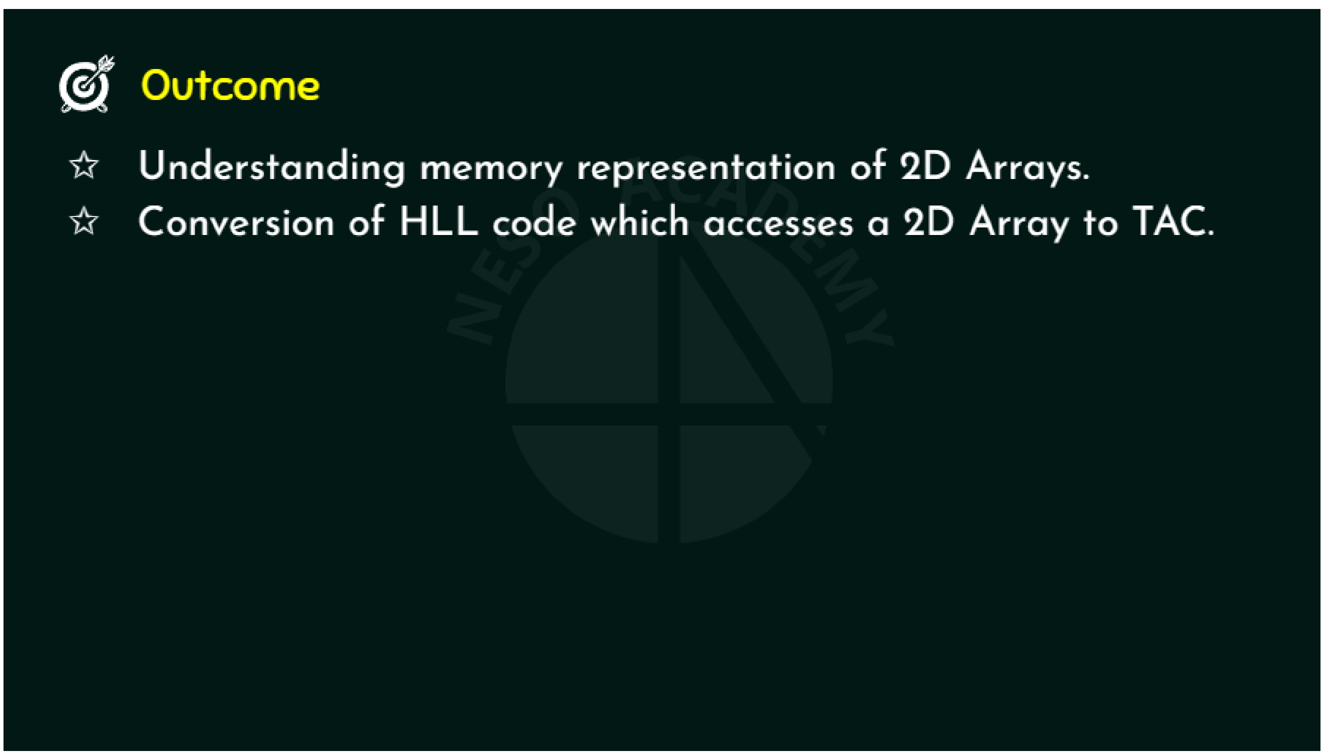
if(a<b) then t = 1else t = 0Conversion to TAC:if(a<b) GOTO (i+3)GOTO (i+4).t = 0t = 1i+0:i+1:i+2:i+3:i+4:The procedure of leaving the labels empty and filling those later is called Back patching.S2TES1F



switch(m+n){case(1): a=b+c; break;case(2): p=q+r;break;Default: x=y+z;break;}TAC of Switch-Case:T:L1:L4:GOTO TGOTO L4.t1 = b + ca = t1GOTO L4t2 = q + rp = t2L2:GOTO L4t3 = y + zx = t3L3:if(t==1) GOTO L1.if(t==2) GOTO L2.GOTO L3t = m + n



Compiler Design TAC - 2D Array



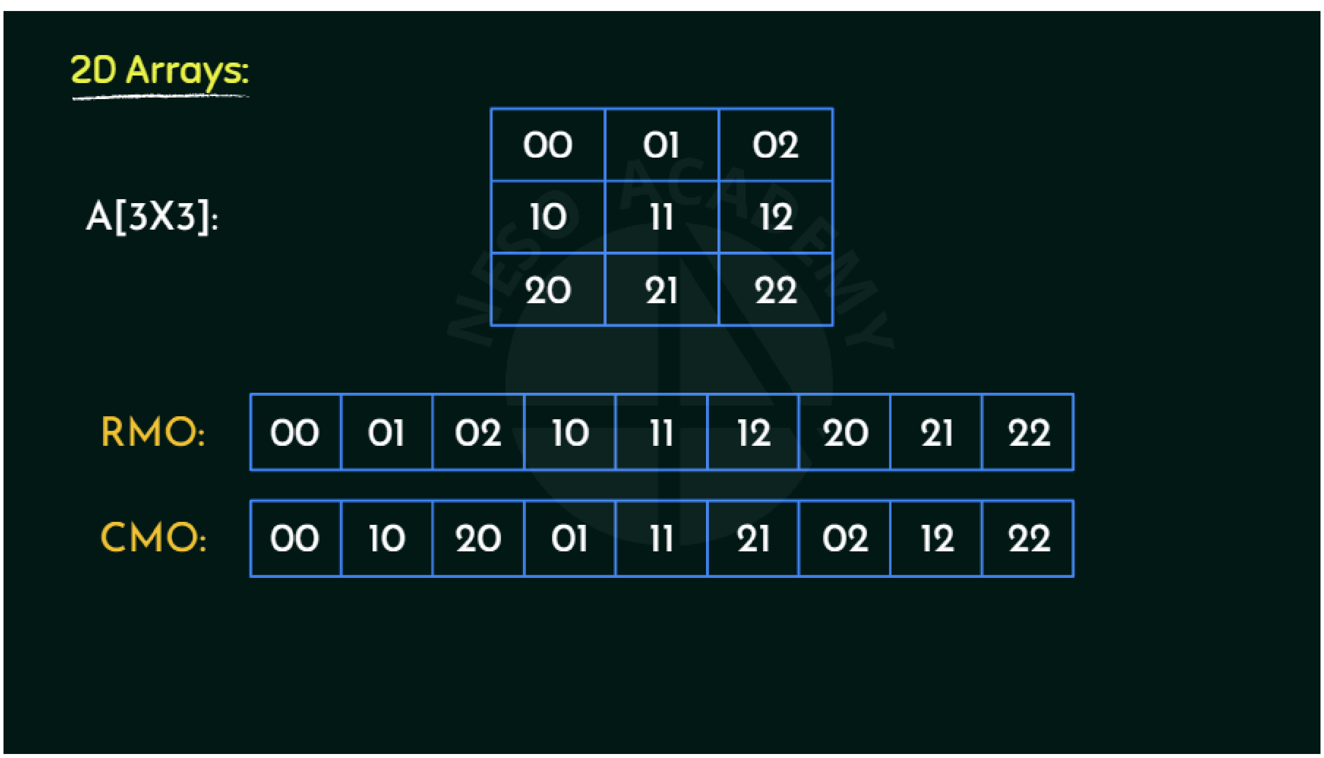
Outcome☆Understanding memory representation of 2D Arrays.☆Conversion of HLL code which accesses a 2D Array to TAC.

Different forms of TAC:

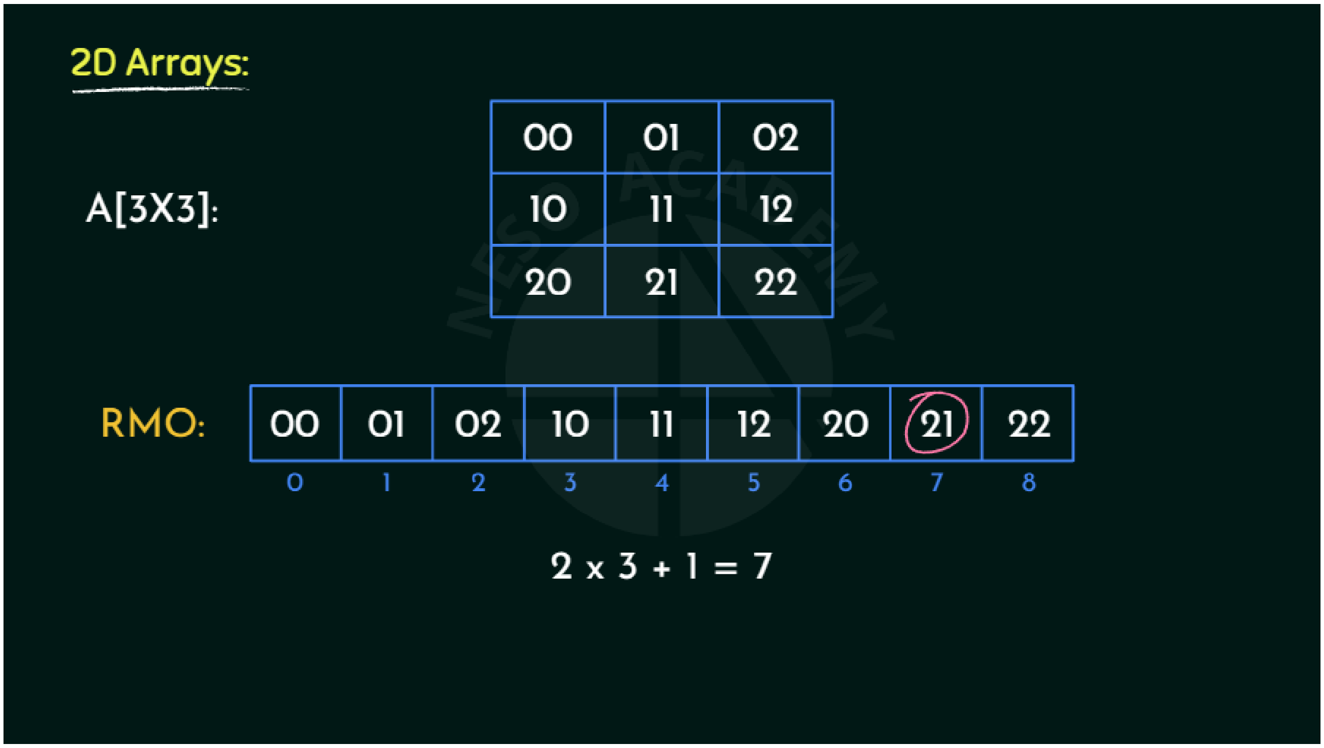
The HLL expressions are converted into the intermediate code using any of the following TAC forms.

TAC Form	Usage
1. x = y op z	For Binary Operation & then assignment.
2. x = op z	For Unary Operation & then assignment.
3. x = y	For simple assignment.
4. if (x <rel op> y) GOTO L	Conditional GOTO.
5. GOTO L	Unconditional GOTO.
6. A[i] = x y = A[i]	Used for arrays.
7. x = *p y = &x	'x' is a value pointed by the pointer 'p'. Address of 'x' is stored in 'y'.

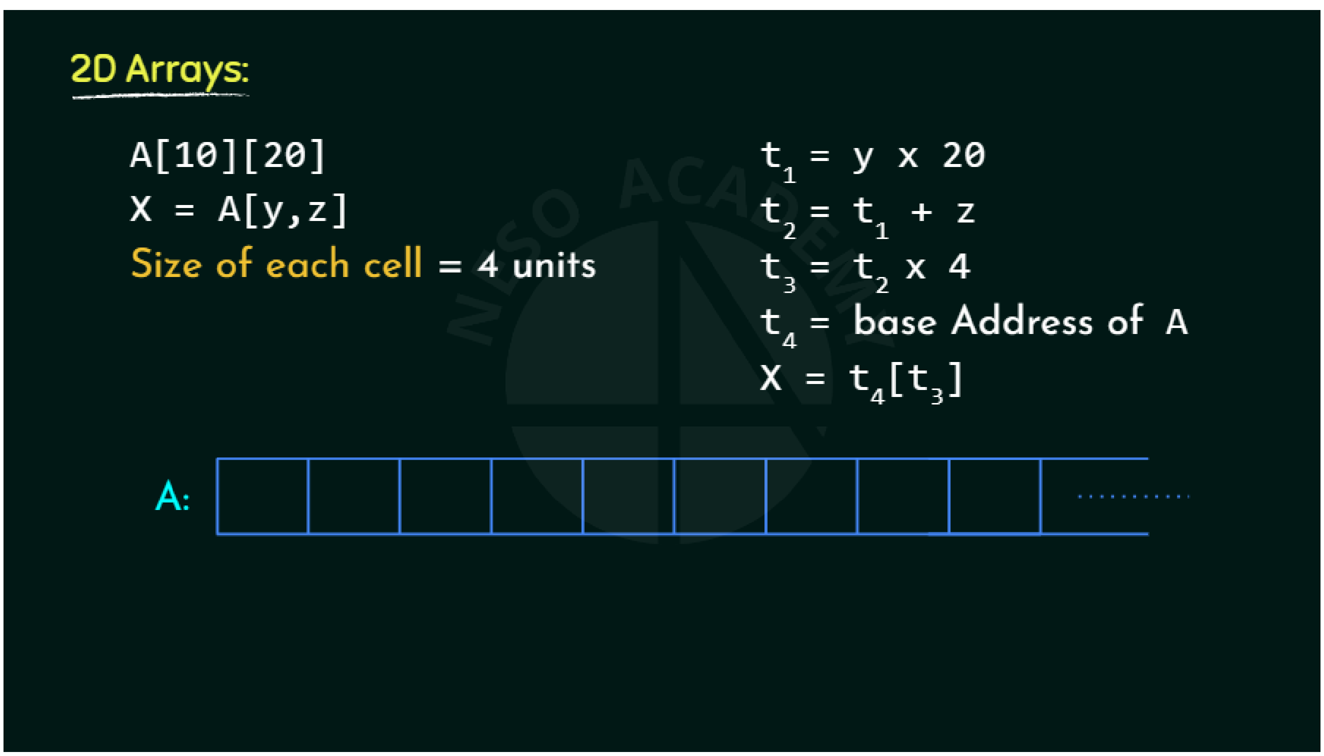
TAC Form Usage Different forms of TAC:The HLL expressions are converted into the intermediate code using any of the following TAC forms.1.x = y op z 2.x = op z 3.x = y4.if (x <rel op> y) GOTO L 5.GOTO L6.A[i] = xy = A[i] 7.x = +py = &x * For Binary Operation & then assignment. For Unary Operation & then assignment. For simple assignment. Conditional GOTO. Unconditional GOTO. Used for arrays. 'x' is a value pointed by the pointer 'p'. Address of 'x' is stored in 'y'.



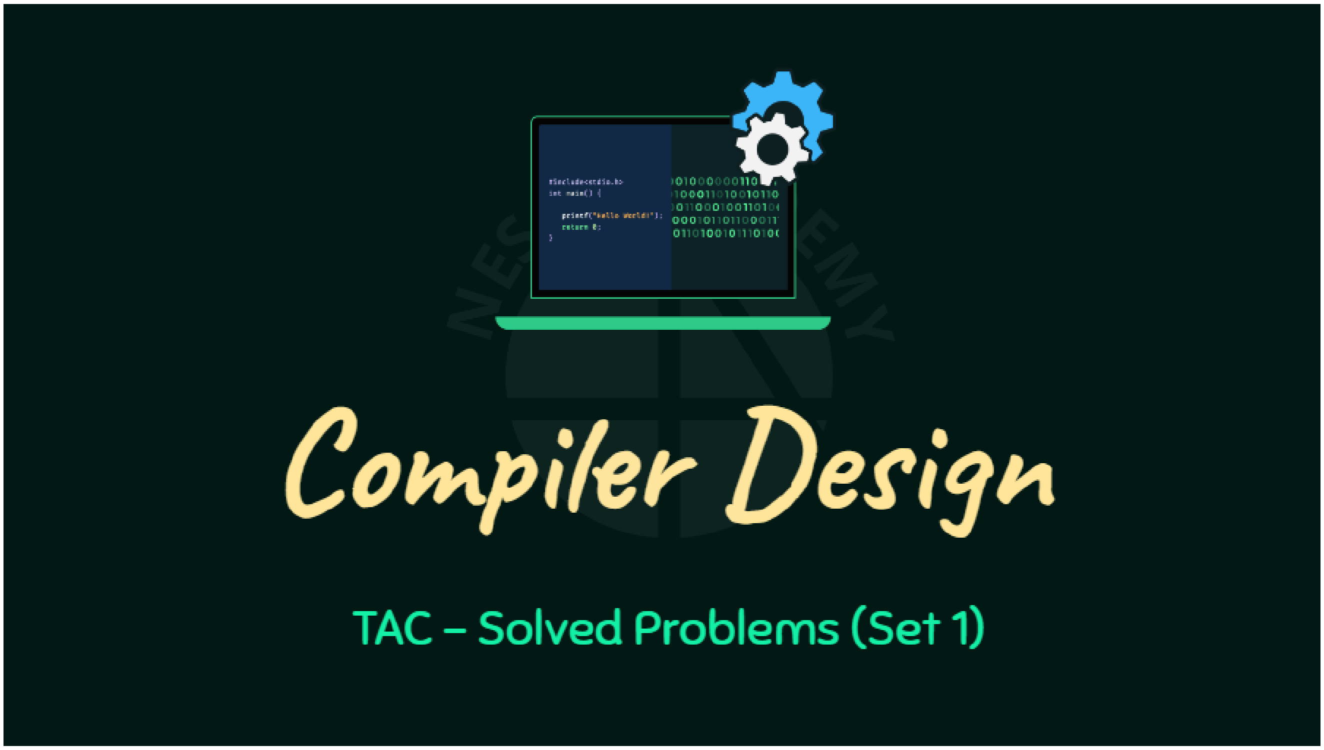
2000100200012D
Arrays:A[3X3]:000102101112202122101112202122011121021222RMO:CMO:



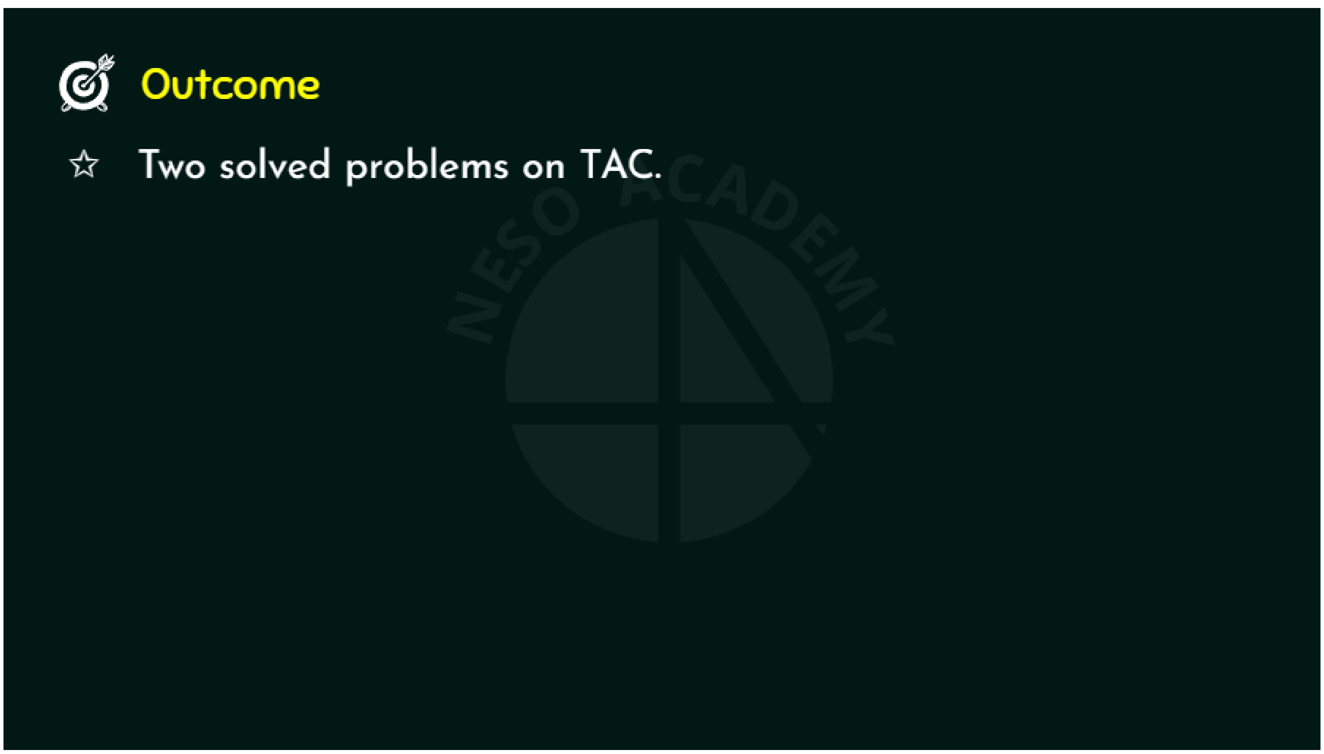
= 72D Arrays:A[3X3]:000102101112202122020001101112202122RMO:2 x 3+ 1 201345678



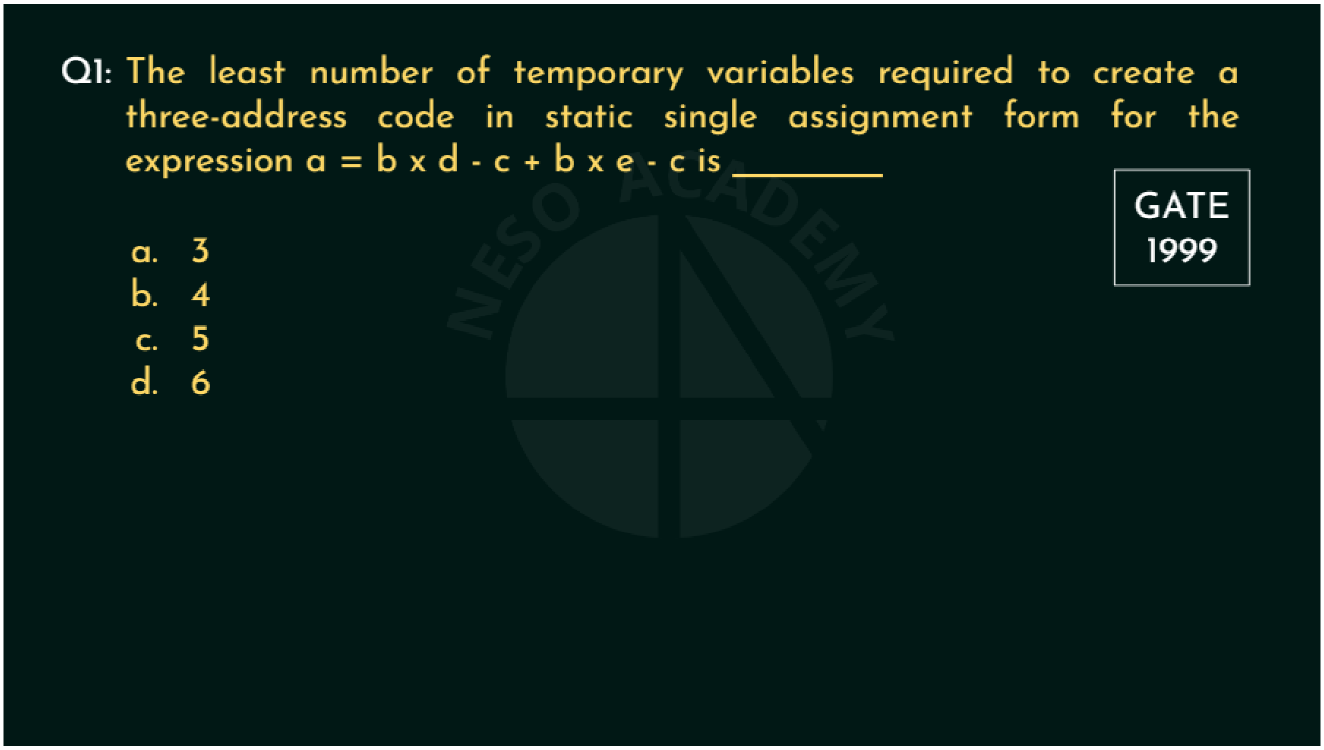
2D Arrays:A[10][20]X = A[y,z]Size of each cell = 4 unitst1= y x 20t2 = t1 + zt3 = t2 x 4t4= base Address of AX = t4[t3]A:



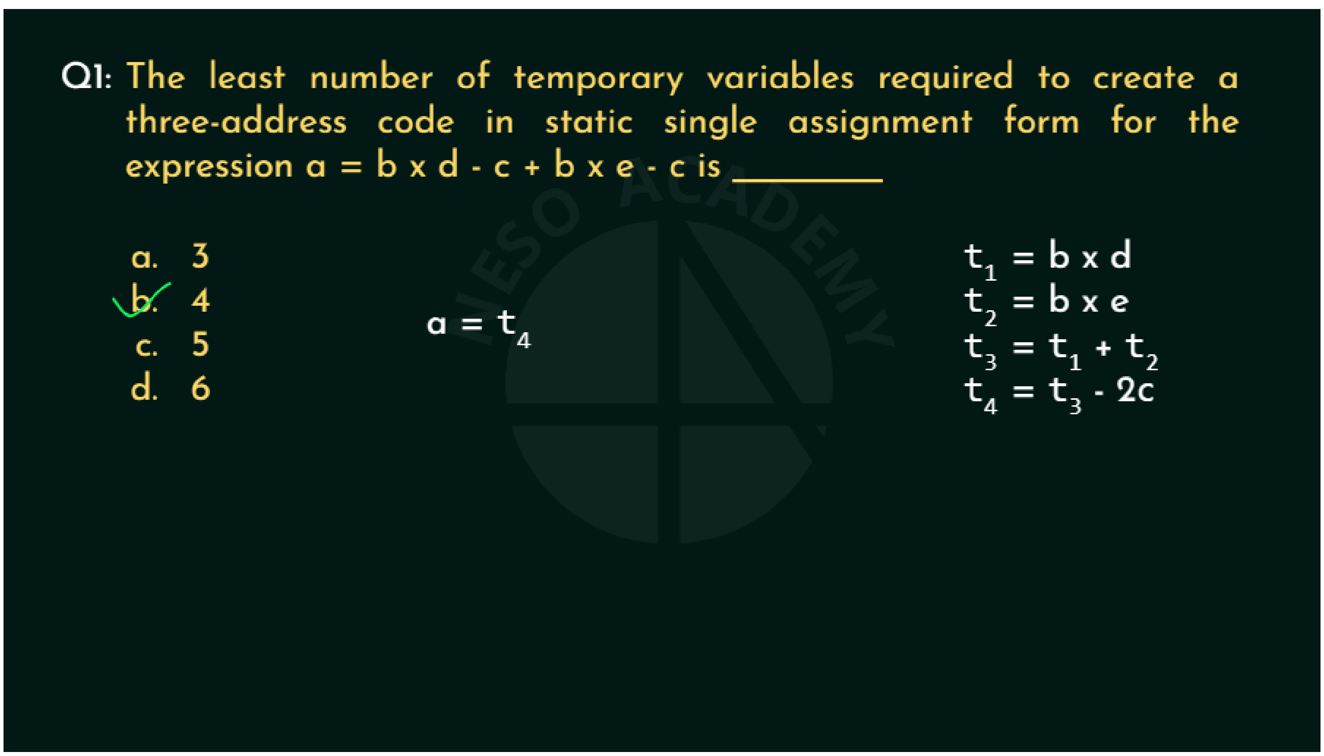
Compiler Design TAC - Solved Problems (Set 1)



Outcome☆Two solved problems on TAC.



Q1:The least number of temporary variables required to create a three-address code in static single assignment form for the expression a = b x d - c + b x e - c is .a.3b.4c.5d.6GATE 1999



Q1:The least number of temporary variables required to create a three-address code in static single assignment form for the expression a = b x d - c + b x e - c is .a.3b.4c.5d.6a = t4t1 = b x dt2 = b x et3 = t1 + t2t4 = t3 - 2c

Q2: In a simplified computer the instructions are:

`OP` R_j, R_i	Performs R_j `OP` R_i and stores the result in register R_j.
`OP m,` R_i	Performs `Val` `OP` R_i and stores the result in register R_i. `Val` is the content of memory location m.
`MOV m,` R_i	Moves the content of memory location `m` to register R_i.
`MOV` R_i`, m`	Moves the content of register R_i to memory location m.

The computer has only 2 registers and OP is either ADD or SUB.

Consider the following basic block:

$t_1 = a + b$

$t_2 = c + d$

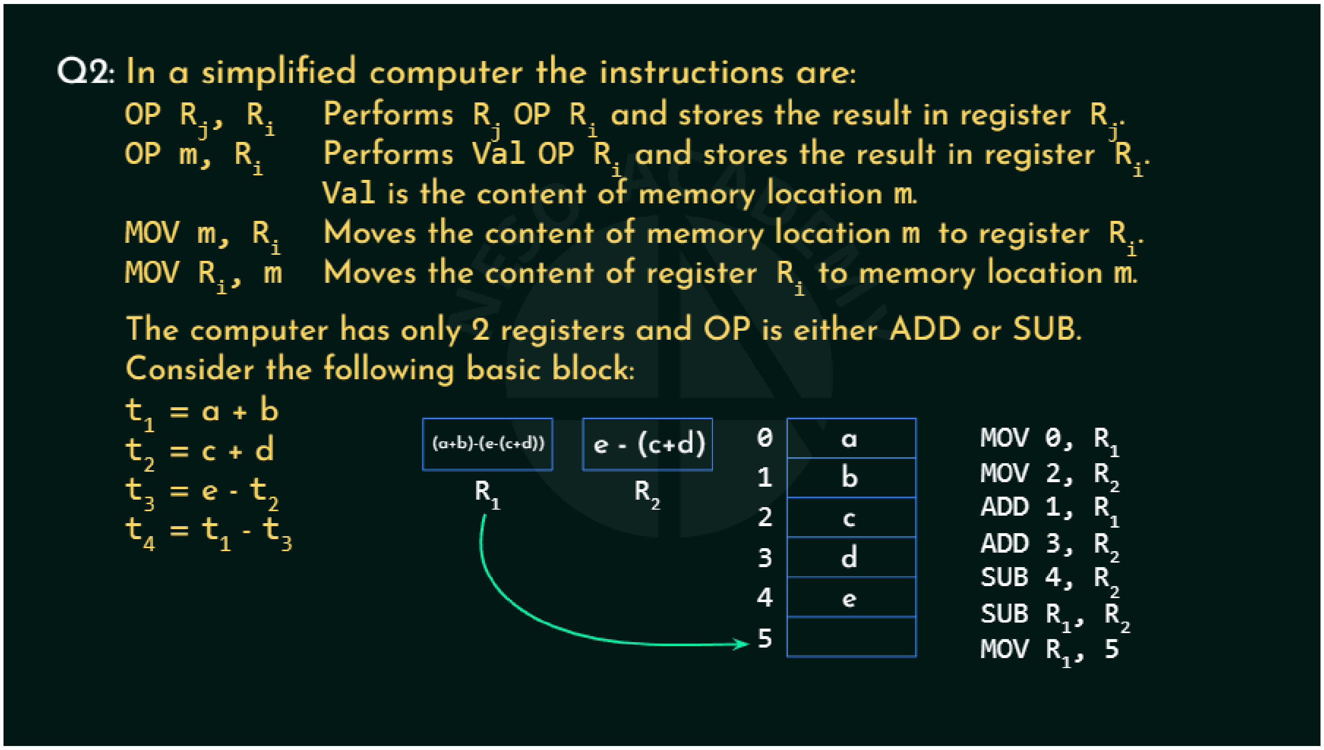
$t_3 = e - t_2$

$t_4 = t_1 - t_3$

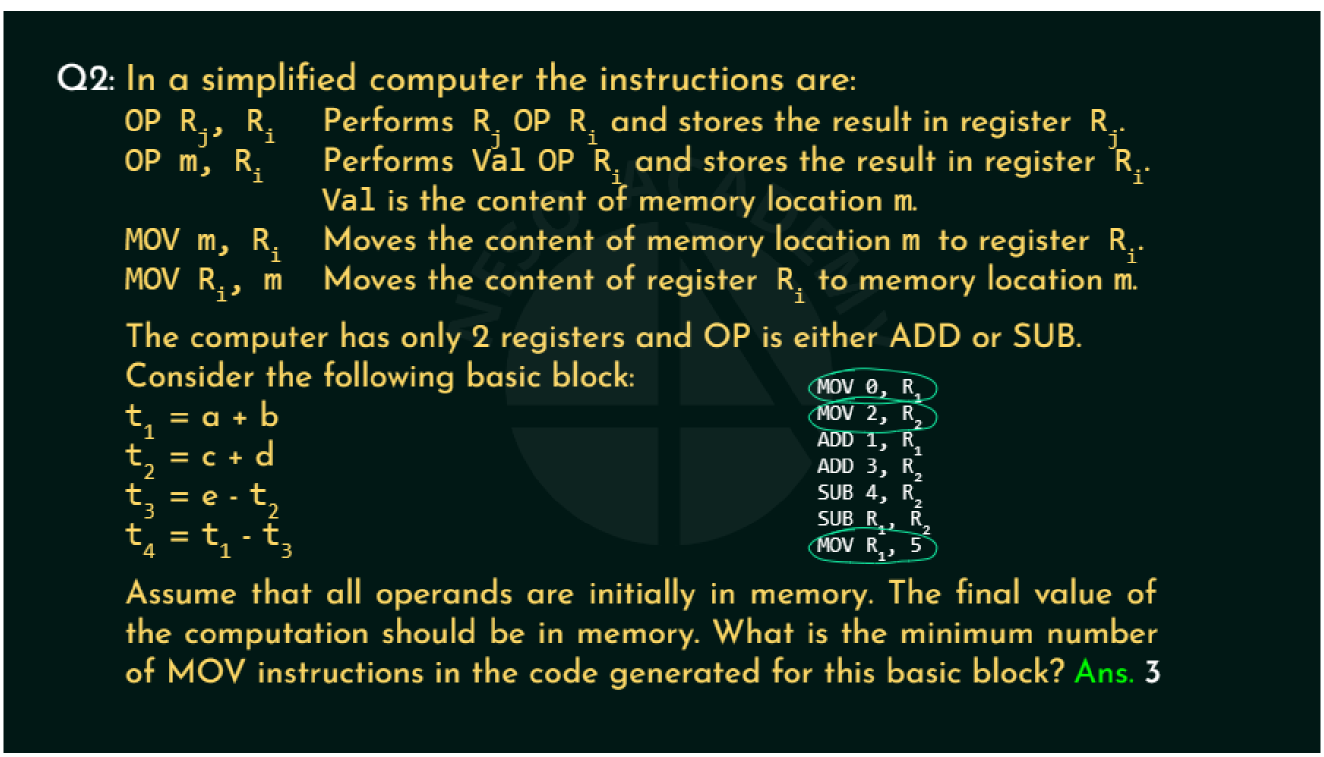
GATE 2007

Assume that all operands are initially in memory. The final value of the computation should be in memory. What is the minimum number of MOV instructions in the code generated for this basic block?

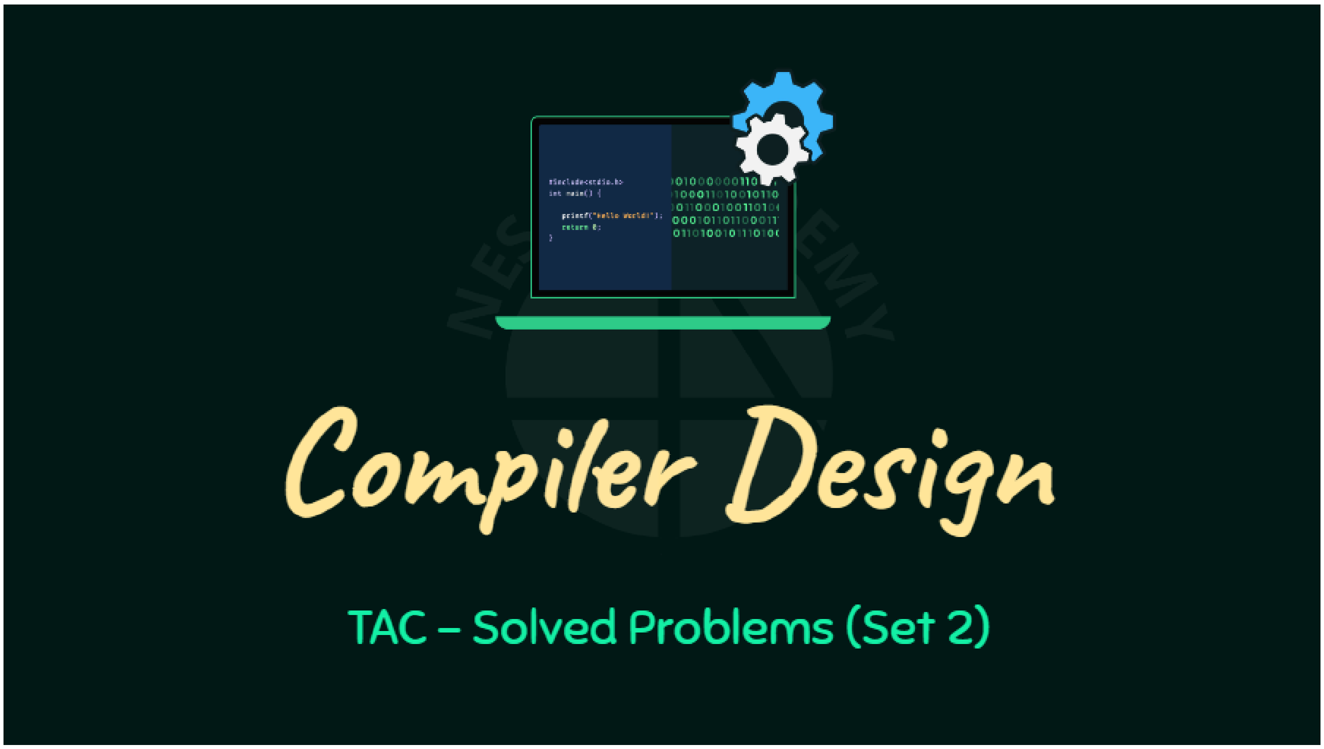
In a simplified computer the instructions are:OP Rj, RiOP m, RiMOV m, RiMOV Ri, mThe computer has only 2 registers and OP is either ADD or SUB.Consider the following basic block:t1 = a + bt2 = c + dt3 = e - t2t4 = t1 - t3Assume that all operands are initially in memory. The final value of the computation should be in memory. What is the minimum number of MOV instructions in the code generated for this basic block?Q2:Performs Rj OP Ri and stores the result in register Rj.Performs Val OP Ri and stores the result in register Ri. Val is the content of memory location m.Moves the content of memory location m to register Ri.Moves the content of register Ri to memory location m.GATE 2007



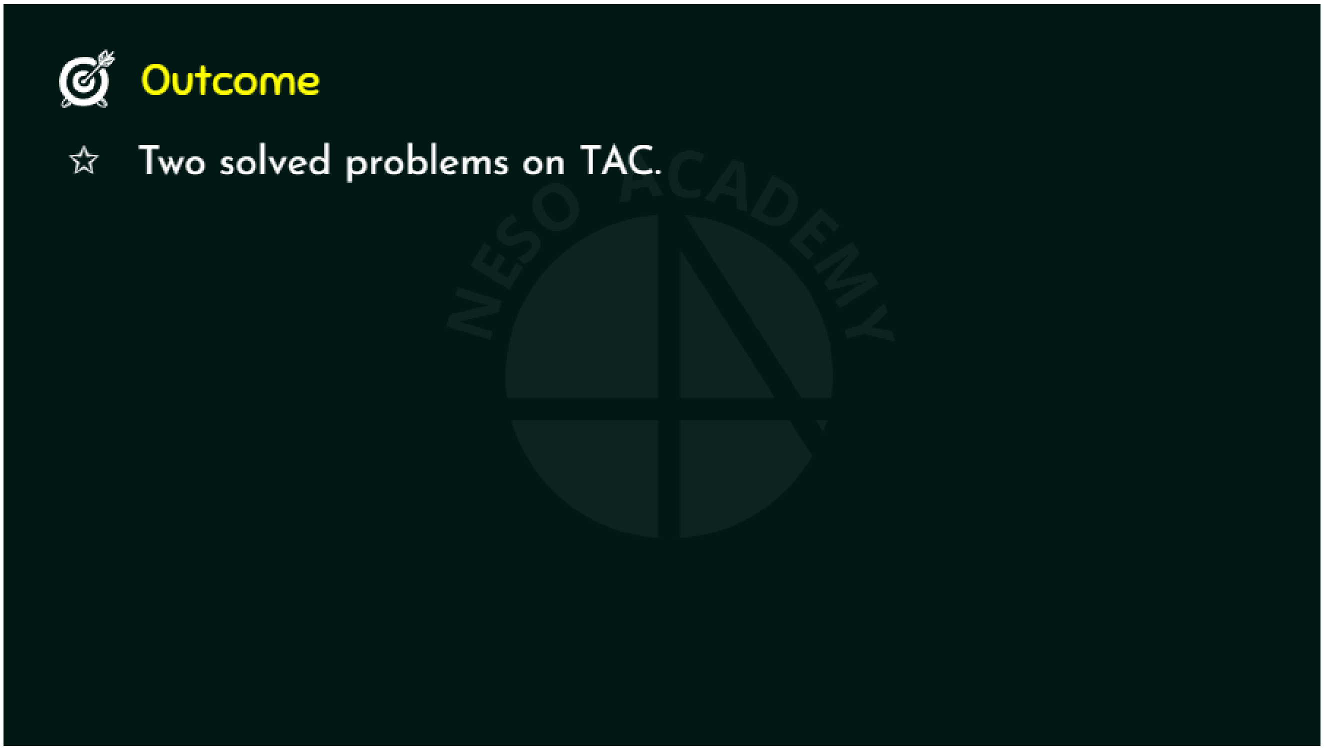
051234R1R2In a simplified computer the instructions are:OP Rj, RiOP m, RiMOV m, RiMOV Ri, mThe computer has only 2 registers and OP is either ADD or SUB.Consider the following basic block:Q2:Performs Rj OP Ri and stores the result in register Rj.Performs Val OP Ri and stores the result in register Ri. Val is the content of memory location m.Moves the content of memory location m to register Ri.Moves the content of register Ri to memory location m.t1 = a + bt2 = c + dt3 = e - t2t4 = t1 - t3abcde(a+b)-(e-(c+d))e - (c+d)MOV R1, 5SUB R1, R2SUB 4, R2ADD 3, R2MOV 0, R1MOV 2, R2ADD 1, R1



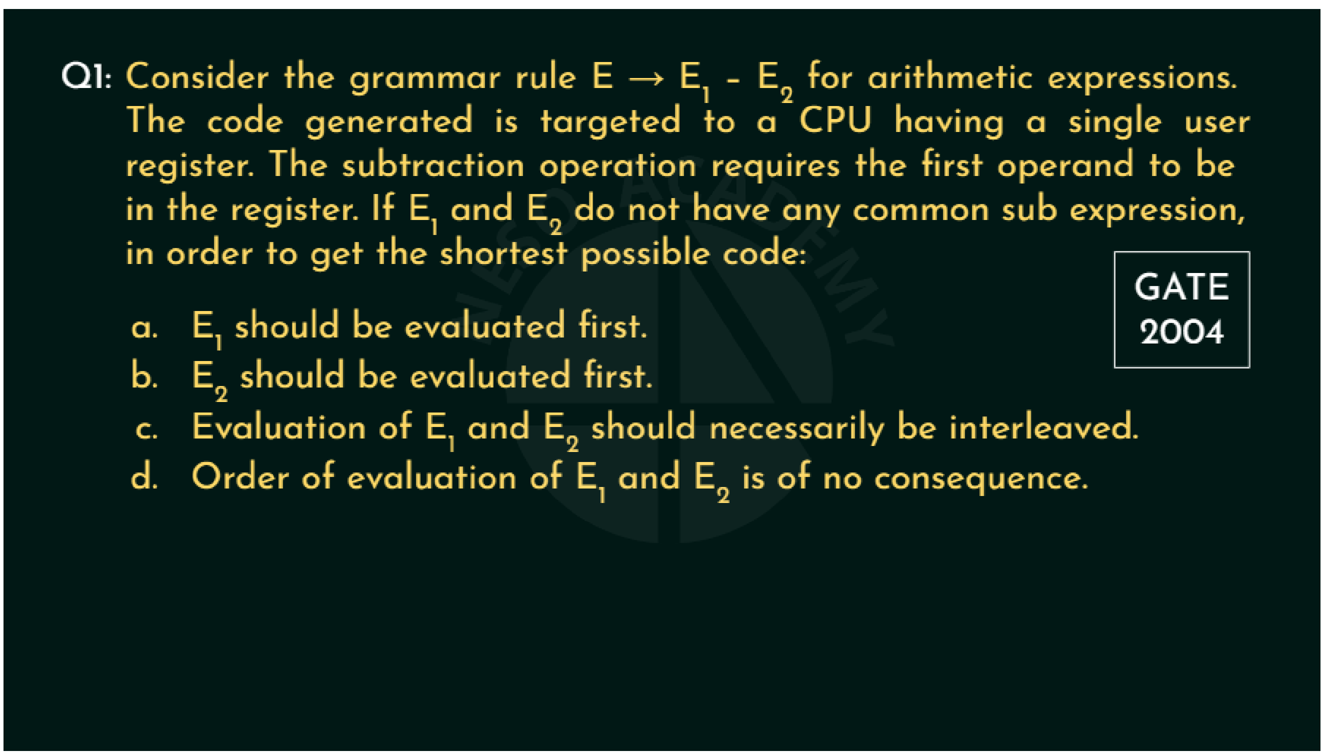
In a simplified computer the instructions are:OP Rj, RiOP m, RiMOV m, RiMOV Ri, mThe computer has only 2 registers and OP is either ADD or SUB.Consider the following basic block:t1 = a + bt2 = c + dt3 = e - t2t4 = t1 - t3Assume that all operands are initially in memory. The final value of the computation should be in memory. What is the minimum number of MOV instructions in the code generated for this basic block?Q2:Performs Rj OP Ri and stores the result in register Rj.Performs Val OP Ri and stores the result in register Ri. Val is the content of memory location m.Moves the content of memory location m to register Ri.Moves the content of register Ri to memory location m.MOV 0, R1MOV 2, R2ADD 1, R1ADD 3, R2SUB 4, R2SUB R1, R2MOV R1, 5Ans. 3



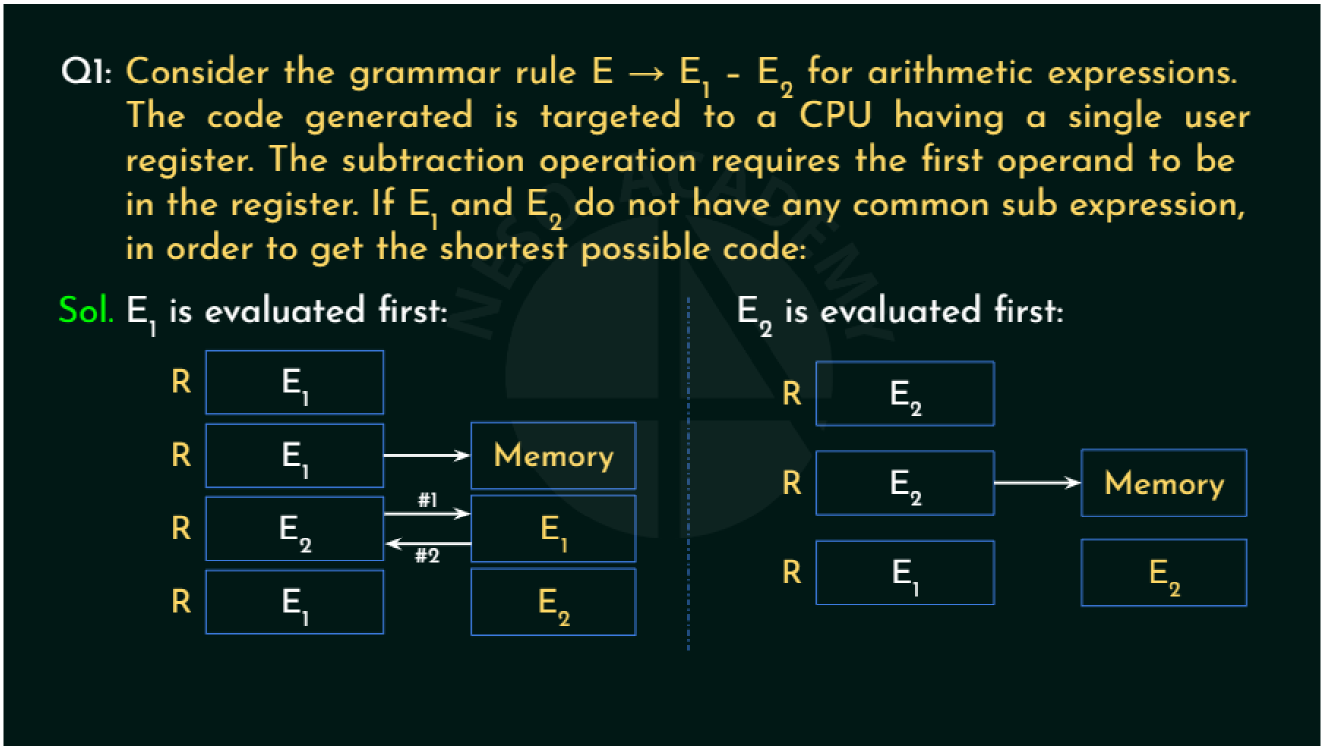
Compiler Design TAC - Solved Problems (Set 2)



Outcome☆Two solved problems on TAC.



Q1:Consider the grammar rule E → E1 – E2 for arith metic expressions. The code generated is targeted to a CPU having a single user register. The sub traction operation requires the first operand to be in the register. If E1 and E2 do not have any com mon sub expression, in order to get the shortest possible code:a.E1 should be evaluated first.b.E2 should be evaluated first.c.Evaluation of E1 and E2 should necessarily be interleaved.d.Order of evaluation of E1 and E2 is of no consequence.GATE 2004



Q1:Consider the grammar rule E → E1 – E2 for arith metic expressions. The code generated is targeted to a CPU having a single user register. The sub traction operation requires the first operand to be in the register. If E1 and E2 do not have any com mon sub expression, in order to get the shortest possible code:E1 is evaluated first:RE1 RE1 RE2 E1RE1 E2Memory#1 #2 E2 is evaluated first:RE2 RE2 MemoryRE1 E2Sol.

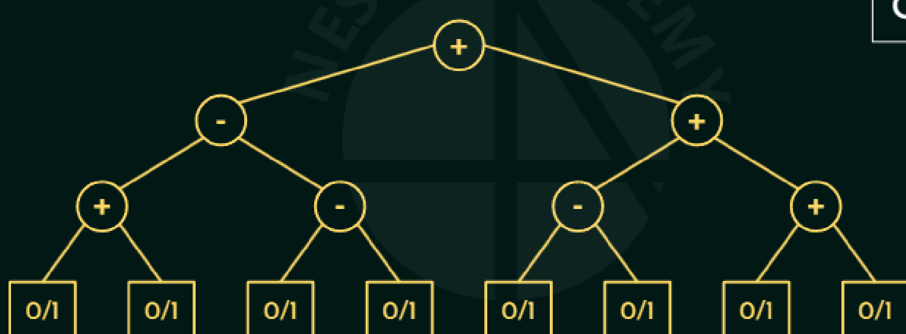
Q1: Consider the grammar rule $E \rightarrow E_1 - E_2$ for arithmetic expressions. The code generated is targeted to a CPU having a single user register. The subtraction operation requires the first operand to be in the register. If E_1 and E_2 do not have any common sub expression, in order to get the shortest possible code:

a. E_1 should be evaluated first.
b. E_2 should be evaluated first.
c. Evaluation of E_1 and E_2 should necessarily be interleaved.
d. Order of evaluation of E_1 and E_2 is of no consequence.

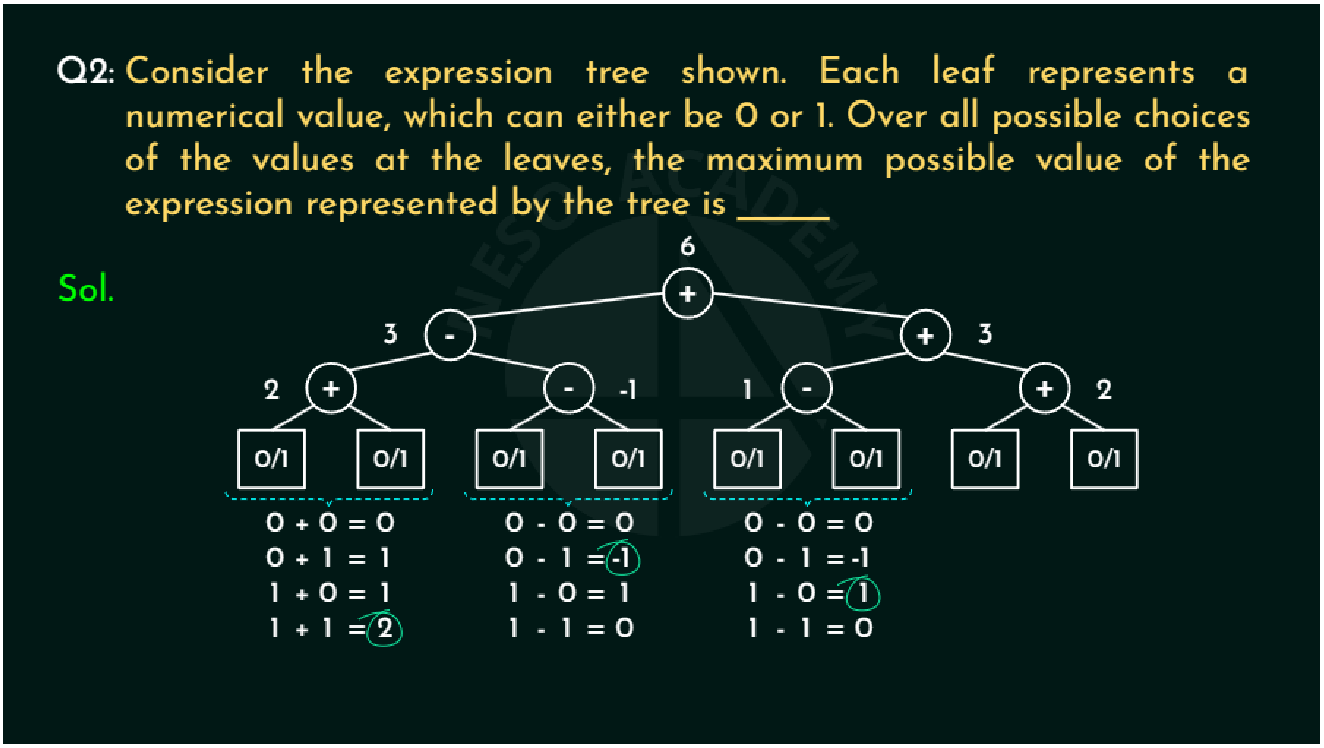
Q1:Consider the grammar rule E → E1 – E2 for arith metic expressions. The code generated is targeted to a CPU having a single user register. The sub traction operation requires the first operand to be in the register. If E1 and E2 do not have any com mon sub expression, in order to get the shortest possible code:a.E1 should be evaluated first.b.E2 should be evaluated first.c.Evaluation of E1 and E2 should necessarily be interleaved.d.Order of evaluation of E1 and E2 is of no consequence.

Q2: Consider the expression tree shown. Each leaf represents a numerical value, which can either be 0 or 1. Over all possible choices of the values at the leaves, the maximum possible value of the expression represented by the tree is ____

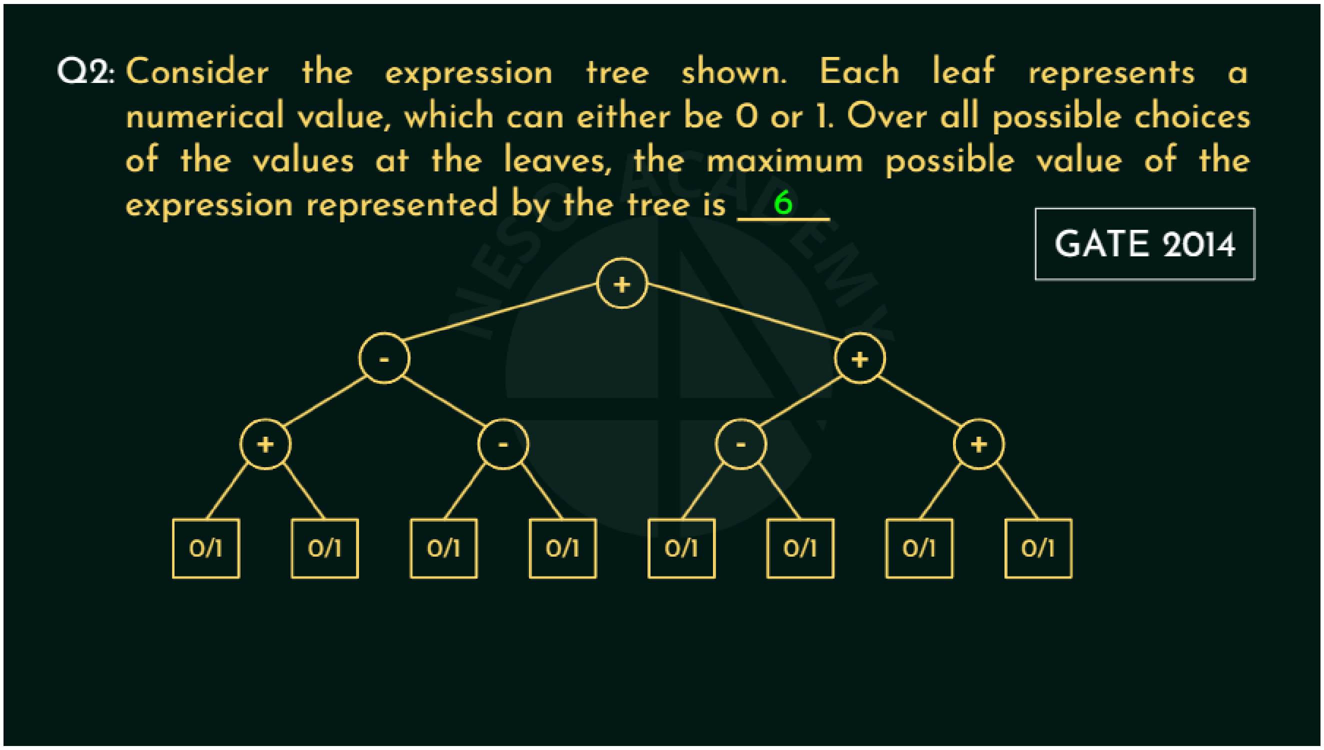
GATE 2014



Q2:Consider the expression tree shown. Each leaf represents a numerical value, which can either be 0 or 1. Over all possible choices of the values at the leaves, the maximum possible value of the expression represented by the tree is .GATE 2014+--+-
++0/10/10/10/10/10/10/10/1



0 = - 0 0 0= -11-1= -1 0 1= -0 1-1= 2= Q2:Consider the expression tree shown. Each leaf represents a numerical value, which can either be 0 or 1. Over all possible choices of the values at the leaves, the maximum possible value of the expression represented by the tree is .+--+-++0/10/10/10/10/10/10/10/10 = 1= 1= + 0 0 + 11+ 1 0 + 0 10 = 0= 1= - 0 0 -11-1 0 -0 12-121336Sol.



Q2:Consider the expression tree shown. Each leaf represents a numerical value, which can either be 0 or 1. Over all possible choices of the values at the leaves, the maximum possible value of the expression represented by the tree is .GATE 2014+--+-
++0/10/10/10/10/10/10/10/16