__/__/__

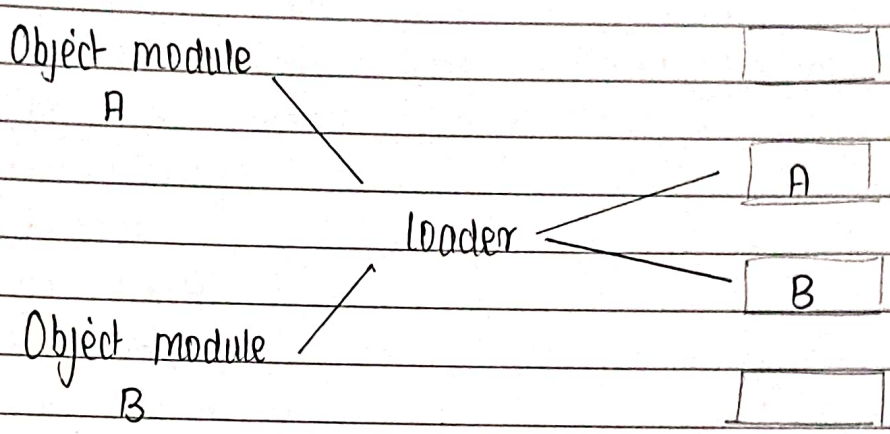
SYSTEM PROGRAMMING AND COMPILER CONSTRUCTION

ASSIGNMENT NO: 02

Q1]. Explain the functions of loaders.

Ans: - Assemblers and compilers are used to convert source code to object code.

- The loader will accept that object code, make it ready for execution and helps to execute.



Program modules A and B are loaded in memory after linking

- Loader performs its task via four functions :

1. Allocation :

In order to allocate memory to the program, the Loader allocates the memory on the basis of the size of the program, this is known as allocation.

The loader gives the space in memory where the object program will be loaded for execution.

2. Linking :
The linker resolves the symbolic reference code or data between the object modules by allocating all of the user sub-routine and library sub-routine addresses.
This process is known as linking.
In any language, a program written has a function, it can be user-defined or can be a library function.

3. Relocation :
There are some address-dependent locations in the program, and these address constants must be modified to fit the available space, this can be done by loader and this is known as relocation.
In order to allow the object program to be loaded at a different address than the one initially supplied, the loader modifies the object program by modifying specific instructions.

4. Loading :
The loader loads the program into the main memory for execution of that program.
It loads machine instruction and data of related programs and sub-routines into the main memory, this process is known as loading.
The loader performs loading; hence, the assembler must provide the loader with the object program.

Q2). Explain different types of loaders in detail.

Ans:- Loaders and system programs responsible for loading executable code into memory for execution.
- They play a vital role in memory management and program execution by linking, relocating and preparing the program for execution.
- The types of loaders are:

(i) Absolute loader:
An absolute loader is a simple type of loader that loads the program into a pre-defined specified memory location.
The object code contains absolute addresses meaning the program must be placed at a fixed location for execution.

- Functioning:
1. Reads the program's object code and transfers it to the specified memory location.
2. No address modification or relocation is performed.
3. The program can only run from its specified memory address.

- Advantages:
Simple and fast since it doesn't involve relocation or linking.

- Disadvantages:
1. Inflexible as it requires the program to be loaded at a fixed address.
2. Inefficient for multitasking or dynamic memory allocation.

- Use Case:
1. Used in early computing systems.
2. Bootstrap loaders are a form of absolute loaders.

(ii) Relocating loader:
A relocating loader loads the program into different memory locations not a fixed one. It adjusts the memory addresses in the object code during loading to match the allocated memory space.

- Functioning:
1. Reads the relocation information in the object file.
2. Adjusts absolute addresses in the program based on the allocated memory location.
3. Loads the modified program into memory.

- Advantages:
1. Provides flexibility by allowing programs to be loaded anywhere in memory.
2. More efficient memory usage.

- Disadvantages:
Slightly slower than absolute loaders due to address modification.

- Use cases:
Modern OS use relocating loaders to manage memory efficiently.

(iii) Direct linking loader:
A direct linking loader performs the task of loading and linking external references during the load time. It combines multiple object modules and resolves external symbols before executing the program.

__/__/____

- Functioning:
1. Reads the object modules and resolves external references.
2. Loads the entire program into memory.
3. Transfers control to the program's entry point.

- Advantages:
1. Simplifies program execution by handling linking at load time.
2. No Runtime overhead for linking.

- Disadvantages:
1. Larger memory footprint due to the complete program being loaded at once.
2. Inefficient for larger programs with many modules.

- Use Case:
Used in smaller systems or when all modules are available during loading.

(iv) Dynamic linking loader:
A dynamic linking loader links external modules at runtime rather than at load time. Instead of loading the entire program at once, it loads only the necessary modules into memory when required.

- Functioning:
1. Loads the main program into memory.
2. Links external modules when they are first called during execution.
3. Frees memory by unloading unused modules after execution.

- Advantages:
1. Efficient memory usage by loading only necessary modules
2. Supports modular programming and shared libraries.

- Disadvantages:
1. Slight performance overhead due to runtime linking.
2. Dependency issues if external modules are missing.

- Use Case:
& Commonly used in modern OS environments.
eg. DDL, .so

(v) Bootstrap loader:
A bootstrap loader is a special-purpose loader responsible for loading the OS into memory when the computer is powered on. It is usually stored in ROM or firmware.

- Functioning:
1. When the system is powered on, the bootstrap loader executes first.
2. It initializes the hardware and loads the OS into RAM.
3. Transfers control to the OS for further operations.

- Advantages:
1. Automatically loads the OS without user intervention.
2. Essential for system bootstrapping.

- Disadvantages:
1. Only executes during startup.
2. No direct involvement in user program execution.

__/__/____

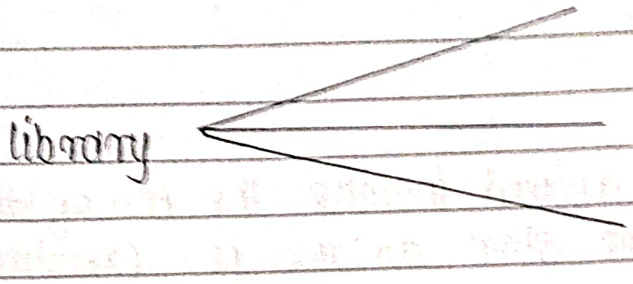
- Use Case:
1. Found in BIOS of PCs.
2. Loads the OS kernel during system boot.

Q8). Explain working of direct linking loader with example, showing entries in different databases built by DLL.

Ans:- Loader is the system program which is responsible for preparing the object program for execution and initiate the execution.
- The loader does the job of co-ordinating with the OS to get initial loading address for the .exe file and load it into the memory.
- Function of loader:
1. Allocation: Allocates the space in the memory where the object program would be loaded for execution.
2. Linking: It links two or more object codes and provides the information needed to allow references between them.
3. Loading: It brings the object program into memory for execution.
- Dynamic linking loader:
1. It is a general re-locatable loader.
2. Allowing the programmer multiple procedure segments and multiple data segments and giving programmer complete freedom in referencing data or instruction contained in other segments.
3. The assembler must give the loader the following information with each procedure or data segment.
4. Dynamic linking defers much of the linking process until a program starts running. It provides a variety of benefits that are hard to get otherwise.

and update

5. Dynamically linked shared libraries are easier to create than static linked shared libraries.
6. The semantics of dynamically linked shared libraries can be much closer to those unshared libraries.
7. Dynamically linking permits ~~to these~~ a program to load and unload routines at runtime, a facility that can otherwise be very difficult to provide.



- Length of segment:

1. A list of all symbols in the segment that maybe referenced by other segments.
2. List of all symbols not defined in the segment but referenced in the segment.
3. Information where the address constant are loaded in the segment.

- Format of Databases:

The assembler provides following types of the record in the object file as follows:

- External Symbol Dictionary (ESD)
- ESD records combine information about all the symbol that are defined in this program.
- But maybe referenced in the program but defined elsewhere.

- Text cards (TXT)
Text card record control the actual object code translated version of the source program.

- Relocation and linkage Directory (RLD) :
- The RLD record contains information about location in the program.
- Whose contents depend on the address at which the program is placed.

- END card :
The END card record initiates the end of the object file and specifies the start address for execution.

eg. RA

```
0     PG1     START
4     ENTRY   A,B
8     EXTRN   PG2, C
        :
        :
20    A       DC
24    B       DC...
        :
        :
40    A+20
44    B-25
48    C-5
  :
  :
52    END
```

__/__/__

- ESD:
It consists of three types of definition:
1. SD - Segment Definition
2. LD - Local Definition
3. ER - External Reference

Symbol	Type	Id	RA	length
PG1	SD	1	0	52
A	LD	1	20	
B	LD	1	24	
PG2	ER	2	-	
C	ER	3	-	

- Relocating and linking Directory (RLD):
It includes different operations performed on it. Flag is important in this table.

Esd id	Symbol	Flag	R.A
1	A	+	40
1	B	-	44
3	C	-	48

- END:
Specifies the END of the program.
Total address is generated by PLA + Slength.

Qu: What is relocation and linking concept with example?

Ans: • Relocation:

- Relocation is the process of modifying the addresses used in a program so that it can be executed at different memory locations.
- It allows a program to run independently of the memory address where it was originally compiled or assembled.
- Need:

1. When multiple programs share memory, they must be loaded at different locations.
2. Relocation ensures that memory references in the program are adjusted based on the actual load address.
3. Helps in dynamic memory allocation and efficient memory usage.

- Types of Relocation:

1. Static Relocation:
   Performed at load time.
   Memory references are modified once when the program is loaded.

2. Dynamic Relocation:
   Done at runtime.
   Uses base registers to adjust memory addresses during execution.
   Used in modern OS.

- eg. A program with the following instructions:

```
MOV AX, [1000] ;
ADD AX, [1002],
```

If the program is loaded at memory address 5000, the loader relocates the addresses:

```
MOV AX, [6000] ;    1000+5000
ADD AX, [6002] ;    1002+5000
```

Relocated addresses:

1000 → 6000
1002 → 6002

- Linking:
- Linking is the process of combining multiple object modules or libraries into a single executable file.
- It resolves external references by connecting different modules.
- Need:

1. Programs are usually divided into multiple modules for modularity and reusability.
2. Linking connects these modules into one executable file
3. Resolves external references.

- Types of linking:

1. Static Linking:
   Performed at compile-time.
   All modules and libraries are combined into a single executable
   The final executable is larger since all dependencies are included.

2. Dynamic linking:
   Done at run-time.
   The program uses shared libraries
   Reduces the size of the executable.

eg: Split a program into two modules.

- main Module (main.o)

```
#include <stdio.h>
int add (int, int);
int main() {
    int result = add (10,20);
    printf ("%d", result);
    return 0;
}
```

- Library Module (math.o)

```
int add (int a, int b) {
    return a+b;
}
```

- linking process:

1. The compiler generates object files: main.o and math.o
2. The linker combines them into a single executable:
   gcc main.o math.o -o program
3. During linking:
   It resolves the reference to add() in main.o by connecting it to the definition in math.o.
   Produces a single executable program.

Q5. Compare Pattern, lexeme and Tokens with example.

	Pattern	lexeme	Token
Definition	A rule or regular expression that defines the structure of a token.	The actual sequence of characters in the source code that matches the pattern.	A categorized representation of the lexeme with a label.
Nature	Abstract	Concrete	Symbolic
Role	Describes how tokens are formed.	Represents the actual occurrence in the code	Used by the parser to interpret the program.
Generated by	Defined by the grammar or RE.	Extracted by the lexical analyzer.	Produced by the lexical analyzer
Format	A rule or expression eg. [0-9]+ for digits	A literal piece of code. eg. 123	A categorized pair eg. <NUMBER, 123>
Example	[a-zA-Z_] [a-zA-Z0$-9_]* → pattern for identifiers	int, a, 10; → lexemes in the code	<KEYWORD, int>, <IDENTIFIER a>, <NUMBER 10>, <SEMICOLON, ;>

Q6]. Explain the phases of compiler Discuss the action taken in various phases to compile the statements :

```
int a, b, c = 1;
a = a * b - 5 * 3 / c ;
```

Ans: - Phases of a compiler :

The compiler converts high-level source code into machine-level code through multiple process phases. Each phase performs a specific task and generates an output used by the next phase.

Phases of Compilation Process :

Phase	Purpose	Output
lexical Analysis	Breaks source code into tokens	Tokens (eg. <IDENTIFIER, a>)
Syntax Analysis	Creates a parse tree from tokens.	Parse tree
Semantic Analysis	Checks for semantic correctness (eg. type checks)	Annotated tree
Intermediate code generation	Converts the source code into intermediate code	Intermediate code (TAC)
Code optimization	Improves the code's efficiency	Optimized intermediate code

Code Generation	Generates machine level code	Target code
Error Handling	Detects and reports errors	Error messages (if any)

Statement Analysis:

```
int a,b,c=1;
a=a*b-5*8/c;
```

Phase 1: Lexical Analysis

Task: Breaks the source code into tokens

Token Generated	Lexeme	Token
int	<KEYWORD, int>	a
<IDENTIFIER, a>	, <COMMA, ,>	b
<IDENTIFIER, b>	, <COMMA, ,>	c
<IDENTIFIER, c>	= <ASSIGN_OP, =>	1
<NUMBER, 1>	; <SEMICOLON, ;>	a
<IDENTIFIER, a>	= <ASSIGN_OP, =>	a
<IDENTIFIER, a>	* <MULTIPLY_OP, *>	b
<IDENTIFIER, b>	- <MINUS_OP, ->	8 5
<NUMBER, 5>	* <MULTIPLY_OP, *>	3
<NUMBER, 3>	/ <DIVIDE_OP, />	c
<IDENTIFIER, c>	; <SEMICOLON, ;>	-

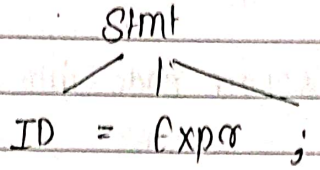
- Phase-2 : Syntax Analysis (Parsing)

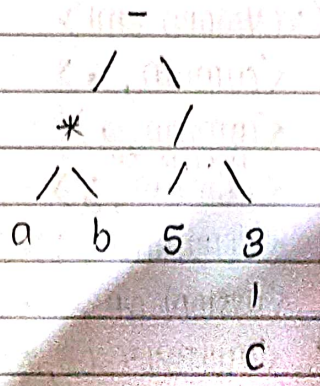
Task : Constructs a parse tree by checking the grammar rules.

Grammar Rules Involved :

Stmt → ID = Expr ;
Expr → Expr + Term | Expr - Term | Term
Term → Term * Factor | Term | Factor | Factor
Factor → ID | NUMBER

Parse Tree :





- Phase-3 : Semantic Analysis

Task : Verifies semantic correctness (eg. type checking, compatibility)

Actions:

1. Ensures the types of a, b and c are compatible with the operations.
2. Checks for division by zero or invalid operations.
3. Ensures the assignment is valid.

Output:
No semantic errors found in this example.

- Phase-4: Intermediate Code Generation

Task: Converts the source code into Intermediate Representation (IR) such as Three-Access Code (TAC).

Three-Access Code:

```
T1 = a * b
T2 = 5 * 3
T3 = T2 / c
T4 = T1 - T3
a = T4
```

- Phase-5: Code Optimization

Task: Improves efficiency by removing redundant operations.

Optimizations Applied:

1. Constant folding: 5 * 3 → 15 is computed at compile time.
2. Common sub-expression elimination: No redundant expression in this case

8. Optimized Intermediate Code :

```
T1 = a * b
T2 = 15/c
T3 = T1 - T2
a = T3
```

- Phase-6 : Code Generation

Task : Converts the intermediate code ~~only~~ into machine-level code.

Generated Assembly Code :

```
MOV AX, a       ; load a into AX
MUL b           ; Multiply with b
MOV BX, 15      ; load 15 into BX
DIV C           ; Divide by c
SUB AX, BX      ; Subtract the result
MOV a, AX       ; Store the final result in a
```

- Phase-7 : Error Handling

Task : Identifies and reports errors during compilation.

Example Errors :

1. Syntax errors : a = a * b - 5 * 3 / ; → missing operand
2. Semantic errors : Division by zero (/0) would raise a runtime error
3. Lexical errors : Invalid characters or unsupported symbols

Q7. Construct LL(1) parsing table for the following grammar and analyze the contents of stack and input buffer and action taken after each step while parsing the input string "abbcbac"

$S \rightarrow Ad$
$A \rightarrow AB \mid BC$
$B \rightarrow b$
$C \rightarrow c \mid \varepsilon$

Ans:- Step-1 : Eliminate left recursion (if any)
The grammar does not have left recursion, so no modification is needed.

- Step-2 : First and Follow Sets

- First Set Calculation :

- FIRST (S)
  - $S \rightarrow Ad \longrightarrow$ FIRST (Ad) = FIRST (A)
  - FIRST (A) $\rightarrow$ AB or BC
    - FIRST (AB) = FIRST (A)
    - FIRST (BC) = FIRST (B)
  - FIRST (B) = {b}
  - FIRST (C) = {c, ε}
  - ∴ FIRST (S) = {b, c}
- FIRST (A)
  - $A \rightarrow AB \rightarrow$ FIRST (AB) = FIRST (A)
  - $A \rightarrow BC \rightarrow$ FIRST (BC) = FIRST (B)
  - FIRST (A) $\rightarrow$ {b, c}
- FIRST (B)
  - $B \rightarrow b$
  - FIRST (B) = {b}

- FIRST (C)
  - C → c → FIRST (C) = {c}
  - C → ε → FIRST (C) = {ε}
  - ∴ FIRST (C) = {c, ε}

- Follow Set Calculation

- FOLLOW (S)
  - Since S is start symbol :
    FOLLOW(S) = {$}

- FOLLOW (A)
  - S → Ad → FOLLOW (A) = {d}
  - A → AB → FOLLOW(A) includes FOLLOW(B)
  - FOLLOW(A) = {b, c, d}

- FOLLOW (B)
  - A → BC → FOLLOW (B) includes FOLLOW(C)
  - FOLLOW (B) = {c, d}

- FOLLOW(C)
  - A → BC → FOLLOW (C) includes FOLLOW (A)
  - FOLLOW (C) = {d}

- LL(1) Parsing Table Construction

Non-Terminal	a	b	c	d	$
S		S → Ad	S → Ad		
A		A → AB	A → BC		
B		B → b			
C			C → c	C → ε	

- Parsing the input string : abbcbac

Initial configuration :
Stack : $S
Input : abbcbac$

Stack	Input	Action Taken
$S	abbcbac$	Apply S → Ad
$Ad	abbcbac$	Apply A → AB
$ABd	abbcbac$	Apply B → b
$bBb	abbcbac$	Match b → consume from input
$Bd	bbcbac$	Apply B → b
$bCd	bcbac$	Match b → consume from input
$Cd	cbac$	Apply C → c
$cCd	cbac$	Match c → consume from input
$d	bac$	Match d → consume from input
$	$	Successful parsing

(Q8). (i) What is left recursion and show how it is removed? Construct SLR parser for the following grammar and parse the input "()()".

$S \rightarrow (S)S \mid \varepsilon$

Ans :

- Left recursion occurs in a grammar when a non-terminal refers to itself as the leftmost symbol in one of its productions.
- This leads to infinite recursion, making the grammar unsuitable for parsing algorithms like LL(1).

eg. $A \rightarrow A\alpha | \beta$
Here, A directly refers to itself at the beginning, making it left-recursive.

- Removing left Recursion:

1. Identify the left-recursive and non-left-recursive productions:
   $A \rightarrow A\alpha$ (left-recursive)
   $A \rightarrow \beta$ (non left-recursive)

2. Rewrite the grammar by introducing a new non-terminal A':
   $A \rightarrow \beta A'$
   $A' \rightarrow \alpha A' | \varepsilon$

- $\beta A'$ generates the valid part of the production
- $A' \rightarrow \alpha A'$ generates the recursive part in a right-recursive manner
- ε allows the recursion to terminate

- SLR Parser Construction:

$S \rightarrow (S)\,S | \varepsilon$
The input string : ( ) ( )

Step 1: Augmented Grammar
Add a new symbol at start S':
$S' \rightarrow S$
$S \rightarrow (S)\,S | \varepsilon$

Step 2: Create LR(0) Items

1. Closure and goto construction

I0:
$S' \rightarrow .S$
$S \rightarrow (S)S$
$S \rightarrow \varepsilon$

goto (I0, S) → I1
goto (I0, '(') → I2
goto (I0, ε) → I3

I1:
$S' \rightarrow S$

I2:
$S \rightarrow (.S)S$
goto (I2, S) → I4

I3:
$S \rightarrow \varepsilon.$

I4:
$S \rightarrow (S).S$
goto (I4, ')') → I5

I5:
$S \rightarrow (S)\ S.$
goto (I5, S) → I6

I6:
$S \rightarrow (S).S$

Step 3: Parse Table

State	(	)	S	$
0	S2		1	
1				Accept
2	S2		4	
3		R1		R1
4		S5		
5	S2		6	
6		R2		R2

SLR Parsing Steps for ()()

Stack	Input	Action
$	()()$	Shift
$(	)()$	Shift
$(S	)()$	Reduce
$(S)	()$	Shift
$(S)S	()$	Shift
$(S)S(	)$	Shift
$(S)S(S	)$	Reduce
$(S)S(S)	$	Reduce
$(S)S(S)S	$	Accept

(ii) Explain syntax-directed translation of semantic analysis.

Ans: - Syntax-directed translation (SDT) is a method of translating high-level language constructs into intermediate code using syntax trees or annotated parse trees.

- It combines parsing and semantic analysis by associating actions with grammar productions.
- Components of SDT:

1. Syntax Tree: Represents the syntactic structure of the input.
2. Attributes:
   - Synthesized attributes: Computed from child nodes and propogated upwards.
   - Inherited attributes: Passed down from parent to child nodes
3. Semantic Rules:
   Used to perform operations like type checking, generating code or building a symbol table.

eg. $E \rightarrow E + T \mid T$

Semantic rules:

$E.val = E1.val + T.val$

$T.val$ = integer value of T

For input 3+4:

Parse tree is built.

The semantic rules calculate the value as 3+4 = 7

- Key applications of SDT:

1. Type checking
2. Symbol table generation
3. Intermediate code generation
4. Optimization

Q]: What is the need of intermediate code generation? Explain the following ways of intermediate code representation with example:

1. Postfix Notation
2. 3-address code
3. Directed acyclic graph (DAG)

Ans: - The Intermediate Code Generator is a crucial phase in a compiler that transforms high-level source code into an intermediate representation (IR) before generating machine code.

- Needs:

1. Platform Independent: It allows compilers to generate code for different target machines by converting source code into an independent IR.
2. Optimization: IR makes it easier to perform optimizations.
3. Translation Efficiency: Separates parsing and code generation making compiler design modular.
4. Ease of Code Generation: IR bridges the gap between high-level languages and machine instructions.

- Ways of intermediate code representation:

1. Postfix notation
   - ~~Operates~~ Operators appear after their operands.
   - Eliminates the need for parentheses.
   - Used in stack-based evaluation.

   eg: Expression: $(3+4)*5$
   Postfix: $3\ 4+5*$

2. Three-Address Code (3-TAC):
- Uses atmost three operands per instruction.
- Introduces temporary variables to 3 breakdown expressions.

eg. Expression: $a = (b+c) * d$

TAC:

$t_1 = b + c$

$t_2 = t_1 * d$

$a = t_2$

- Simple and easy to optimize.
- Used in IR like LLVM IR.

3. Directed Acyclic Graph (DAG):
- Represents expressions as a graph, avoiding redundant
- Helps in common sub-expression elimination (CSE).

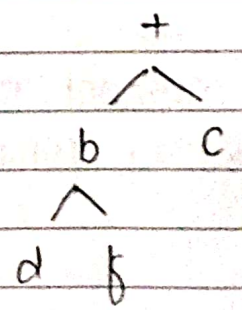
eg. Expression:

$a = b+c;$

$d = a+e;$

$f = b+c$

DAG Representation:



The sub-expression b+c is computed once and reused for a and f.

- CSE optimizes redundant computations, reducing execution time.

Q.6) Explain the following code optimization techniques in detail with the help of example.

1. Dead Code Elimination :

- Definition :

Removes unreachable or unused code that does not affect the program's output. It reduces code size and improves the performance.

eg.

```
int a = 10;
int y = a * 2;
return 5;
```

Optimized code :

```
return 5;
```

The variable y is never used, so its computation is eliminated.

2. Constant Propogation :

- Definition :

Replaces variables with constant values whenever possible. It reduces unnecessary computations.

eg.

```
int a = 5;
int y = a + 3;
printf ("%d", y);
```

Optimized code:

```
int y = 8;
printf("%d", 8);
```

- x is always 5, so x+3 is replaced by 8.

3. Common sub-expression elimination (CSE):

- Definition:

It removes redundant expressions that compute the same result multiple times.

eg.
```
int a = b+c;
int d = b+c+e;
```

Optimized code:

```
int a = b+c;
int d = a+e;
```

b+c is computed once and stored in a, then reused in d.

4. Code Motion:

- Definition:

It moves invariant expressions outside of loops to reduce redundant computation.

eg.
```
for (int i=0; i<n; i++) {
    int x = a+b;
    arr[i] = x*i; }
```

Optimized code:

```
int x = a+b;
for (int i=0; i<n; i++) {
    arr[i] = x*i; }
```

a+b does not change in the loop, so we compute it once before looping.

5. Strength Reduction:

- Definition:

It replaces expensive operations with cheaper ones.

eg. for (int i=0; i<n; i++) {
arr[i] = i * 2; }

Optimized code:

for (int i=0; i<n; i++) {
arr[i] = i << 1; }

i * 2 is replaced with i<<1, which is faster on most architectures.

Q11]. Explain the concept of basic block and flow graph and construct flow graph with the help of basic block for the following example:

min = a[0];
for (i=1; i<n; i++) {
if (a[i] > max) {
max a[i];
flag = 1;

Ans: 1. Basic Block:

A basic block is a sequence of instructions with:

- A single entry point (first instruction)
- A single exit point (last instruction)
- No branching except at the end (i.e. no jump/conditional statements inside)

eg: x = a + b ;
y = x * c ;
z = y - d ;

The execution always from one statement to the next without interruption.

2. Flow Graph :

A flow graph is a directed graph where :

- Nodes (vertices) represent basic blocks
- Edges (Arrows) represent control flow (jumps, loops, conditions)

eg: Given code :

```
min = a[0];
for (i=1; i<n; i++) {
    if (a[i] > max) {
        max = a[i];
        flag = 1; }
}
```

Step 1 : Identify Basic Blocks

1. B1 (Initialization Block)

```
min = a[0];
i = 1;
```

2. B2 (loop Condition)

```
if (i<n) goto B3;
else goto B5;
```

3. B3 (Condition Check Block):

```
if (a[i] > max) goto B4;
else goto B6;
```

4. B4 (Update Max - Block - Executed if condition is True):

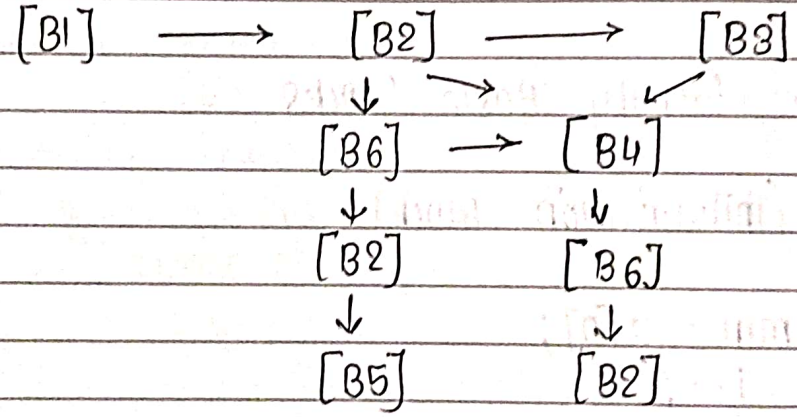
```
max = a[i];
flag = 1;
```

5. B6 (Increment loop Variable and Repeat)

```
i = i+1;
goto B2;
```

6. B5 (Exit Block - Ends Execution)

Step 2: Flow Graph Representation



- B1 initializes min and i
- B2 checks if i < n. If true goes to B3 else ends at B5.
- B3 checks if a[i] > max
  - If true jumps to B4

- If false, skips to BB
- BB increments i and loops back to BB for the next iteration.
- BB marks the program's end.

Q10] Explain design issues of code generator in detail.

Ans:
- A code generator translates intermediate code (IR) into machine code or assembly code.
- Several design issues must be addressed to ensure efficiency, correctness and optimization.
- The design issues are:

1. Target Machine Architecture:
- The generated code must match the CPU architecture. (RISC, CISC, registers, memory).
- Consider instruction set, number of registers and addressing modes.
- eg. For RISC machines (eg. ARM), load/store architecture means operations must use registers instead of direct memory access.

2. Intermediate Code Selection:
- The choice of IR (Three-Address Code, Abstract Syntax Tree, DAG) affects code generation.
- Some IRs allow better optimizations than others.

eg. a = b+c;
d = a+e;

using TAC:

t1 = b+c;
d = t1+c;

Helps identify common sub-expressions for optimization.

3. Instruction Selection:
- Different instructions can be used for the same operation.
- The best instruction should minimize execution time and memory usage.

eg. `x = y * 2;`

Optimized code:

```
x = y << 1;
```

- Strength reduction improves performance.

4. Register Allocation and Assignment:
- Registers are faster than memory, so the compiler should maximize register usage.
- Uses Register Allocation (which values go into registers) and Register Assignment (which register to use)
- Techniques:
1. Graph Coloring Algorithm
2. Spilling

5. Handling Conditional and Unconditional Jumps:
- Efficiently generate code for loops, conditions and functional calls.
- Avoids unnecessary jumps to improve execution speed.

eg.

```
for (i=0; i<10; i++)
    sum += i;
```

- Instead of multiple jump instructions, unrolling the loop can reduce jumps.