Assignment-V

1. Explain detail about code optimization and its techniques in detail.

Code optimization:-

* code optimization is compiler design is the process of improving the intermediate code so that the final machine code is more efficient - without changing the program's output or meaning.
* It's a stage in the compiler that makes the program:
    1) Run faster
    2) use less memory.

Example:-

```
x = 5 * 2;
y = x * 4;
```

After code optimization

```
y = 40
```

* The technique used in code optimization is peephole optimization.

Peephole optimization:-

* peephole optimization is a type of code-optimization performed on a small part of the code.

* It is performed on a very small set of instructions in a segment of code.

* The small set of instructions or small part of code on which peephole optimization is performed is known as peephole or window.

* It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without a change in output.

* The peephole is machine-dependent optimization.

* peephole optimization improves the efficiency of compiled code by eliminating unnecessary instructions in small code sequences.

* The techniques of peephole optimization is

1) Redundant instruction elimination
2) Removal of unreachable code
3) flow of code optimization
4) Algebraic simplification
5) Machine idioms

1) Redundant instruction elimination:-

* It is the process of removing unnecessary or duplicate instructions in a small window of code that do not affect the final program result.

* Since redundant instructions only waste CPU cycles without changing the program's output, eliminating them improves efficiency.

* Remove instructions that don't change program state.

* Remove duplicate assignments to the same variable/ register without intermediate use.

* Remove instructions that cancel each other.

Example :-

1. Same instruction repeated

```
Mov R1, R2;
Mov R1, R2;
```

→ The second instruction is unnecessary and can be removed.

2. operation followed by its reverse

```
INC R1
Dec R1;
```

→ both instructions can be removed as they have no net effect.

3. Useless stored-load pairs

```
Store X
Load X;
```

2) Removal of unreachable code

* Removal of unreachable code is a common technique in peephole optimization for improving compiled code efficiency.

* In peephole optimization, the compiler examines small windows of code to detect and eliminate instructions that will never be executed, such as unlabeled instructions immediately following unconditional jumps or code after a return statement.

Example:-

```
goto L2
x=x+1;
L2: a=a+b
```

The statement x=x+1 can be removed.

3) flow of code optimization:-

* The flow of code optimization in peephole optimization follows a simple, repeated process focused on small, local sections of code.
* peephole optimization operates after code generation and looks for specific patterns that can be improved for performance or code size.

Example:-

Input code

```
Mov R1,R2
Mov R1,R2
INC R3
DEC R3
Jmp L1
ADD R4,R5
L1: Mul R6,2
```

Optimized code

```
Mov R1,R2
L1: SHL R6,1
```

4) Algebraic simplification:-

* It is a peephole optimization technique where algebraic identities and simple arithmetic rules are applied to make instructions shorter and faster, without changing the meaning of the program.

Example:-

Before optimization	After optimization
Mul R_1,1	SHL R_3,1
ADD R_2,0	Mov R_4,0
Mul R_3,2	
Sub R_4,R_4	

5) Machine idioms:-

* A machine idioms is a sequence of instructions in intermediate or assembly code that can be replaced by a single, efficient machine instruction supported directly by the hardware.
* peephole optimization detects these patterns and replaces them with the corresponding idiomatic instruction.
* This makes the code smaller, faster, and closer to machine capabilities.

Example:-

The machine idioms are

1) SHL R_1,1
2) SHR R_2,1
3) CLR R_3
4) INC R_4
5) Dec R_4
6) Test R_5,R_5
7) SHL R_6,3

Basic block and flow graph:-

Basic block is a straight line code sequence that has no branches in and out branches except to the entry and at the end respectively.

* Basic block is a set of statements that always executes one after other, in a sequence.
* The first task is to partition a sequence of three-address codes into basic blocks.
* A new basic block is begun with the first instruction and instructions are added until a jump or label is met.
* In the absence of jump, control moves further consecutively from one instruction to another.

Algorithm for converting three address code into basic blocks:-

Input- A sequence of three address instructions.

process- Instructions from intermediate code which are leaders are determined. The following are the rules used for finding the leader:

1) The first three address instruction of the intermediate code is a leader.
2) Instructions that are targets of unconditional or conditional jump/goto statements are leaders.
3) Instructions that immediately follow unconditional or conditional jump/goto statements are considered leaders.

Example:-

The three address code is:

$t_1 = a * a$

$t_2 := a * b$

$t_3 := 2 * t_2$

$t_4 := t_1 + t_3$

$t_5 := b * b$

$t_6 := t_4 + t_5$

The intermediate code to set a 10*10 matrix to an identity matrix:

1) i=1

2) j=1

3) $t_1 = 10 * i$

4) $t_2 = t_1 + j$

5) $t_3 = 8 * t_2$

6) $t_4 = t_3 - 88$

7) a[t_4] = 0.0

8) j=j+1

9) if j<=10 goto (3)

10) i=i+1

11) if i<=10 goto (2)

12) i=1

13) $t_5 = i - 1$

14) $t_6 = 88 * t_5$

15) a[t_6] = 1.0

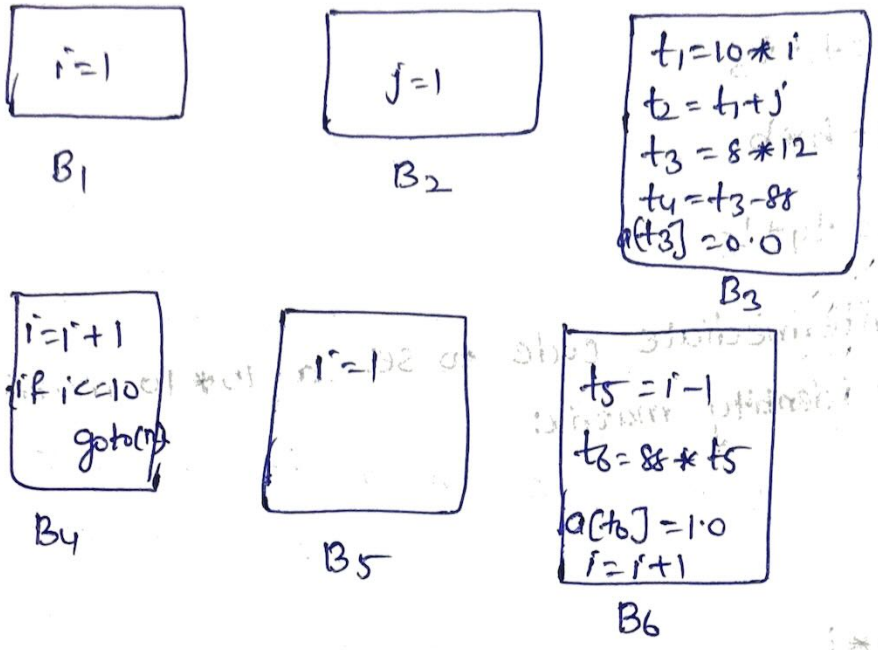
16) i=i+1

17) if ic<=10 goto (13)

The basic blocks are

The leaders are

stmt-1, stmt-2, stmt-3, stmt-10, stmt-12, stmt-13



Flow graphs:-

A flow graph is simply a directed graph. For the set of basic blocks, a flow graph shows the flow of control information. A control flow graph is used to decipt how the program control is being parsed among the blocks. A flow graph is used to illustrate the flow of control between basic blocks once an intermediate code has been partitioned into basic blocks. when the beginning instruction of the y block follows the last instruction of the x block, an edge might flow from one block x to another block y.

Example:-

```
if (x>0) {
  y=y+1;
}else{
  y=y-1;
}
z=z+y;
```

step-1:- Identify basic blocks

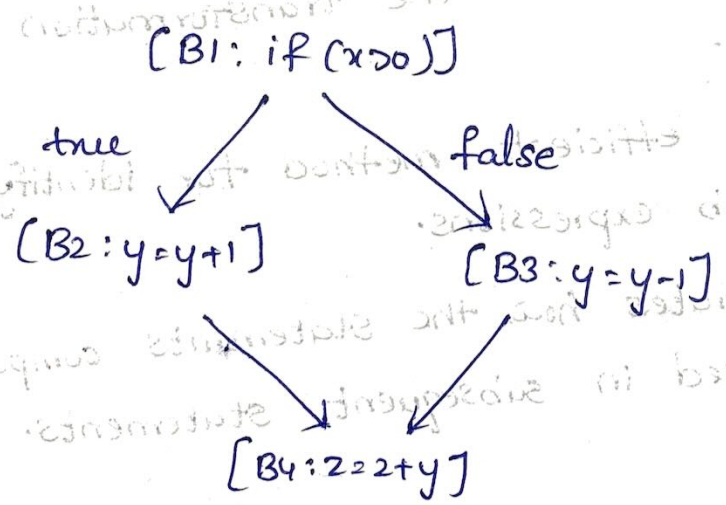
B1: if (x>0) - condition block

B2: y=y+1 (true branch)

B3: y=y-1 (false branch)

B4: z=z+y (common successor)

step 2: Draw flow graph



Directed acyclic graph:-

In compiler design, a Directed acyclic graph (DAG) plays a crucial role in representing expressions and optimize code. A DAG is a graph containing directed edges but no cycles, ensuring no path leads back to the starting node. DAG's are particularly useful in eliminating redundant

Computations and detecting common sub-expressions, making program execution more efficient. This graphical representation helps optimize the intermediate code generation phases of a compiler.

* The DAG is used to represent the structure of basic blocks, visualize the flow of values between basic blocks, and provide optimization techniques in basic block.
* DAG is a type of data structure and they are used to apply transformations to basic blocks.
* The DAG faciliates the transformation of basic blocks.
* DAG is an efficient method for identifying common sub-expressions.
* It demonstrates how the statements computed value is used in subsequent statements.

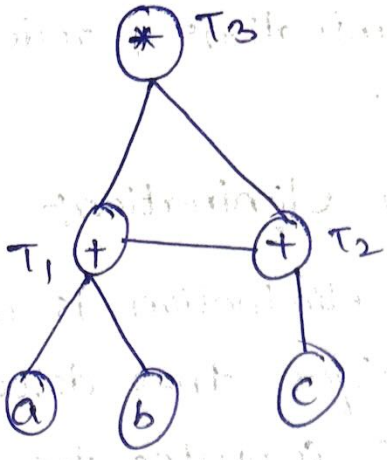
Example:-

The code is

$T_1 = a + b$

$T_2 = T_1 + c$

$T_3 = T_1 \times T_2$



Applications:-

1) Identification of common subexpressions.
2) Tracking variable usage
3) statement value tracking
4) code representation
5) Reactive value systems

principle sources of code optimization:-

The principle sources of code optimization in compiler design include techniques that enhances the performance and efficiency of generated code without changing its output.

* The principle source of code optimization contains:
  1) common sub expression elimination
  2) constant folding
  3) copy progation
  4) dead code elimination
  5) code motion

6) Induction variable eliminations & reduction in strength.

1) Common sub-expression elimination:-

Common sub-expression elimination is a compiler optimization technique that detects repeated expressions in code, computes them once, and resuses the result, thereby improving performance and reducing redudant calculation.

Example:-

Before common sub-expression elimination

```
a = b * c + g
d = b * c * e
```

After common sub-expression elimination

```
temp = b * c
a = temp + g
d = temp * e
```

2) Constant folding :-

Constant folding is a principle of code optimization which involves recognizing and calculating expressions with constants during compilation rather than at runtime.

* By using this technique we can reduce runtime compilation and improve efficiency.

Example:-

```
k = 5
x = 2 * k      →      k = 10
y = k + 5             y = 10
```

3) Copy propagation:-

Copy propagation is an optimization used in compiler design to replace usexs of a variable that is simplify copied from another variable with the value of original variable, reducing unnecessary assignments and helping enable further optimizations like dead-code elimination and constant propagation.

Example:-

```
X = pi ;
A = X * r * r;   →   A = pi * r * r;
```

4) Dead code elimination:-

Dead code elimination is an important compiler optimization technique that removes portions of code that do not effect the final output of the program. Dead code includes code that is never executed during runtime or code whose results are never used.

Example:-

```
z = x * y;                  z = x * y;
a = x;          →           w = x * y + 6;
w = x * y + 6;
```

5) Code motion:-

Code motion identifies statements or expressions that do not change within a loop and moves them outside the loop so they are executed

only once instead of repeatedly in each iteration. Reducing redundant computations speeds up program execution and improves resources usage.

Example:-

```
a=200;
while (a>0) {
  b=x+y;
  if (a%b==0)
    printf ("%d", a);
  a--;
}
```

→

```
a=200;
b=x+y;
while (a>0){
if (a%b==0)
  printf ("%d", a);
  a--;
}
```

6) Induction variable elimination and reduction in strength:-

Induction variable elimination and reduction in strength are loop optimization techniques used by compilers to improve performance by simplifying arithmetic operations and inside loops.

An induction variable in a loop is a variable in a loop is a variable that changes by a fixed amount on each iteration, commonly used for indexing array and controlling loop bounds.

Example:-

```
int i=0, result=0;
for(i<max; i++){
    result+=2;
}
```

→

```
int result=0;
for (i=0; result<max*2; result
                        +=2)
{
}
```

code generation

2) Explain detail about code genrations, its issues and simple code generation.

Code generation:-

Code generation is the final phase in compiler design, where the intermediate representation of the source program is translated into target machine code or assembly language that the target system can execute directly.

Example:-

```
a=b+c  →  Mov b R0
          Add R0 c
          Mov R0 a
```

Issues in design of code generation:-

A code generator is a crucial part of compiler that converts the intermediate representation of source code into machine-readable instructions. Its main task is to produce the correct and efficient code that can be executed by a computer. The design of the code generator should ensure that it is easy to implement, test and maintain. There are several issues that can arise in code generation phase:-

1) Input to code generator

The input to the code generator comes from the intermediate code generator by the compiler-

front-end. This intermediate code is usually a higher-level representation of the program, like triples, quadruples, or abstract syntax trees. Along with this intermediate code, the code generator also uses information from the symbol table, which holds the addresses of variables and other data objects. One key challenge here is that the input must be free from syntactic and semantic errors, as the code generator assumes that proper type-checking and other checks have already been handled by the front-end. Handling the input correctly is crucial for generating the correct target code.

2) Target program:-

The target program is the final output of the code generator, which can be in the form of absolute machine language, relocatable machine language, or assembly language. Each type of output has its own set of challenges:

i) Absolute machine language: It is easy to execute but lacks flexibility because it is bound to specific memory locations.

ii) Relocatable machine language: It allows parts of the program to be moved around in memory, making it suitable for linking multiple modules, but it requires a linking loader and has some overhead.

iii) Assembly language: It is symbolic and needs an additional step to convert it into machine code, but it makes the code generation process easier.

Choosing the appropriate form for the target program depends on factors such as the program's needs, execution environment, and whether the program will be linked with other modules.

Memory management:-

Memory management in the code generation phase involves mapping variable names to their corresponding memory locations. The code generator work closely with the front-end to access the symbol table, where memory addresses for variables are stored. A major challenge is ensuring that the code generator uses memory efficiently, avoids memory conflicts, and correctly handles dynamic memory allocation. This requires careful handling of variable storage, particulary for dynamically allocated objects or large data structures, such as arrays or objects in object-oriented languages.

4) Instruction selection:-

* Instruction selection is the process of choosing the most suitable machine instructions to translate intermediate code into executable code. The goal is to optimize the generated code by selecting instructions that are efficient and approximate

for the target machine.

* If the right instructions are not selected, the resulting code can be inefficient and slow.
* A code generator might need to decide between different ways of implementing the same operation, such as optimizing for processor-specific features.

Assemb

Register allocation issues:-

* Efficient use of registers is important because registers are faster than memory, and utilizing them efficiently can significantly improve program performance. The challenge lies in selecting the right variables to store in registers at different points in the program.

Register allocation involves two stages:

1. Register allocation: It is selecting which variables will reside in the registers at each point in the program.
2. Register assignment: Assigning specific registers to those variables selected in register allocation.

The difficulty arises in managing which variables are allocated to registers, especially when the number of available register is limited. Poor register allocation can lead to spills, where data is temporarily stored in memory, causing slower performance.

7) Evalution order:-

The evalution order refers to the sequence in which expressions are evaluated in the generated code. This order can significantly affect the efficiency of the program.

Simple code generation:-

Simple code generation in compiler design refers to the phase where the intermediate representation of a source program is converted into target machine code, often in the form of assembly or object code.

* The code generation phase of a compiler takes a IR and translates it into a sequence of machine instructions for the target architecture.
* This phase directly affects the efficiency and performance of the resulting program, as it determines what instructions run on the hardware.

Basic steps in simple code generation:-

1) Register Descriptor set-up
2) Instruction selection
3) Register allocation
4) Instruction selectio
4) Instruction generation
5) Basic block generation and ending

Example:-

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

$t_1 = a-b$

$t_2 = a-c$

$t_3 = t_1 + t_2$

$d = t_3 + t_2$

Statement	Code generate	Register	address

$t_1 = a-b$	Mov a, R_0
	Sub b, R_0
	Mov R_0, t_1
		R_0 is empty
		R_0 is t_1
			t_1 have R_0

$t_2 = a-c$	Mov a, R_1
	sub c, R_1
	Mov R_1, t_2
		R_0 contains t_1
		R_1 contains t_2
			t_1 in R_0
			t_2 in R_1

$t_3 = t_1 + t_2$	add R_1, R_0 add R_1, R_0	R_0 contains t_3 R_1 contains t_2	t_3 in R_0 t_2 in R_1
$d = t_3 + t_2$	Mov R_1, d	R_0 contains d	d will be answer