GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL

COMPUTER ENGINEERING DEPARTMENT

A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: **System Programming & Compiler Construction**

Experiment No: 8

Title of Experiment: Implementation of target code generator.

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	10	
2	Oral	5	
	Total	15	

Signature of Subject Teacher

Theory:

Target Code Generator –

Target code generation is the final Phase of Compiler.

1. Input: Optimized Intermediate Representation.
2. Output: Target Code.
3. Task Performed: Register allocation methods and optimization, assembly level code.
4. Method: Three popular strategies for register allocation and optimization.
5. Implementation: Algorithms.

Target code generation deals with assembly language to convert optimized code into machine understandable format. Target code can be machine readable code or assembly code. Each line in optimized code may map to one or more lines in machine (or) assembly code, hence there is a 1:N mapping associated with them.

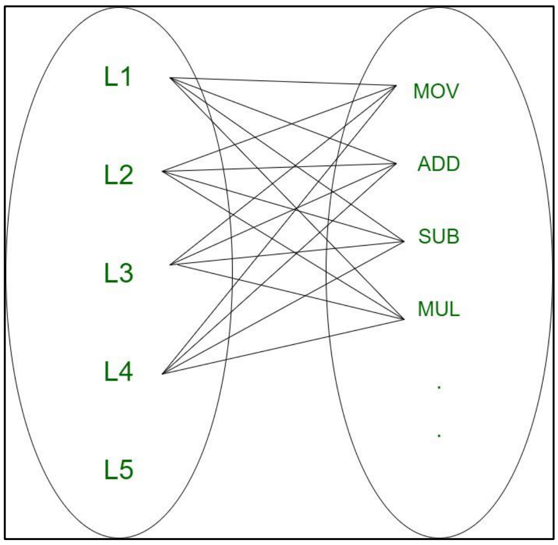


Fig. 1:N Mapping

Computations are generally assumed to be performed on high speed memory locations, known as registers. Performing various operations on registers is efficient as registers are faster than cache memory. This feature is effectively used by compilers, however registers are not available in large amount and they are costly. Therefore we should try to use minimum number of registers to incur overall low cost.

Optimized Code –

```
Example 1 :
L1: a = b + c * d

optimization :
t0 = c * d
a  = b + t0
Example 2 :
L2: e = f - g / d

optimization :
t0 = g / d
e  = f - t0
```

Register Allocation –

Register allocation is the process of assigning program variables to registers and reducing the number of swaps in and out of the registers. Movement of variables across memory is time consuming and this is the main reason why registers are used as they available within the memory and they are the fastest accessible storage location.

```
Example 1:
R1<--- a
R2<--- b
R3<--- c
R4<--- d

MOV R3, c
MOV R4, d
MUL R3, R4
MOV R2, b
```

```
ADD R2, R3

MOV R1, R2

MOV a, R1

Example 2:

R1<--- e

R2<--- f

R3<--- g

R4<--- h


MOV R3, g

MOV R4, h

DIV R3, R4

MOV R2, f

SUB R2, R3

MOV R1, R2

MOV e, R1
```

Advantages –

- Fast accessible storage
- Allows computations to be performed on them
- Deterministic as it incurs no miss
- Reduce memory traffic
- Reduces overall computation time

Disadvantages –

- Registers are generally available in small amount (up to few hundred Kb)
- Register sizes are fixed and it varies from one processor to another
- Registers are complicated
- Need to save and restore changes during context switch and procedure calls

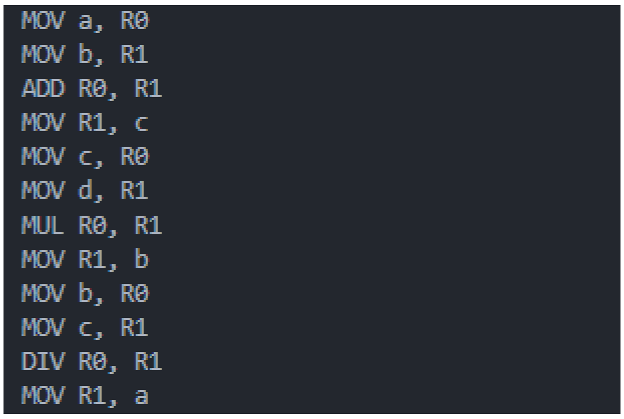
Program Code -

```
# Mapping of operators to their corresponding three address code instructions
ops_dict = {'+': 'ADD', '-': 'SUB', '*': 'MUL', '/': 'DIV'}

# Read input expressions from a txt file
with open('Expt8.txt', 'r') as f:
    input_str = f.read()

# Split the input string into lines by newline character
input_lines = input_str.split('\n')

# Remove any empty lines from the list
input_lines = [line for line in input_lines if line]

# Initialize the target code as an empty string
target_code = ''

# Loop over the input lines to generate the target code for each expression
for input_line in input_lines:
    # Split the input line into tokens by whitespace
    input_tokens = input_line.split()

    # Validate that the input expression is well-defined
    if len(input_tokens) < 4 or input_tokens[1] != '=':
        print(f'Error: invalid input expression {input_line}')
    else:
        # Extract the left-hand side variable
        left_var = input_tokens[0]
        # Initialize the counter for the register number
        register_counter = 0


        # Loop over the input tokens to generate the target code
        for token in input_tokens[2:]:
            if token in ops_dict:
                # If the token is an operator, save it for later use
                operator = token
            else:
                # If the token is an operand, assign it to a register and
generate the appropriate three address code instruction
                target_code += f'MOV {token}, R{register_counter}\n'
                if register_counter == 0:
                    register_counter += 1
                else:
                    target_code         +=         f'{ops_dict[operator]}
R{register_counter-1}, R{register_counter}\n'
                    register_counter += 1


        # Generate the final instruction to assign the result of the
expression to the left-hand side variable
        target_code += f'MOV R{register_counter-1}, {left_var}\n'


# Write the final target code to a txt file
print(target_code)
```

Input File –

```
c = a + b
b = c * d
a = b / c
```

Output –



```
MOV a, R0
MOV b, R1
ADD R0, R1
MOV R1, c
MOV c, R0
MOV d, R1
MUL R0, R1
MOV R1, b
MOV b, R0
MOV c, R1
DIV R0, R1
MOV R1, a
```

Conclusion –

Thus we studied the implementation of Target Code Generator.