Experiment No 6

Aim : Generate a target code for the optimized code.

Source Code:

```
import random
import re

input_op = input("Enter an expression: ")
expression=input_op

# # intermediate representation
# intermediate_code = [
#     {'op': '+', 'arg1': 'b', 'arg2': 60.0, 'result': 't1'},
#     {'op': '=', 'arg1': 't1', 'arg2': None, 'result': 'a'}
# ]

def generate_intermediate_code(expression):
    # Initialize intermediate code list
    intermediate_code = []

    # Parse the expression into variables and operators
    variables = re.findall(r'[a-zA-Z_][a-zA-Z0-9_]*', expression)
    operators = re.findall(r'[+\-*\/=]', expression)

    # Generate intermediate code for arithmetic operations
    t_count = 1
    result = variables[0]
    for i in range(len(operators)-1):
        arg1 = variables[i]
        arg2 = variables[i+1]
        op = operators[i]
        if op == '+':
            result = f't{t_count}'
            intermediate_code.append({'op': '+', 'arg1': arg1, 'arg2': arg2, 'result': result})
            t_count += 1
        elif op == '-':
            result = f't{t_count}'
            intermediate_code.append({'op': '-', 'arg1': arg1, 'arg2': arg2, 'result': result})
            t_count += 1
        elif op == '*':
            result = f't{t_count}'
            intermediate_code.append({'op': '*', 'arg1': arg1, 'arg2': arg2, 'result': result})
            t_count += 1
        elif op == '/':
            result = f't{t_count}'
            intermediate_code.append({'op': '/', 'arg1': arg1, 'arg2': arg2, 'result': result})
            t_count += 1
        elif op == '=':
            intermediate_code.append({'op': '=', 'arg1': arg1, 'arg2': None, 'result': arg2})
            result = arg2
```

```
    # Return the intermediate code list
    return intermediate_code

intermediate_code=generate_intermediate_code(expression)

# extract all variables used in the program
var_list = set()
for statement in intermediate_code:
    var_list.add(statement['result'])
    var_list.add(statement['arg1'])
    var_list.add(statement['arg2'])
var_list.discard(None) # remove None from the variable list

# initialize address descriptor for all variables used in the program
addr_descriptor = {}
for var in var_list:
    addr_descriptor[var] = {'in_register': None, 'in_memory': None, 'next_use': None}


# Register descriptor list to keep track of register usage
reg_descriptor = [{'name': f"R{i}", 'occupied_by': None} for i in range(8)]

# Function to get the location to hold the result of an assignment
def getreg(y, z, x):
    # Check if y is in a register that holds the value of no other names, and y is not live and has
no next use
    for reg in reg_descriptor:
        if reg['occupied_by'] == y and not is_live(y, x) and not has_next_use(y, x):
            # Update the address descriptor of y to indicate that y is no longer in this register
            addr_descriptor[y]['in_register'] = None
            reg['occupied_by'] = None
            return reg['name']

    # Check if there is an empty register available
    for reg in reg_descriptor:
        if reg['occupied_by'] is None:
            return reg['name']

    # Find an occupied register R if x has a next use in the block or operator requires a register
    for reg in reg_descriptor:
        if reg['occupied_by'] == y or reg['occupied_by'] == z:
                # Store the value of the register into a memory location if it is not already in the
proper memory location
            M = addr_descriptor[reg['occupied_by']]['in_memory']
            if M != None and M != reg['occupied_by']:
                print(f"MOV {reg['occupied_by']}, {M}")
            addr_descriptor[reg['occupied_by']]['in_memory'] = None

            # Update the address descriptor for M and return R
            addr_descriptor[M]['in_register'] = None
            addr_descriptor[x]['in_register'] = reg['name']
```

```
            addr_descriptor[reg['occupied_by']]['in_memory'] = x
            reg['occupied_by'] = x
            return reg['name']

    # Select the memory location of x as L
    addr_descriptor[x]['in_register'] = None
    return addr_descriptor[x]['in_memory']

# Function to check if a variable has a next use in the block
def has_next_use(var, x):
    # Simulating with random value
    return random.choice([True, False])

# Function to check if a variable is live after execution of an assignment
def is_live(var, x):
    # Simulating with random value
    return random.choice([True, False])



# Parse the input operation to get the variable names and the operator
tokens = input_op.split()
x = tokens[0]
y, op, z = tokens[2], tokens[3], tokens[4]

# Invoke getreg function to determine the location L where the result of computation y op z
should be stored
L = getreg(y, z, x)

# Generate instruction MOV y, L to place a copy of y in L if y is not already in L
if addr_descriptor[y]['in_register'] != L:
    print(f"MOV {y}, {L}")

# Generate instruction corresponding to the operator
if op == '+':
    print(f"ADD {L}, {z}")
elif op == '-':
    print(f"SUB {L}, {z}")
elif op == '*':
    print(f"MUL {L}, {z}")
elif op == '/':
    print(f"DIV {L}, {z}")

# Generate the final MOV instruction to move the result to variable a
print(f"MOV {L}, {x}")
```

Output:

```
PS D:\App Develop> & C:/Users/kris/
Enter an expression: a = c + 50
MOV c, R0
ADD R0, 50
MOV R0, a
PS D:\App Develop>
```

```
PS D:\App Develop> & C:/Users/kris/
Enter an expression: a = b / 10.0
MOV b, R0
ADD R0, 10.0
MOV R0, a
PS D:\App Develop>
```