Experiment No 6

Aim : Generate a target code for the optimized code.

Algorithm:

The final phase in the compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

The code generation algorithm takes as input a sequence of three address statements constituting a basic block. For each three address statement of the form x=y op z we perform following function:

1. Invoke a function getreg to determine the location L where the result of computation y op z should be stored. (L cab be a register or memory location .

2. Consult the address descriptor for y to determine y, the current locations of y. Prefer the register for y if the value of y is currently both in memory and register. If the value of y is not already in L , generate the instruction MOV y, L to place a copy of y in L.

3. Generate instruction po z, L where z is a current location of z. Again address descriptor of x to indicate that x is in location L. if L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptor.

4. If the current values of y and z have no next uses , are not live on exit from the block , and are in registers alter the register descriptor to indicate that , after execution of x=y op z, those register no longer will contain y and z, resply.

The function getreg:

The function getreg returns the location L to hold the values of x for the assignment x= y op z.

1.If the name y is in a reg that holds the value of no other names, and y is not live and has no next use after execution of x= y op z ,then return the register of y for L. Update the address descriptor of y to indicate that y is no longer in L.

2. Failing (1), return an empty register for L if there one.

3. Failing (2) , if X has a next use in the block, or op is an operator , such as indexing, that requires a register find an occupied register R. Store the values of R into a memory location (MOV R ,M) if it is not already in the proper memory location M, update the address descriptor for M , and return R. if R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable register might be one whose data is referenced furthest in the future, or one whose value is also in

memory. We leave the exact choice unspecified, since there is no one proven best way to make the selection.

4. If x is not used in the block , or no suitable occupied register can found, select the memory location of x as L.

Code:

```
registers = []
result = []
arg1 = []
arg2 = []
op = []

def get_instr(operator):
    if operator == "+":
        return "ADD"
    return "SUB"

def find_reg(operand):
    for index in range(len(registers)):
        if registers[index] == operand:
            return index
    return -1

def gen_register():
    return len(registers)

code = ""
line_count = 0

with open("input.txt", "r") as file:
    for line in file:
        line = line.rstrip().split(" ")
        result.append(line[0])
        arg1.append(line[2])
        if len(line) > 3:
            op.append(line[3])
            arg2.append(line[4])
        else:
            op.append(None)
            arg2.append(None)
        line_count += 1
```

```
for i in range(line_count):
    if op[i]:
        temp1 = find_reg(arg1[i])
        temp2 = find_reg(arg2[i])
        tempReg = 0

        if temp1 == -1 and temp2 == -1:  # registers containing that operand does not exist
            tempReg = gen_register()  # generate new register
            code += f"MOV {arg1[i]}, R{tempReg}\n"
            code += f"{get_instr(op[i])} {arg2[i]}, R{tempReg}\n"
            registers.append(result[i])

        elif not temp1 == -1 and not temp2 == -1:  # registers containing that operand exist
            code += f"{get_instr(op[i])} R{temp1}, R{temp2}\n"
            registers[temp2] = result[i]

    else:
        temp1 = find_reg(arg1[i])
        if not temp1 == -1:  # registers containing that operand exist
            code += f"MOV R{temp1}, {result[i]}\n"
        else:
            tempReg = gen_register()
            code += f"MOV {arg1[i]}, R{tempReg}\n"
            code += f"MOV R{tempReg}, {result[i]}"

print(code)
```

Input.txt:

```
t = b + c
v = d + c
u = t - v
w = t + u
a = w
```

Output:

```
MOV b, R0
ADD c, R0
MOV d, R1
ADD c, R1
SUB R0, R1
ADD R0, R1
MOV R1, a
```

Conclusion:

Learnt and understood how to generate target code from optimized code and illustrate it in python.

Postlab:

1. **Explain design issues of code generator phase?**
2. **What are basic blocks? State their properties**