Department of Computer Engineering

Academic Term : Jan-May 23-24

Class : T.E. (Computer)
Subject Name : System Programming and Compiler Construction
Subject Code : (CPC601)

Practical No:	6
Title:	Target Code Generator
Date of Performance:	
Date of Submission:	
Roll No:	9601
Name of the Student:	Ivan Dsouza

Evaluation:

Sr. No	Rubric	Grade
1	Time Line (2)	
2	Output(3)	
3	Code optimization (2)	
4	Postlab (3)	

Signature of the Teacher :

Experiment No 6

Aim : Generate a target code for the optimized code.

Algorithm:

The final phase in 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 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.

Conclusion:

Postlab:

Explain design issues of code generator phase?

What are basic blocks? State their properties

```
import re
operatorTable = {
    "+": "ADD",
    "-": "SUB",
    "*": "MUL",
    "/": "DIV",
}

registerTable = {}


def getRegisterByIndex(index):
    for [key, value] in enumerate(registerTable):
        if value == 'R' + index:
            return key

    return -1


def getRegisterByOperand(operand):
    if operand not in registerTable.keys():
        registerTable[operand] = "R" + str(len(registerTable))
```

```
    return registerTable[operand]



def parseLine(line):
    line = line.strip().replace(" ", "")
    print(line)

    # Check for equals
    if '=' not in line:
        return

    expression = line.split("=")

    operands = re.split(r'[-+*/()]', expression[1])
    operators = re.findall(r'[-+*/()]', expression[1])

    # print(operands)
    # print(operators)

    setVarCode = ""
    operationCode = ""

    for op in operators:
```

```
if op not in operatorTable:
    raise "Invalid Operator " + op

operationCode += operatorTable[op] + " "

# Get Operand 1 and its register
operand1 = operands.pop(0)

# Allocate Register to operand
if operand1 not in registerTable.keys():
    setVarCode += f"MOV {operand1},"

register1 = getRegisterByOperand(operand1)

if len(setVarCode) > 1:
    setVarCode += register1

# Get Operand 2 and its register
operand2 = operands.pop(0)

if operand2 in registerTable.keys():
    operand2 = getRegisterByOperand(operand2)
```

```
        operationCode += f"{operand2},{register1}"

    if len(setVarCode) > 0:
        print(setVarCode)
    if len(operationCode) > 0:
        print(operationCode)
    print(f"{expression[0]} : {getRegisterByOperand(expression[0])}")
    print()


code = open("6-code.txt").readlines()

for line in code[:]:
    parseLine(line)
```

PROBLEMS DEBUG CONSOLE TERMINAL PORTS

```
    code = open("Exp 6\6-code.txt").readlines()
           ^^^^^^^^^^^^^^^^^^^^^^^^^
OSError: [Errno 22] Invalid argument: 'Exp 6\x06-code.txt'
PS C:\Users\ivana\Desktop\College\Third Year\SEM 6\SPCC Pracs> python -u "c:\Users\ivana\Desktop\College\Third Year\SEM 6\SPCC Pracs\Exp 6\e6.py"
t=b+c
MOV b,R0
ADD c,R0
t : R1

v=d+e
MOV d,R2
ADD e,R2
v : R3

u=t-v
SUB R3,R1
u : R4

w=t+u
ADD R4,R1
w : R5

a=w
a : R6

PS C:\Users\ivana\Desktop\College\Third Year\SEM 6\SPCC Pracs>
```

Code
Code

IVAN DSOUZA
9601
T.E. COMPS A

Exp 6

Q1]

Ans The code generator phase is a compiler translates the intermediate representation of the source code into executable machine.

The code generator has the following issues:

i) Target Architecture: designing the code generator to abstract away target arch. intricacies while efficiently utilizing its features is critical

ii) Optimization: Balancing code size and execution speed while applying various techniques is challenging.

iii) Target Language Representation: Efficiently mapping high level language constructs to M/C code instructions & Data structures is essential

iv) Handling Control flow: Generating code that accurately reflects control flow.

v) Error Handling: Dealing with errors during code generation.

Q2)

Ans) Basic blocks are fundamental units of code used in various computer optimizations and analysis. It has the following properties.

i) Single Entry, Single Exit: Simplifies the control flow analysis

ii) No internal Branching: Enables treating basic ~~flows~~ blocks as linear sequences of instructions.

iii) Atomic Execution: Ensures sequential execution without interruption for correct program semantics.