BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to University of Mumbai)
[Knowledge is Nectar]
Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

Aim	The aim is to simulate code generation, managing registers, and variables, ensuring correct data movement for arithmetic operations.
Objective	Develop a code generator to manage registers and variables, ensuring accurate data flow during arithmetic operations in simulated environments.

Theory	**Code Generator**
	
	Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.
	
	Example:
	
	Consider the three address statement x:= y + z. It can have the following sequence of codes:
	
	MOV x, R_0
	
	ADD y, R_0
	
	Register and Address Descriptors:
	
	o A register descriptor contains the track of what is currently in each register. The register descriptors show that all the registers are initially empty.
	o An address descriptor is used to store the location where current value of the name can be found at run time.
	
	A code-generation algorithm:
	
	The algorithm takes a sequence of three-address statements as input. For each three address statement of the form a:= b op c perform the various actions. These are as follows:
	
	1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.

2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction **MOV y' , L** to place a copy of y in L.
3. Generate the instruction **OP z' , L** where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptor.
4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

Generating Code for Assignment Statements:

The assignment statement d:= (a-b) + (a-c) + (a-c) can be translated into the following sequence of three address code:

1. t:= a-b
2. u:= a-c
3. v:= t +u
4. d:= v+u

Code sequence for the example is as follows:

Statement	Code Generated	Register descriptor Register empty
t:= a - b	MOV a, R0 SUB b, R0	R0 contains t
u:= a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u
v:= t + u	ADD R1, R0	R0 contains v R1 contains u
d:= v + u	ADD R1, R0 MOV R0, d	R0 contains d

Code Generation Process

The experiment involves simulating a simplified code generation process to understand how compilers translate high-level code into machine instructions. This process is crucial for understanding computer architecture, programming languages, and optimization techniques.

1. Code Generation Process Overview:

Code generation is a key stage in the compilation process where high-level code (e.g., from a programming language like Python) is translated into low-level machine instructions executable by hardware. This process typically involves several steps: parsing the input code, semantic analysis, optimization, and finally, generating target code.

2. Register Allocation:

Registers are small, fast storage locations within the CPU used to hold data temporarily during program execution. Register allocation involves assigning variables or intermediate results to these registers to optimize performance. In this experiment, we simulate register allocation for a simplified architecture with a fixed number of registers.

3. Address Descriptors:

Address descriptors are data structures used by compilers to track the location of variables or data in memory or registers. In our experiment, we use a dictionary to map variable names to their corresponding address descriptors, which include information about their current location, such as register name or memory address.

4. The CodeGenerator Class:

The heart of our experiment is the `CodeGenerator` class, which encapsulates the logic for generating machine code instructions. It maintains registers, address descriptors, and a list to store generated code. The `generate_code` method processes each statement, allocating registers, generating instructions, and updating address descriptors accordingly.

5. Generating Machine Code:

We parse each statement to identify operands and operators. We allocate a register for the result and load operands into registers if necessary. We then generate machine code instructions based on the operation (e.g., addition, subtraction). Finally, we update the address descriptors to reflect the new location of variables.

6. Handling Last Statement:

To ensure correctness, we add logic to handle the last statement. If it is the final operation in the sequence, we move the result from the register back to the corresponding variable, ensuring that the final value is stored appropriately.

7. PrettyTable Display:

We use the PrettyTable library to display the experiment results in a tabular format. Each row of the table represents a statement, showing the generated code, register descriptor, and address descriptor for the variable.

10. Applications:

Understanding code generation is essential for compiler developers, system programmers, and anyone interested in understanding how high-level code is translated into machine instructions. The knowledge gained from this experiment is applicable to various domains, including software engineering, computer architecture, and optimization techniques.

Implementation/Code

```
import prettytable as pt

class CodeGenerator:
    def __init__(self):
        self.registers = {f"R{i}": None for i in range(4)}  # Simulate 4
registers
        self.address_descriptors = {}  # Map variable names to address
descriptors
        self.code = []  # List to store generated machine code instructions
(simplified)

    def getreg(self, var_name):
    # Check if variable is already in a register
        if var_name in self.address_descriptors:
```

```
            descriptor = self.address_descriptors[var_name]
            if isinstance(descriptor["location"], str):  # Check if location is
a register name
                # Free up the current register
                current_register = descriptor["location"]
                self.registers[current_register] = None
        # Find an empty register
        for reg in self.registers:
            if self.registers[reg] is None:
                self.registers[reg] = var_name
                return reg


    def generate_code(self, statement, is_last_statement=False):
        parts = statement.split()  # Split the statement into words
        if len(parts) < 3:
            raise ValueError(f"Invalid statement format: {statement}")
        operand, op, operand2 = parts[2:]  # Get first three words (assuming op
is binary)

        result_reg = self.getreg(operand)
        var = parts[0]
        # Handle operand (assuming it's already in a register or memory)
        operand_descriptor = self.address_descriptors.get(operand)
        operand_loc = operand_descriptor["location"] if operand_descriptor else
operand

        # Handle operand2 (assuming it's already in a register or memory)
        operand2_descriptor = self.address_descriptors.get(operand2)
        operand2_loc = operand2_descriptor["location"] if operand2_descriptor
else operand2

        # Generate instructions (replace with actual machine code for specific
architecture)
        if operand_loc != result_reg:
            self.code.append(f"MOV {operand_loc}, {result_reg}")  # Move operand
if needed
        if op == "+":
            self.code.append(f"ADD {operand2_loc}, {result_reg}")
        elif op == "-":
            self.code.append(f"SUB {operand2_loc}, {result_reg}")

        # If it's the last statement, move the result from register to variable
        if is_last_statement:
            self.code.append(f"MOV {result_reg}, {var}")

        # Update address descriptors
```

```
        self.address_descriptors[var] = {"location": result_reg}

        # Return generated code and reset for the next statement
        generated_code = "\n".join(self.code)
        self.code = []  # Reset code for next statement
        return generated_code

# Initialize PrettyTable
table = pt.PrettyTable(["Statement", "Generated Code", "Register Descriptor",
"Address Descriptor"])

codegen = CodeGenerator()
statements = ["t = a - b", "u = a - c","v = t + u","d = v + u"]

def find_next_use(x,index):
    if index+1 == len(statements) and x == letters[-1]:
        return True
    for s in statements[index+1::]:
        if x in s:
            return True
    return False
letters = [x[0] for x in statements ]
# Generate code and populate the table
for i, statement in enumerate(statements):
    generated_code = codegen.generate_code(statement, is_last_statement=(i ==
len(statements) - 1))
    address_descriptor =
codegen.address_descriptors[statement.split()[0]]["location"]
    actual_adr = []
    for l in letters:
        if find_next_use(l,i):
            if l in codegen.address_descriptors:

actual_adr.append((l,codegen.address_descriptors[l]["location"]))
    reg_desc = ''
    for a in actual_adr:
        reg_desc+=f'{a[1]} contains {a[0]}\n'
    adr_desc = ''
    for a in actual_adr:
        adr_desc+=f'{a[0]} in {a[1]}\n'
    if i+1 == len(statements):
        adr_desc+=f'{letters[-1]} in memory'
    result_reg = address_descriptor if address_descriptor is not None else "N/A"
    table.add_row([statement, generated_code, reg_desc, adr_desc])

print("Table:")
print(table)
```

Output

```
pranaysinghvi@Pranays-MacBook-Air 7_ % python3 Experiment\ 7.py
Table:
+-----------+---------------+---------------------+--------------------+
| Statement | Generated Code | Register Descriptor | Address Descriptor |
+-----------+---------------+---------------------+--------------------+
| t = a - b |   MOV a, R0    |    R0 contains t    |      t in R0       |
|           |   SUB b, R0    |                     |                    |
| u = a - c |   MOV a, R1    |    R0 contains t    |      t in R0       |
|           |   SUB c, R1    |    R1 contains u    |      u in R1       |
|           |                |                     |                    |
| v = t + u |   ADD R1, R0   |    R1 contains u    |      u in R1       |
|           |                |    R0 contains v    |      v in R0       |
|           |                |                     |                    |
| d = v + u |   ADD R1, R0   |    R0 contains d    |      d in R0       |
|           |   MOV R0, d    |                     |    d in memory     |
+-----------+---------------+---------------------+--------------------+
```

Conclusion

In conclusion, the experiment deepened my understanding of code generation, registers, and data movement in compiler development processes.

References

[1] Javatpoint: Code Generator
https://www.javatpoint.com/code-generation

[2] ChatGPT (April 22, 2024) Code Generation
https://chat.openai.com/share/8a0f7ed2-da00-4b73-bc45-057b2842b928