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)

UID	2021300101
Name	Adwait Purao
Class and Batch	TE Computer Engineering - Batch B
Date	25-04-24
Lab #	7
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
	
	The code generator is a crucial component in a compiler, responsible for translating the intermediate representation of the source code into executable machine code or assembly language. Its primary objective is to produce efficient target code for three-address statements.[1]
	
	### Register Utilization
	
	To generate code, the code generator employs registers to store the operands of three-address statements. For instance, consider the three-address statement `x := y + z`. It can be translated into the following sequence of code:[1]
	
	MOV x, R0
	ADD y, R0
	
	## Register and Address Descriptors
	
	Two essential data structures assist the code generator in keeping track of values and their locations:

1. **Register Descriptor**: This descriptor maintains information about the contents of each register. Initially, all registers are considered empty according to the register descriptors.
2. **Address Descriptor**: This descriptor stores the runtime location where the current value of a name (variable or temporary) can be found.

Code Generation Algorithm

The code generation algorithm takes a sequence of three-address statements as input. For each three-address statement of the form `a := b op c`, the algorithm performs the following actions:

1. Invoke a function `getreg` to determine the location `L` where the result of the computation `b op c` should be stored.
2. Consult the address descriptor for `y` to determine `y'`, which represents the current location of `y`. If the value of `y` is present in both memory and a register, prefer the register `y'`. If `y` is not already in `L`, generate the instruction `MOV y', L` to copy the value of `y` into `L`.
3. Generate the instruction `OP z', L`, where `z'` represents the current location of `z`. If `z` is present in both memory and a register, prefer a register location. Update the address descriptor of `x` to indicate that `x` is now in location `L`. If `x` is already in `L`, update its descriptor and remove `x` from all other descriptors.
4. If the current values of `y` or `z` have no further uses, or if they are not live on exit from the block, or if they are not in registers, update the register descriptor to indicate that after executing `x := y op z`, those registers no longer contain `y` or `z`.

Generating Code for Assignment Statements

Consider the assignment statement `d := (a-b) + (a-c) + (a-c)`. It 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`

The code sequence for this example is as follows:

The code sequence for this example is as follows:

Statement	Code Generated	Register Descriptor	Address Descriptor
`t := a - b`	`MOV a, R0` `SUB b, R0`	`R0` contains `t`	`t` in `R0`
`u := a - c`	`MOV a, R1` `SUB c, R1`	`R0` contains `t` `R1` contains `u`	`t` in `R0` `u` in `R1`
`v := t + u`	`ADD R1, R0`	`R0` contains `v` `R1` contains `u`	`u` in `R1` `v` in `R0`
`d := v + u`	`ADD R1, R0` `MOV R0, d`	`R0` contains `d`	`d` in `R0` and memory

In this table, we can observe the sequence of instructions executed, the values stored in registers, and the memory locations accessed or updated during the code generation process.

Issues in Code Generation

Input to Code Generator

The input to the code generator is the intermediate code generated by the front-end of the compiler, along with information from the symbol table that determines the run-time addresses of the data objects denoted by the names in the intermediate representation. The intermediate code can be represented in various forms, such as quadruples, triples, indirect triples, postfix notation, syntax trees, DAGs, etc. The code generation phase assumes that the input is free from all syntactic and semantic errors, necessary type checking has taken place, and type-conversion operators have been inserted wherever required. [2]

Target Program

The target program is the output of the code generator. The output can be in one of the following forms:

1. **Absolute Machine Language**: In this form, the code can be placed in a fixed memory location and executed immediately. For example, the WATFIV compiler produces absolute machine code as output.
2. **Relocatable Machine Language**: This form allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by a linking loader, but there is an added expense of linking and loading.

3. **Assembly Language**: Generating assembly language as output makes the code generation easier. Symbolic instructions can be generated, and the macro facilities of assemblers can be utilized in code generation. However, an additional assembly step is required after code generation.

Memory Management

Mapping the names in the source program to the addresses of data objects is done by the front-end and the code generator. A name in the three-address statements refers to the symbol table entry for the name, from which a relative address can be determined.

Instruction Selection

Selecting the best instructions can improve the efficiency of the program. Instruction selection should ensure that the instructions are complete and uniform. Instruction speeds and machine idioms also play a major role when efficiency is considered. If efficiency is not a concern, instruction selection is straightforward. For example, the three-address statements `P := Q + R` and `S := P + T` can be translated into the following code sequence:

MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S

However, the fourth statement is redundant since the value of P has already been stored in the previous statement, leading to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. Prior knowledge of instruction costs is needed to design good sequences, but accurate cost information is difficult to predict.

Register Allocation Issues

The use of registers makes computations faster compared to memory access, so efficient utilization of registers is important. The use of registers is subdivided into two subproblems:

1. **Register Allocation**: Selecting the sets of variables that will reside in registers at each point in the program.
2. **Register Assignment**: Picking the specific register to access the variable.

To understand this concept, consider the following three-address code sequence:

t := a + b
t := t * c
t := t / d

An efficient machine code sequence for this would be:

MOV a, R0
ADD b, R0
MUL c, R0
DIV d, R0
MOV R0, t

Evaluation Order

The code generator decides the order in which the instructions will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require fewer registers to hold intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.[2]

Approaches to Code Generation Issues

The code generator must always generate correct code, as it is essential due to the number of special cases it might face. Some of the design goals of a code generator are:

- Correctness
- Maintainability
- Testability
- Efficiency

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","g = 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)
```

<table>
<tr><td></td><td></td></tr>
<tr><td>Output</td><td>
<pre>
aspur@LAPTOP-LG4IQEFB MINGW64 ~/OneDrive/SPCC/EXPERIMENTS/07. Code Generation Algorithm

$ python 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
g = v + u	ADD R1, R0	R0 contains g	g in R0
	MOV R0, g		g in memory
+-----------+---------------+--------------------+--------------------+
</pre>
</td></tr>
<tr><td>Conclusion</td><td>In conclusion, I've gained invaluable insights into how compilers translate high-level programming constructs into efficient machine code, optimizing resource utilization and enhancing program performance. This practical experience has not only solidified theoretical concepts but also provided a deeper appreciation for the intricate interplay between software and hardware in the compilation process. Overall, this experiment has been instrumental in broadening my expertise and refining my skills in compiler design and implementation.</td></tr>
<tr><td>References</td><td>[1] Javatpoint: Code Generator
https://www.javatpoint.com/code-generation

[2] Issues in the design of a code generator
https://www.geeksforgeeks.org/issues-in-the-design-of-a-code-generator/</td></tr>
</table>