Fr. Conceicao Rodrigues College of Engineering
Department of Computer Engineering

Academic Term : Jan-May 2024 - 25

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

Practical No:	4
Title:	Generate a target code for the optimized code.
Date of Performance:	11/03/2025
Date of Submission:	21/03/2025
Roll No:	9913
Name of the Student:	Mark Lopes

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 4

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 descriptors.

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 registers no longer will contain y and z, reply.

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 is 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 be found, select the memory location of x as L.

Conclusion:

Code:

```
#include <stdio.h>

#include <string.h>

#include <stdbool.h>

#include <stdlib.h>


#define MAX_REGISTERS 4

#define MAX_VARIABLES 26  // One for each letter of the alphabet

#define MAX_STATEMENTS 50

#define MAX_EXPR_LEN 100


// Register descriptor - stores which variable is held in which register

char registers[MAX_REGISTERS][10];

bool is_register_free[MAX_REGISTERS];


// Address descriptor - stores the current location of each variable

typedef struct {

    char location[10]; // "MEMORY" or "REGISTER"

    int register_num;  // Register number if in register

} AddressDesc;


AddressDesc address_descriptor[MAX_VARIABLES];


// Structure to store the statements
```

```
typedef struct {
    char x[10];
    char y[10];
    char op[10];
    char z[10];
} Statement;

// Array to store all statements
Statement statements[MAX_STATEMENTS];

// Utility functions
int get_register_for(char *var) {
    // Check if var is a single letter
    if (strlen(var) == 1 && var[0] >= 'a' && var[0] <= 'z') {
        int idx = var[0] - 'a';
        if (strcmp(address_descriptor[idx].location, "REGISTER") == 0) {
            return address_descriptor[idx].register_num;
        }
    }

    // Alternative method: check the register descriptor
    for (int i = 0; i < MAX_REGISTERS; i++) {
        if (strcmp(registers[i], var) == 0) {
            return i;
        }
    }
    return -1;
```

```
}

int get_free_register() {
    for (int i = 0; i < MAX_REGISTERS; i++) {
        if (is_register_free[i]) {
            return i;
        }
    }
    return -1;
}

// Get index in address_descriptor for a variable
int get_var_index(char *var) {
    // Simple mapping: 'a' -> 0, 'b' -> 1, etc.
    if (strlen(var) == 1 && var[0] >= 'a' && var[0] <= 'z') {
        return var[0] - 'a';
    }
    return -1;  // Invalid variable name
}

// Map operator to assembly instruction
void map_operator(char *op, char *assembly_op) {
    if (strcmp(op, "-") == 0) {
        strcpy(assembly_op, "SUB");
    } else if (strcmp(op, "+") == 0 || strcmp(op, "=") == 0) {
        strcpy(assembly_op, "ADD");
    } else if (strcmp(op, "*") == 0) {
        strcpy(assembly_op, "MUL");
    } else if (strcmp(op, "/") == 0) {
        strcpy(assembly_op, "DIV");
    } else {
        // Default case - use the operator as is
        strcpy(assembly_op, op);
    }
}

int getreg(char *x, char *y, char *z) {
    // Check if y is in a register
    int y_reg = get_register_for(y);
    if (y_reg != -1) {
        return y_reg;
    }

    // Try to find a free register
    int free_reg = get_free_register();
    if (free_reg != -1) {
        return free_reg;
    }

    // If no free registers, make one free by spilling
    // (For simplicity, just use R0)
    return 0;
}
```

```
void generate_code(char *x, char *y, char *op, char *z) {
    int reg = getreg(x, y, z);
    int y_reg = get_register_for(y);
    int z_reg = get_register_for(z);
    int x_index = get_var_index(x);

    char assembly_op[10];
    map_operator(op, assembly_op);

    // Handle Y operand
    if (y_reg == -1) {
        // Y is not in a register, load it
        printf("MOV %s, R%d\n", y, reg);
    } else {
        // Y is already in register, use that register
        reg = y_reg;
    }

    // Handle operation with Z operand
    if (z_reg == -1) {
        printf("%s %s, R%d\n", assembly_op, z, reg);
    } else {
        printf("%s R%d, R%d\n", assembly_op, z_reg, reg);
    }

    // Update descriptors
    if (x_index >= 0 && x_index < MAX_VARIABLES) {
        strcpy(registers[reg], x);
        strcpy(address_descriptor[x_index].location, "REGISTER");
        address_descriptor[x_index].register_num = reg;
        is_register_free[reg] = false;
    }
}

// Parse an expression like "t = a - b" into components
bool parse_expression(char *expr, Statement *stmt) {
    char *token;
    char *saveptr;

    // Make a copy of the expression to tokenize
    char expr_copy[MAX_EXPR_LEN];
    strcpy(expr_copy, expr);

    // Get the left side (result variable)
    token = strtok_r(expr_copy, "=", &saveptr);
    if (token == NULL) return false;

    // Trim spaces
    while (*token == ' ') token++;
    char *end = token + strlen(token) - 1;
    while (end > token && *end == ' ') end--;
    *(end + 1) = '\0';

    strcpy(stmt->x, token);
```

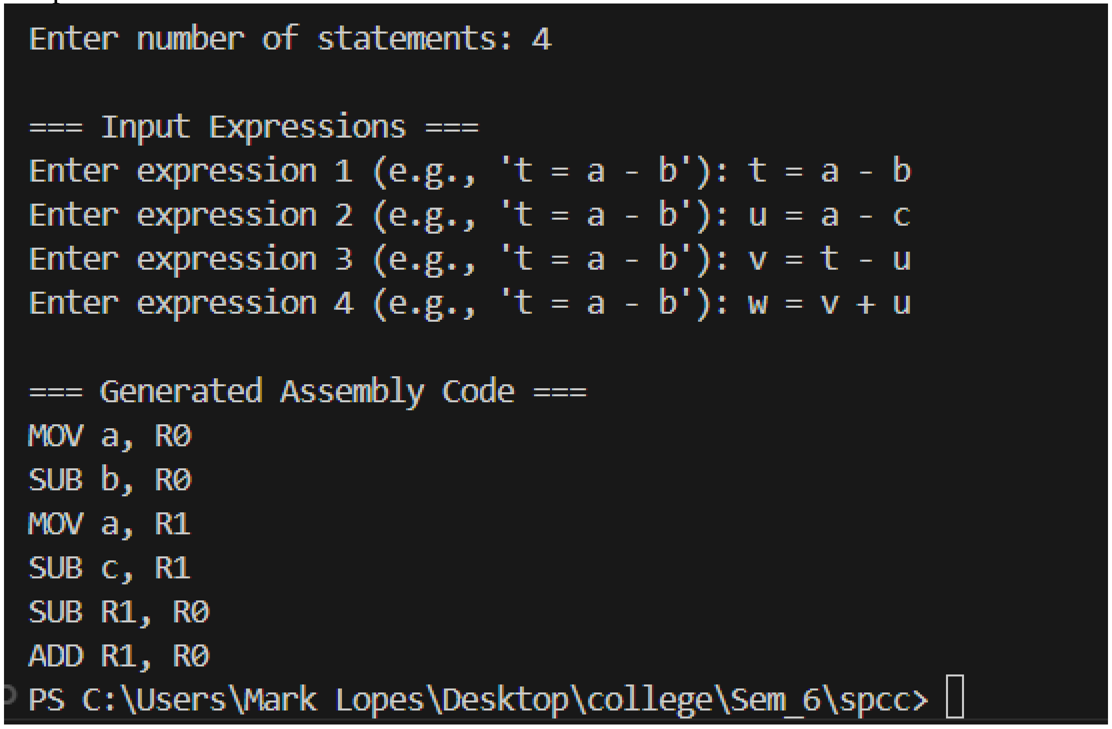
```
// Get the right side of the expression
token = strtok_r(NULL, "", &saveptr);
if (token == NULL) return false;

// Trim spaces
while (*token == ' ') token++;

// Find the operator
char *op_pos = NULL;
if (strchr(token, '+') != NULL) op_pos = strchr(token, '+');
else if (strchr(token, '-') != NULL) op_pos = strchr(token, '-');
else if (strchr(token, '*') != NULL) op_pos = strchr(token, '*');
else if (strchr(token, '/') != NULL) op_pos = strchr(token, '/');

if (op_pos == NULL) {
    // No operator found - just a single variable on the right
    strcpy(stmt->y, token);
    strcpy(stmt->op, "="); // assignment
    strcpy(stmt->z, "0");  // dummy value
    return true;
}

// Extract the operands and operator
char op = *op_pos;
*op_pos = '\0'; // Split the string at the operator
```

```
    // Get the first operand
    char *first_operand = token;
    while (*first_operand == ' ') first_operand++;
    end = first_operand + strlen(first_operand) - 1;
    while (end > first_operand && *end == ' ') end--;
    *(end + 1) = '\0';

    // Get the second operand
    char *second_operand = op_pos + 1;
    while (*second_operand == ' ') second_operand++;
    end = second_operand + strlen(second_operand) - 1;
    while (end > second_operand && *end == ' ') end--;
    *(end + 1) = '\0';

    strcpy(stmt->y, first_operand);
    stmt->op[0] = op;
    stmt->op[1] = '\0';
    strcpy(stmt->z, second_operand);

    return true;
}

int main() {
    // Initialize registers and address descriptors
    for (int i = 0; i < MAX_REGISTERS; i++) {
        is_register_free[i] = true;
        registers[i][0] = '\0';
```

```c
}
for (int i = 0; i < MAX_VARIABLES; i++) {
    strcpy(address_descriptor[i].location, "MEMORY");
    address_descriptor[i].register_num = -1;
}

int num_statements;
printf("Enter number of statements: ");
scanf("%d", &num_statements);
getchar(); // Consume the newline

// First collect all statements
printf("\n=== Input Expressions ===\n");
for (int i = 0; i < num_statements; i++) {
    char expr[MAX_EXPR_LEN];
    printf("Enter expression %d (e.g., 't = a - b'): ", i + 1);
    fgets(expr, MAX_EXPR_LEN, stdin);
    expr[strcspn(expr, "\n")] = 0; // Remove newline

    if (!parse_expression(expr, &statements[i])) {
        printf("Error parsing expression: %s\n", expr);
        i--; // Try again
        continue;
    }
}

// Then generate code for all statements
```

```
    printf("\n=== Generated Assembly Code ===\n");
    for (int i = 0; i < num_statements; i++) {
        generate_code(statements[i].x, statements[i].y, statements[i].op, statements[i].z);
    }

    return 0;
}
```

Output:



```
Enter number of statements: 4

=== Input Expressions ===
Enter expression 1 (e.g., 't = a - b'): t = a - b
Enter expression 2 (e.g., 't = a - b'): u = a - c
Enter expression 3 (e.g., 't = a - b'): v = t - u
Enter expression 4 (e.g., 't = a - b'): w = v + u

=== Generated Assembly Code ===
MOV a, R0
SUB b, R0
MOV a, R1
SUB c, R1
SUB R1, R0
ADD R1, R0
PS C:\Users\Mark Lopes\Desktop\college\Sem_6\spcc>
```

Postlab:

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

SPCC Postlab 4

Q.1 The code generation phase of a compiler is responsible for translating intermediate code into target machine code.
Several design issues must be considered to ensure efficiency, correctness and optimal performance.
The design issues are:-

i) Intermediate Representation selection:
The of IR affects the complexity of code generation.

ii) Target machine Architecture Consideration
The code generator must be aware of the CPU register, instruction set, addressing modes and available memory.

iii) Instruction Selection:-
The generator must select the most efficient machine instruction for each operation.

iv) Register Allocation and assignment
Efficient way of registers minimize memory access and improves performance

v) optimization
local optimization (within a base block) and global optimization (across blocks)

Q.2 A basic block is a sequence of consecutive statements in a program where:

i) Control enters at the beginning and exits at the end without any jumps

ii) It has only one entry point and one exit point

Properties of basic blocks:-

i) Single Entry, single exit:
No branching into the middle of a basic block. Only the first instruction is the entry point.

ii) Sequential execution:
Instructions are executed one after another without any jumps or branches.

iii) Atomicity:
If any statement in a block executes, the entire block executes.

iv) Local optimization Scope:
Since basic blocks have no jumps inside them, local optimizations like constant folding, dead code elimination and common subexpression elimination can be ~~ca~~ easily applied.