**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';

    }

    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");

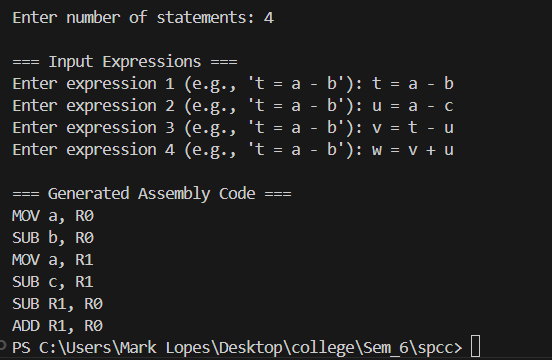
    for (int i = 0; i < num\_statements; i++) {

        generate\_code(statements[i].x, statements[i].y, statements[i].op, statements[i].z);

    }

    return 0;

}

Output:  


**Postlab:**

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

