Name: Munna Chauhan

Reg. No.: CH.EN.U4CSE22176

Lab Exp.: 08

-----------------------------------------------------------------------------------------------------------------------------------

Aim: To write a program that implements the target code generation.

Code:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

// Global variables

int label[20]; // Array to store instruction numbers that are jump targets

int no = 0; // Counter for the number of labels stored

// Function to check if a given instruction number 'k' is a jump target

int check\_label(int k) {

int i;

for (i = 0; i < no; i++) {

if (k == label[i])

return 1; // It is a jump target

}

return 0; // It is not a jump target

}

int main() {

FILE \*fp1, \*fp2;

char fname[10], op[10], ch;

char operand1[8], operand2[8], result[8];

int i = 0, j = 0;

printf("\n Enter filename of the intermediate code: ");

scanf("%s", fname);

// Open the intermediate code file for reading and the target file for writing

fp1 = fopen(fname, "r");

fp2 = fopen("target.txt", "w");

if (fp1 == NULL || fp2 == NULL) {

printf("\n Error opening the file");

exit(0);

}

// Process the intermediate code file line by line

while (!feof(fp1)) {

fprintf(fp2, "\n"); // New line for formatting in the target file

fscanf(fp1, "%s", op); // Read the operation/opcode

// Increment the instruction counter

i++;

// Check if the current instruction is a target of a previous jump

if (check\_label(i)) {

fprintf(fp2, "\nlabel#%d:", i); // Print the label

}

// --- Specific Operations (using strcmp for multi-character opcodes) ---

// PRINT operation

if (strcmp(op, "print") == 0) {

fscanf(fp1, "%s", result);

fprintf(fp2, "\n\t OUT %s", result);

}

// GOTO operation (Unconditional Jump)

else if (strcmp(op, "goto") == 0) {

fscanf(fp1, "%s %s", operand1, operand2); // Reads condition and target instruction number

fprintf(fp2, "\n\t JMP %s,label#%s", operand1, operand2);

label[no++] = atoi(operand2); // Store the target instruction number as a label

}

// Array assignment: []= (e.g., A[i] = B)

else if (strcmp(op, "[]=") == 0) {

fscanf(fp1, "%s %s %s", operand1, operand2, result);

// Assuming intermediate code is: []= A i B (meaning A[i] = B)

fprintf(fp2, "\n\t STORE %s[%s],%s", operand1, operand2, result);

}

// Unary Minus operation: uminus (e.g., T1 = uminus A)

else if (strcmp(op, "uminus") == 0) {

fscanf(fp1, "%s %s", operand1, result); // Reads operand and result

fprintf(fp2, "\n\t LOAD -%s,R1", operand1); // Load the negative value into R1

fprintf(fp2, "\n\t STORE R1,%s", result); // Store R1 into the result variable

}

// --- Arithmetic and Relational Operations (using switch for single-character opcodes) ---

else {

switch (op[0]) {

case '\*': // Multiplication: \* A B T1 (T1 = A \* B)

fscanf(fp1, "%s %s %s", operand1, operand2, result);

// NOTE: The original code's LOAD line is missing an operand. Correcting to a likely intent.

// Original: fprintf(fp2,"\n \tLOAD",operand1);

fprintf(fp2, "\n \t LOAD %s,R0", operand1);

fprintf(fp2, "\n \t LOAD %s,R1", operand2);

fprintf(fp2, "\n \t MUL R1,R0"); // R0 = R0 \* R1

fprintf(fp2, "\n \t STORE R0,%s", result);

break;

case '+': // Addition: + A B T1 (T1 = A + B)

fscanf(fp1, "%s %s %s", operand1, operand2, result);

fprintf(fp2, "\n \t LOAD %s,R0", operand1);

fprintf(fp2, "\n \t LOAD %s,R1", operand2);

fprintf(fp2, "\n \t ADD R1,R0"); // R0 = R0 + R1

fprintf(fp2, "\n \t STORE R0,%s", result);

break;

case '-': // Subtraction: - A B T1 (T1 = A - B)

fscanf(fp1, "%s %s %s", operand1, operand2, result);

fprintf(fp2, "\n\t LOAD %s,R0", operand1); // Load A into R0

fprintf(fp2, "\n \tLOAD %s,R1", operand2); // Load B into R1

fprintf(fp2, "\n \t SUB R1,R0"); // R0 = R0 - R1 (A - B)

fprintf(fp2, "\n \t STORE R0,%s", result);

break;

case '/': // Division: / A B T1 (T1 = A / B)

// NOTE: The original code has a typo: "%s %s s". Correcting to "%s %s %s".

fscanf(fp1, "%s %s %s", operand1, operand2, result);

fprintf(fp2, "\n \t LOAD %s,R0", operand1);

fprintf(fp2, "\n \t LOAD %s,R1", operand2);

fprintf(fp2, "\n \t DIV R1,R0"); // R0 = R0 / R1

fprintf(fp2, "\n \t STORE R0,%s", result);

break;

case '%': // Modulo (Using DIV instruction, which is often used for MOD/REM)

fscanf(fp1, "%s %s %s", operand1, operand2, result);

fprintf(fp2, "\n \t LOAD %s,R0", operand1);

fprintf(fp2, "\n \t LOAD %s,R1", operand2);

fprintf(fp2, "\n \t DIV R1,R0"); // In many architectures, DIV sets a remainder register.

// This code simply uses DIV and stores R0, which is likely incorrect for MOD.

// Sticking to the code's original instruction pattern.

fprintf(fp2, "\n \t STORE R0,%s", result);

break;

case '=': // Assignment: = A T1 (T1 = A)

fscanf(fp1, "%s %s", operand1, result);

// NOTE: The instruction STORE is commonly used for this, but the original code is STORE %s %s.

// Correcting to a more standard pattern: LOAD into a register, then STORE.

// Sticking to the code's original instruction pattern, assuming it means STORE operand1 to result.

fprintf(fp2, "\n\t STORE %s, %s", operand1, result);

break;

case '>': // Greater Than Conditional Jump: > A B target (If A > B, goto target)

j++;

fscanf(fp1, "%s %s %s", operand1, operand2, result); // Reads A, B, and target instruction number

fprintf(fp2, "\n \t LOAD %s,R0", operand1); // Load the first operand A into R0

fprintf(fp2, "\n\t JGT %s,label#%s", operand2, result); // Jump if Greater Than

label[no++] = atoi(result);

break;

case '<': // Less Than Conditional Jump: < A B target (If A < B, goto target)

fscanf(fp1, "%s %s %s", operand1, operand2, result);

fprintf(fp2, "\n \t LOAD %s,R0", operand1);

// NOTE: The original code has a typo: label#%d. Correcting to label#%s to match 'result' being a string.

fprintf(fp2, "\n\t JLT %s,label#%s", operand2, result); // Jump if Less Than

label[no++] = atoi(result);

break;

}

}

}

// Close and reopen the target file to read and display the generated code

fclose(fp2);

fclose(fp1);

fp2 = fopen("target.txt", "r");

if (fp2 == NULL) {

printf("Error opening the file\n");

exit(0);

}

// Print the generated target code to the console

printf("\n\nGenerated Target Code:\n");

do {

ch = fgetc(fp2);

printf("%c", ch);

} while (ch != EOF);

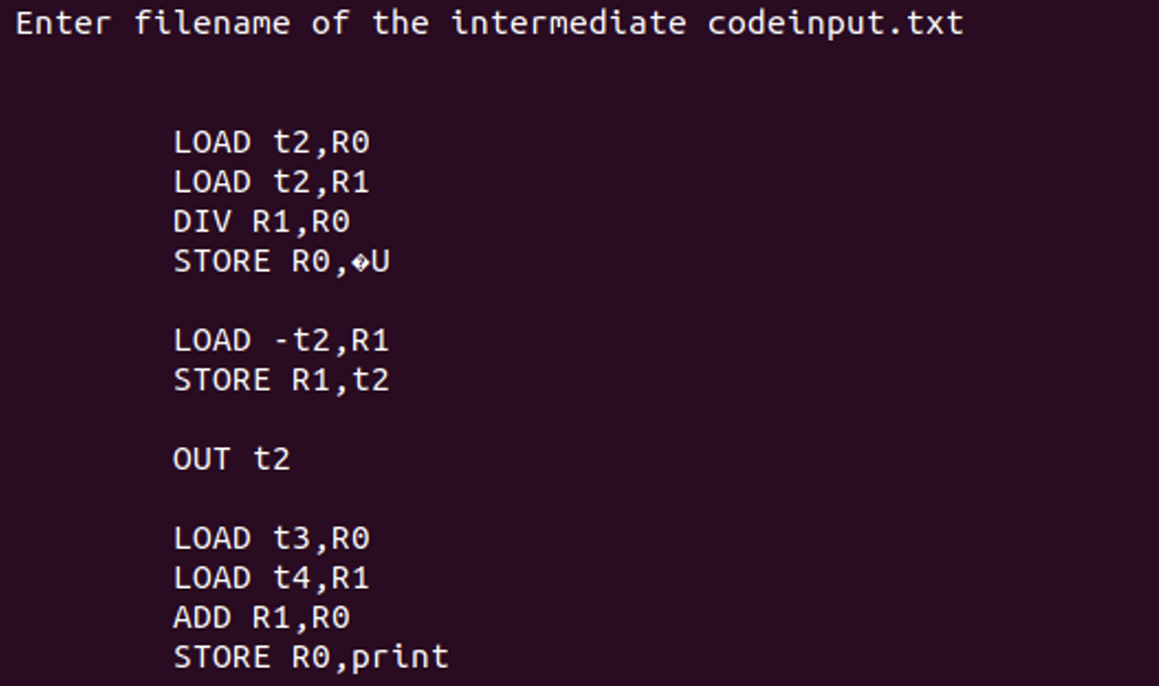
fclose(fp2);

// NOTE: The original code tries to close fp1 again here, which is redundant.

return 0;

}

Output:



Result: Thus, the program to implement the target code generation has been executed successfully.