DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COLLEGE OF ENGINEERING GUINDY,

ANNA UNIVERSITY

SUBSET OF C TO MIPS COMPILER

Mini Project – CS6109 Compiler Design

AKSHARA ACHUTHAN	2022103060
ANAND KARTHIKEYAN S	2022103305
ASHWIN ARUL M	2022103306

DEPARTMENT: CSE

BATCH: P

DATE: 14-11-2024

ABSTRACT

This project implements a compiler that translates a subset of the C programming language into MIPS assembly code. The compiler utilizes **Lex** for lexical analysis, **Yacc** for syntax parsing, and **CtoMIPS.c** for the generation of MIPS code. The system is designed to handle basic C constructs such as arithmetic operations, conditional statements, loops, and functions. It outputs MIPS assembly code, which can be tested on a **SPIM emulator**. The core components include grammar definitions, tokenization, parsing for semantic errors, constructing an Abstract Syntax Tree (AST), and using a Symbol Table for variable tracking.

INTRODUCTION

The purpose of this project is to develop a compiler that translates a subset of the C programming language into MIPS assembly code, utilizing **Lex** for tokenization and **Yacc** for parsing. The output MIPS code can then be tested using a **SPIM emulator**, which simulates the execution of MIPS instructions.

This compiler supports basic constructs found in C, including arithmetic expressions, control structures like if statements, for and while loops, and function definitions.

cGrammer : Grammar used for this project

clexer.l : lexer file for defining valid language tokens

cparser.y : Yacc based parser. Checks for semantic errors and generates Abstract Syntax Tree and outputs final MIPS code.

CtoMIPS.c : takes AST as input and generates MIPS code into mips_code.s

symbolTable.c : defines helping functions used for creating Symbol Table.

definition.c : helping functions for generating AST.

definition.h : defines structure for AST and Symbol Table.

test.c : some test programs for testing.

The workflow begins with the user running the command make, which compiles the necessary files into an executable. By passing a C file (e.g., example.c) into the compiler, the tool generates a corresponding MIPS assembly file (mips_code.s) that can be executed on a MIPS emulator. The project serves as a practical tool for understanding the inner workings of a compiler, with a focus on translating high-level C constructs into low-level MIPS assembly language.

METHODOLOGY

Below is a step-by-step overview of how the compiler processes the input C code and generates MIPS code:

1. Lexical Analysis (Lex)

The first phase of the compilation process involves lexical analysis, performed by the **Lex** tool. The role of the lexical analyzer is to read the raw input C source code and break it down into a sequence of tokens. These tokens include:

- **Keywords** (e.g., int, if, for, while)
- **Operators** (e.g., +, -, *, /)
- **Identifiers** (variable names and function names)
- **Literals** (integer constants)
- **Punctuation** (e.g., parentheses (), braces {}, semicolons ;)

The Lex file (clexer.l) defines patterns for each of these tokens using regular expressions. These patterns are used to recognize and classify the input C code. The Lex tool generates a lexer that will scan the C code and output tokens, which will then be passed to the parser.

2. Syntax Analysis (Yacc)

The **Yacc** tool is used for the syntax analysis, which ensures that the input C code conforms to the grammar defined in the **cGrammer** file. The parser created by Yacc processes the tokens generated by the lexical analyzer and constructs an Abstract Syntax Tree (AST).

- **Grammar Definition:** The grammar defines the syntactic structure of the C subset, including rules for arithmetic expressions, control structures (such as if, for, while), and function definitions.
- **Parsing Process:** Yacc checks for errors in the syntax of the C code based on the grammar rules. If any syntactic issues are found, an error is raised. Otherwise, the parser constructs an AST, which represents the hierarchical structure of the program.

The **cparser.y** file defines the parsing rules and includes semantic actions to handle errors and AST generation. The AST is a tree-like structure that captures the relationships between different parts of the code, such as expressions, statements, and function calls.

3. Symbol Table Construction

The **symbolTable.c** file is responsible for managing the **Symbol Table**, which tracks information about variables, such as their names, types, and memory locations. During the parsing phase, the compiler will use the symbol table to store and retrieve information about variables and functions.

- **Variable Tracking:** When variables are declared (e.g., int x;), their information is added to the symbol table.
- **Scope Management:** The symbol table helps manage variable scopes, ensuring that variables are only accessed within their defined scope (e.g., inside functions or loops).
- **Function Handling:** The symbol table also manages function definitions and their parameters.

The symbol table is crucial for generating correct MIPS code, as it helps determine the location of variables and ensures that the generated code accesses variables correctly.

4. Abstract Syntax Tree (AST) Generation

The AST represents the logical structure of the C code in a tree format. Each node in the tree corresponds to a construct in the C program, such as a statement, expression, or declaration.

- **Node Types:** The AST will contain nodes for expressions (e.g., addition or multiplication), control structures (e.g., if statements or loops), and declarations (e.g., variable definitions).
- **Expression Evaluation:** During AST generation, the compiler performs semantic analysis and ensures that expressions are evaluated correctly based on operator precedence and operand types.

The **definition.c** and **definition.h** files define the structure of the AST, which allows for easy traversal and modification during the code generation phase.

5. Code Generation (CtoMIPS.c)

The final phase of the compilation process involves translating the AST into MIPS assembly code. The **CtoMIPS.c** file takes the AST as input and generates the corresponding MIPS instructions. The key steps in this phase include:

- **Arithmetic Operations:** The compiler translates arithmetic operations (e.g., addition, subtraction, multiplication, division) into MIPS instructions using registers for operand storage.

- **Control Structures:** The compiler generates MIPS code for control flow structures such as if statements, for loops, and while loops, using conditional branches (beq, bne, etc.) and jump instructions (j, jal).
- **Function Calls:** The compiler generates MIPS code for function calls, managing the stack and registers appropriately for parameter passing and return values.

For each construct in the AST, the code generator outputs a sequence of MIPS instructions. The MIPS code is written to a file (mips_code.s), which can then be loaded into a **SPIM emulator** for execution and testing.

6. Testing and Debugging

The compiler includes a set of test programs (in test.c) to verify that the generated MIPS code works as expected. These test cases cover basic arithmetic, control flow, and function calls. The generated MIPS code can be executed using the **SPIM emulator** to ensure that it behaves as intended.

- **Error Handling:** If the C code contains any syntax or semantic errors, the compiler will flag them during parsing or AST generation. These errors will be displayed to the user, providing helpful feedback for debugging.

7. Execution and Output

Once the compiler successfully translates the C code into MIPS assembly code, the resulting mips_code.s file can be executed on a MIPS simulator like SPIM. The user can run the emulator to observe the execution of the MIPS code and validate its correctness.

CODE IMPLEMENTATION

In this section, I will provide an overview of the key components of the compiler design project, including the relevant code snippets and explanations for each phase of the compiler.

1. Lexical Analysis

File Name: **clexer.l**

The lexical analyzer (lexer) is responsible for scanning the source code and converting it into a sequence of tokens. It identifies keywords, identifiers, literals, and operators using regular expressions.

```
clexer.l
D     [0-9]
L     [a-zA-Z_]

%{

#include "y.tab.h"
#include "definition.h"
extern YYSTYPE yylval;
int LINE=1;
%}

%%
"//"    { comment2();}
"/*"    { comment(); }
"break"   {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=BREAK;return(BREAK); }
"char"    {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=CHAR;return(CHAR); }
"continue"  {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=CONTINUE;return(CONTINUE);}
"else"    {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=ELSE;return(ELSE); }
"float"   {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=FLOAT;return(FLOAT); }
"for"   {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=FOR;return(FOR); }
"if"    {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=IF;return(IF); }
"int"   {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=INT;return(INT); }
"return"  {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=RETURN;return(RETURN); }
"sizeof"  {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=SIZEOF;return(SIZEOF); }
"struct"  {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=STRUCT;return(STRUCT); }
"void"    {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=VOID;return(VOID); }
"while"   {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=WHILE;return(WHILE); }
"read"    {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=READ;return(READ); }
"print"   {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=PRINT;return(PRINT);}
"max"   {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=MAX;return(MAX);}
"min"   {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=MIN;return(MIN);}
"swap"    {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=SWAP;return(SWAP);}
{L}({L}|{D})* { yylval.Sval.text=strdup(yytext); yylval.Sval.type=IDENTIFIER;return(IDENTIFIER); }

{D}+    { yylval.Sval.text=strdup(yytext); yylval.Sval.type=INT;return(CONSTANT); }
'(\\[t0n'"\\]|[^\\'])'  {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=CHAR;return(CONSTANT); }

{D}*"."{D}+ {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=FLOAT;return(CONSTANT); }

\"(\\.|[^\\"])*\" {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=STRING_LITERAL;return(STRING_LITERAL); }

">>="     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=RIGHT_ASSIGN;return(RIGHT_ASSIGN); }
"<<="     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=LEFT_ASSIGN;return(LEFT_ASSIGN); }
"+="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=AND_ASSIGN;return(ADD_ASSIGN); }
"-="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=SUB_ASSIGN;return(SUB_ASSIGN); }
"*="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=MUL_ASSIGN;return(MUL_ASSIGN); }
"/="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=DIV_ASSIGN;return(DIV_ASSIGN); }
"%="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=MOD_ASSIGN;return(MOD_ASSIGN); }
"&="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=AND_ASSIGN;return(AND_ASSIGN); }
"^="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=XOR_ASSIGN;return(XOR_ASSIGN); }
"|="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=OR_ASSIGN;return(OR_ASSIGN); }
">>"      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=RIGHT_OP;return(RIGHT_OP); }
"<<"      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=LEFT_OP;return(LEFT_OP); }
"++"      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=INC_OP;return(INC_OP); }
"--"      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=DEC_OP;return(DEC_OP); }
```

clexer.l

```
"||"      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=OR_OP;return(OR_OP); }
"<="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=LE_OP;return(LE_OP); }
">="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=GE_OP;return(GE_OP); }
"=="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=EQ_OP;return(EQ_OP); }
"!="      {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=NE_OP;return(NE_OP); }
";"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return(';'); }
("{")     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1; return('{'); }
("}")     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('}'); }
","     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return(','); }
"="     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('='); }
"("     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('('); }
")"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return(')'); }
("[")     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('['); }
("]")     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return(']'); }
"."     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('.'); }
"&"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('&'); }
"!"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('!'); }
"~"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('~'); }
"-"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('-'); }
"+"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('+'); }
"*"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('*'); }
"/"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('/'); }
"%"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('%'); }
"<"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('<'); }
">"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('>'); }
"^"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('^'); }
"|"     {  yylval.Sval.text=strdup(yytext); yylval.Sval.type=-1;return('|'); }

[ \t\v\n\f]   {  }
.     { /* ignore bad characters */ }

%%

yywrap()
{
  return(1);
}


comment()
{
  char c, c1;

  c=input();
  while(1){
    c1=input();
    if(c=='*' && c1=='/') break;
    c=c1;
  }
}

comment2(){
  char c;
  while ((c = input()) != '\n');
}
```

2. Syntax Analysis

File Name: **cparser.y**

The syntax analyzer (parser) checks the sequence of tokens against the grammar of the programming language to ensure that the structure of the code is valid. It constructs an Abstract Syntax Tree (AST) for further processing.

Sample Code:

cparser.y

```


print
        : PRINT '(' type ',' IDENTIFIER ')'
                {
                $$ = MakeNode(2); $$->type = 4; strcpy($$->lexeme,$1.text);
                $$->child[0] = $3; $3->parent = $$;
                $$->child[1] = MakeNode(0); strcpy($$->child[1]->lexeme,$5.text);
                $$->child[1]->type = $5.type;
                $$->child[1]->parent = $$;

                check.array = 0; check.struct_member = 0;
                struct symbolTable *temp = FindSymbol($5.text,check,currentT);
                if(temp==NULL) printf("not found %s\n",$5.text);
                else if(temp->type != $3->type) printf("PRINT type mismatch\n");
                $$->child[1]->where = temp;
              }
max
    : MAX '(' IDENTIFIER ',' IDENTIFIER ')'
              {
                $$ = MakeNode(2); $$->type = $3.type; strcpy($$->lexeme,$1.text);
                $$->child[0] = MakeNode(0); strcpy($$->child[0]->lexeme,$3.text);
                $$->child[1] = MakeNode(0); strcpy($$->child[1]->lexeme,$5.text);

                $$->child[1]->type = $5.type;
                $$->child[1]->parent = $$;
                $$->child[0]->type = $3.type;
                $$->child[0]->parent = $$;

                if($3.type != $5.type){
                  printf("max type error");
                  Totalerrors++;
                }

                struct symbolTable *temp = FindSymbol($3.text,check,currentT);
                if(temp==NULL) printf("not found %s\n",$3.text);
                $$->child[0]->where = temp;

                temp = FindSymbol($5.text,check,currentT);
                if(temp==NULL) printf("not found %s\n",$5.text);
                $$->child[1]->where = temp;

                $$->type = temp->type;

              }

min
    : MIN '(' IDENTIFIER ',' IDENTIFIER ')'
              {
                $$ = MakeNode(2); $$->type = $3.type; strcpy($$->lexeme,$1.text);
                $$->child[0] = MakeNode(0); strcpy($$->child[0]->lexeme,$3.text);
                $$->child[1] = MakeNode(0); strcpy($$->child[1]->lexeme,$5.text);

                $$->child[1]->type = $5.type;
                $$->child[1]->parent = $$;
```

3. Semantic Analysis

File Name: **definition.c**

The semantic analyzer checks for semantic errors in the AST, such as type mismatches and undeclared variables. It ensures that the code adheres to the semantic rules of the language.

Sample Code:

C definition.c > ...

```c
#include "definition.h"
#include "y.tab.h"
int init_count=0;

struct AST * MakeNode(int num){
    struct AST * node = (struct AST*)malloc(sizeof(struct AST));
    node->parent = NULL;
    node->NumChild = num;
    node->child = (struct AST**)malloc(num*sizeof(struct AST *));
    node->array = 0;
    node->pointer = 0;
    node->dim1 =0;
    node->dim2 = 0;
    node->order = 0;
    node->where = NULL;
    node->scope=currentT->scope;
    node->scopenode = currentT;
    return node;
}

void TerminalChild(struct AST * p,int num,char *text,int type){
    p->child[num] = MakeNode(0);
    p->child[num]->parent = p;
    strcpy(p->child[num]->lexeme,text);
    p->child[num]->type = type;
}

void AST_print(struct AST *t){
    int i;
    if(t->NumChild==0) return;
    struct AST *t2=t;
    for(i=0;i<t2->NumChild;++i)
        AST_print(t->child[i]);
    printf("\n%s  -->",t2->lexeme);
    for(i=0;i<t2->NumChild;++i) printf("%s ",t2->child[i]->lexeme);
}
```

4. Intermediate Code Generation

File Name: **CtoMIPS.c**

Intermediate code generation produces an intermediate representation of the source code, often in a lower-level format that is easier to translate into machine code. In this project, MIPS assembly code is generated.

Code:

CtoMIPS.c > GenerateMIPS()

```c
void GenerateMIPS(){
    lcount = 0;
    current = 0;
    myvar = 0;
    must = 0;
    if(checkmain()==0){
        printf("Error : main function not found\n");
        return;
    }
    FILE *fp;
    fp = fopen("mips_code.s","w");
    int i;
  // Data segment
    fprintf(fp,".data\n");
    fprintf(fp,"global: .word %d\n",sym->tableSize);

    // Text segment
    fprintf(fp,".text\n");
    fprintf(fp,".globl main\n\n");

    // Program entry point
    fprintf(fp,"# Program entry\n");
    fprintf(fp,"j main\n\n");
    // Generate all non-main functions
    for(i=0;i<funcount;++i){
        if(strcmp(ALL[i]->name,"main") != 0) {
            now = i;
            fprintf(fp,"%s:\n",ALL[i]->name);
            fprintf(fp,"\t li $t0, %d\n",ALL[i]->st->tableSize);
            fprintf(fp,"\t sub $sp, $sp, $t0\n");
            fprintf(fp,"\t move $t2, $sp\n");
            current = ALL[i]->st->scope;
            outputCode(fp,ALL[i]->t->child[1]);
            fprintf(fp,"\t li $t0, %d\n",ALL[i]->st->tableSize);
            fprintf(fp,"\t add $sp, $sp, $t0\n");
            fprintf(fp,"\t move $t2, $sp\n");
            fprintf(fp,"\t jr $ra\n\n");
        }
    }
    // Generate main function
    for(i=0;i<funcount;++i){
        if(strcmp(ALL[i]->name,"main") == 0) {
            now = i;
            fprintf(fp,"main:\n");
            fprintf(fp,"\tla $ra, exit\n");  // Store exit address in $ra
            fprintf(fp,"\t li $t0, %d\n",ALL[i]->st->tableSize);
            fprintf(fp,"\t sub $sp, $sp, $t0\n");
            fprintf(fp,"\t move $t2, $sp\n");
            current = ALL[i]->st->scope;
            outputCode(fp,ALL[i]->t->child[1]);
            fprintf(fp,"\t li $t0, %d\n",ALL[i]->st->tableSize);
            fprintf(fp,"\t add $sp, $sp, $t0\n");
            fprintf(fp,"\t move $t2, $sp\n");
            fprintf(fp,"\t jr $ra\n\n");
            break;
        }
    }
  // Add exit function at the bottom
    fprintf(fp,"exit:\n");
    fprintf(fp,"\tli $v0, 10\n");
    fprintf(fp,"\tsyscall\n");
    fclose(fp);
}
```

5. Code Generation

File Name: **CtoMIPS.c**

The code generation phase translates the intermediate representation into the final executable code. This involves generating machine code or assembly instructions for the target architecture.

Sample Code:

```c
void operations(FILE *fp, char *s) {
    if (!strcmp(s, "+")) {
        fprintf(fp, "\tadd $t1, $t0, $t1\n");
    } else if (!strcmp(s, "-")) {
        fprintf(fp, "\tsub $t1, $t0, $t1\n");
    }
    // Additional operations
}
```

Explanation: The **operations** function translates high-level operations (like addition and subtraction) into corresponding MIPS assembly instructions.

EXECUTION:

To compile and execute the compiler design project, the following sequence of commands is used:

Step 1: Generate the Parser

```bash
yacc -d cparser.y
```

This command uses yacc (Yet Another Compiler Compiler) to process the cparser.y file, which contains the grammar rules. The -d option generates the y.tab.h header file, which defines token types that will be shared with the lexical analyzer. The main output file, y.tab.c, includes C code for parsing, generated from the grammar rules.

Step 2: Generate the Lexical Analyzer

```bash
lex clexer.l
```

lex reads the clexer.l file, which defines the lexical analysis rules, and generates lex.yy.c. This file contains code for tokenizing the input program, transforming it into tokens that the parser can use for syntax analysis.

Step 3: Compile the Source Files

```bash
gcc lex.yy.c y.tab.c CtoMIPS.c symbolTable.c definition.c -o start.exe
```

The gcc command compiles all source files, including lex.yy.c, y.tab.c, and additional components like CtoMIPS.c, symbolTable.c, and definition.c, which manage symbol tables and translate intermediate code to MIPS assembly. The final executable file is named start.exe.

Step 4: Execute the Compiler

```bash
./start.exe <file.c
```

The executable, start.exe, processes the input C file, file.c, and generates MIPS assembly code as output. This output is stored in the mips_code.s file.

SCREENSHOTS:

Input: test1.c

```
int fun(int a){

        return a*10;
}

void main(){
                int x,z;
                read(int,x);
                z=fun(x);
                print(int,z);
}
```

Execution:

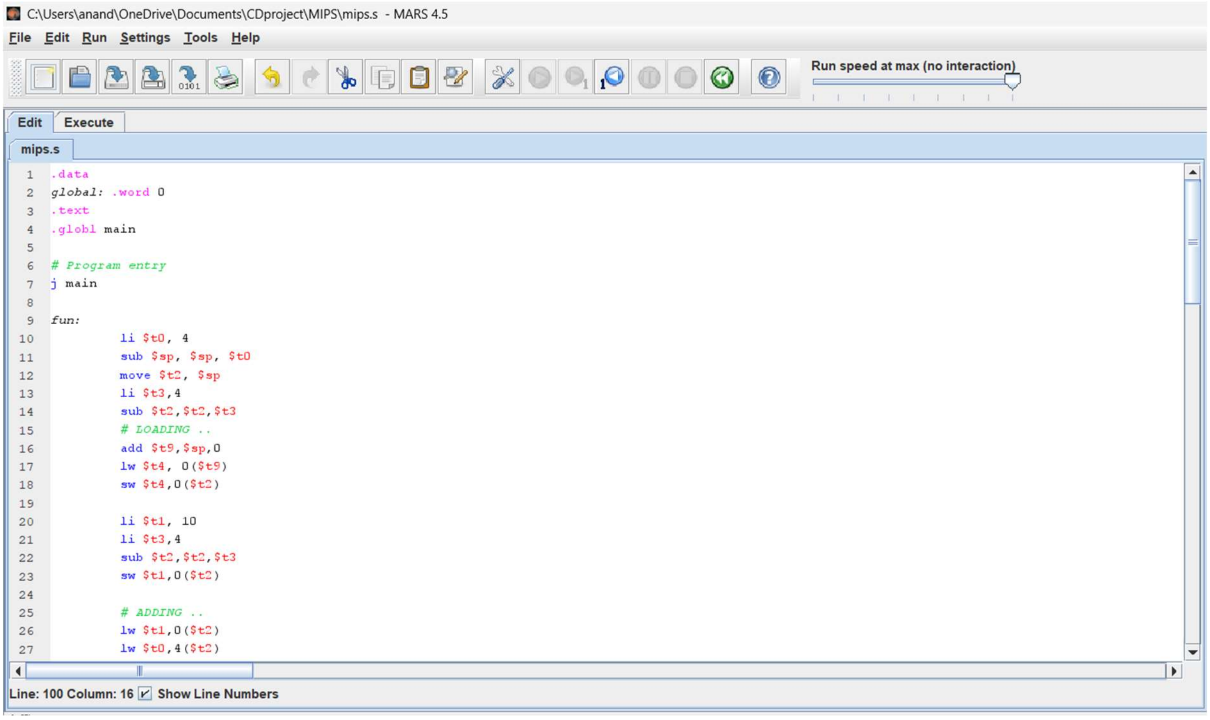
```
[s2022103305@sflinuxonline 14.11.2024-00:42:52 - /project]$ ./start.exe<test
1.c
Print ut
fun
shjfhds
fun ::-- fun

0 0 is size
main 0 0:: fun 0 0::
4 4 is size
a 0 4::
8 8 is size
z 4 4:: x 0 4:: Generating MIPS code...
save ---> fun
save2  20
[s2022103305@sflinuxonline 14.11.2024-00:43:11 - /project]$
```

Output: mips_code.s

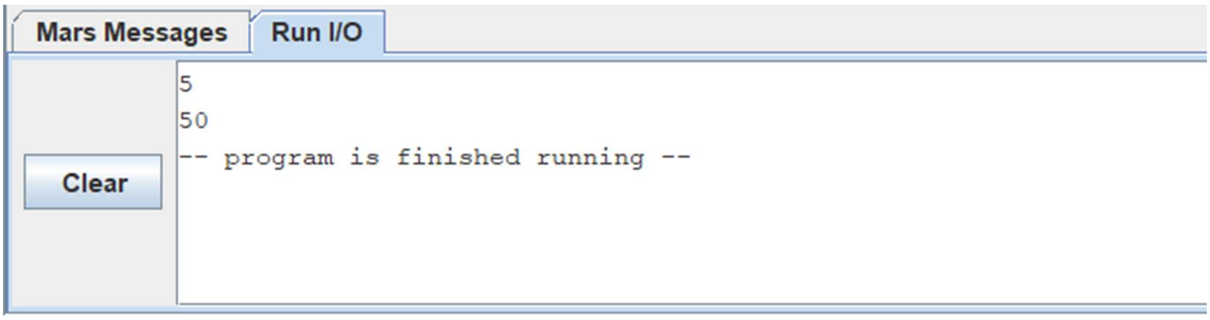
```
.data
global: .word 0
.text
.globl main

# Program entry
j main

fun:
        li $t0, 4
        sub $sp, $sp, $t0
        move $t2, $sp
        li $t3,4
        sub $t2,$t2,$t3
        # LOADING ..
        add $t9,$sp,0
        lw $t4, 0($t9)
        sw $t4,0($t2)

        li $t1, 10
        li $t3,4
        sub $t2,$t2,$t3
        sw $t1,0($t2)

        # ADDING ..
        lw $t1,0($t2)
        lw $t0,4($t2)
        li $t3,4
        add $t2,$t2,$t3
       # *
        mul $t1,$t0,$t1
        sw $t1,0($t2)

       lw $v0,0($t2)
        add $sp, $sp, 4
        move $t2, $sp
        jr $ra
        li $t0, 4
        add $sp, $sp, $t0
        move $t2, $sp
        jr $ra

main:
       la $ra, exit
        li $t0, 8
        sub $sp, $sp, $t0
        move $t2, $sp
        li $v0,5
```

```
         # STORING ..
         add $t9,$sp,0
         sw $v0, 0($t9)
         li $t3,4
         sub $t2,$t2,$t3
         # LOADING ..
         add $t9,$sp,0
         lw $t4, 0($t9)
         sw $t4,0($t2)

        move $t0,$sp
        li $t1, 20
        sub $t0,$t0,$t1
        lw $t4,0($t2)
        sw $t4,0($t0)
        li $t0,4
        sub $sp,$sp,$t0
        sw $ra,0($sp)
        move $t2,$sp
        li $t0,12
        sub $sp,$sp,$t0
        jal fun

        add $sp,$sp,12
        lw $ra,0($sp)
        add $sp,$sp,4
        move $t2,$sp
        li $t0,4
        sub $t2,$t2,$t0
        sw $v0,0($t2)
         lw $t1,0($t2)
         # STORING ..
         add $t9,$sp,4
         sw $t1, 0($t9)
         add $t2,$t2,4
         # LOADING ..
         add $t9,$sp,4
         lw $a0, 0($t9)
         li $v0,1
         syscall
         li $a0,32
         li $v0,11
         syscall
         li $t0, 8
         add $sp, $sp, $t0
         move $t2, $sp
         jr $ra
```

MIPS Execution:



Output :



CONCLUSION

This project successfully demonstrates the process of converting a subset of C code into MIPS assembly code, with a focus on integer functions, variable assignments, and input-output operations. By utilizing a custom-defined C grammar, the project parses and translates simple C programs into MIPS instructions, effectively bridging high-level code and assembly-level operations.

The system is structured into several modules, each contributing to a distinct phase of compilation: clexer.l performs lexical analysis, identifying valid tokens; cparser.y carries out syntax and semantic analysis using Yacc, building an Abstract Syntax Tree (AST) while checking for errors. The code generation phase, implemented in CtoMIPS.c, transforms the AST into executable MIPS instructions, stored in mips_code.s. Additional modules such as symbolTable.c and definition.c handle symbol management and AST manipulation, ensuring efficient data handling and code structure.

By implementing this process, the project provides a foundational framework for understanding compiler construction, the use of Lex and Yacc for parsing, and generating target code for the MIPS architecture. It demonstrates the practical application of theoretical concepts in compiler design and assembly language programming, making it a valuable learning experience for exploring the compilation pipeline from high-level code to assembly.