南開大學
Nankai University

计算机学院

编译系统原理实验报告

定义编译器 & 汇编编程

姓名：刘荟文 孙蕗

学号：2114019 2112060

专业：计算机科学与技术 信息安全

指导教师：王刚老师

2023 年 10 月 9 日

目录

1 摘要

在此次实验中，为了未来成功完成编译器的设计，经小组讨论后确定了编译器支持的 SysY 语言特性，并参考 SysY 中巴克斯瑙尔范式定义，用上下文无关文法描述了 SysY 语言子集。根据所选择的 SysY 语言特性设计了斐波那契数列求解和矩阵相乘程序，包含函数调用、条件分支、循环结构等 SysY 语言特性，自主编写等价的 ARM 汇编程序并进行优化，通过解决遇到的栈帧调整、函数调用等困难对 ARM 汇编有了基本掌握，最后用汇编器生成可执行程序，调试通过并得到正确结果。

关键字：CFG SysY ARM 汇编

2 定义编译器

2.1 支持的 SysY 语言特性

2.1.1 基础 track

- 数据类型：int
- 变量声明，常量声明，常量、变量的初始化
- 语句：赋值语句（=），表达式语句，语句块，if 分支语句，while/for 循环，return
- 表达式：算数运算（+、-、*、/、%，其中 +、-都可以是单目运算符），逻辑运算（&&（与）、||（或）、!（非）），关系运算（==、>、<、>=、<=、!=）
- 注释
- 输入输出（实现连接 SysY 运行时库，参见文档《SysY 运行时库》）

2.1.2 竞赛 track

- 浮点数
  - 浮点数常量识别、变量声明、存储、运算
- 函数、语句块
  - 函数：函数声明、函数调用
  - 变量、常量作用域：在函数、语句块（嵌套）中包含变量、常量声明的处理，break、continue 语句
- 数组：数组（一维、二维、…）的声明和数组元素访问
- 代码优化
  - 寄存器分配优化方法
  - 基于数据流分析的强度削弱、代码外提、公共子表达式删除、无用代码删除等
- 自由发挥

2.2 CFG 描述 SysY 语言特性

2.2.1 Context free grammars

上下文无关文法是一种用于描述程序设计语言语法的表示方式，由四个元素组成：

- 终结符号集合 V_T

  有时也称“词法单元”，是该文法所定义的语言的基本符号的集合。

- 非终结符集合 V_N

  有时也称为“语法变量”。每个非终结符号表示一个终结符号串的集合。

- 产生式集合 P

  每个产生式包括一个称为产生式头或左部的非终结符号，一个箭头，和一个称为产生式体或右部的由终结符号及非终结符号组成的序列。产生式主要用来表示某个语法构造的某种书写形式。如果产生式头非终结符号代表一个语法构造，那么该产生式体就代表了该构造的一种书写方式。

- 开始符号 S

  指定一个非终结符号为开始符。

利用 CFG 对所选 SysY 语言特性子集进行形式化定义。

2.2.2 终结符集合 V_T

终结符是由单引号引起的字符串或者是标识符和数值常量。

- 标识符

  identifier → identifier_nondigit
  |identifier identifier_nondigit
  |identifier digit

  identifier_nondigit → _|a|b|c|d|e|f|g|h|i|j|k|l|m|n
  |o|p|q|r|s|t|u|v|w|x|y|z
  |A|B|C|D|E|F|G|H|I|J|K|L|M|N
  |O|P|Q|R|S|T|U|V|W|X|Y|Z

  digit → 0|1|2|3|4|5|6|7|8|9

  对于同名标识符的规定：

  – 全局变量和局部变量的作用域可以重叠，重叠部分局部变量优先. 在一个函数中定义了和全局变量同名的局部变量时，局部变量会覆盖全局变量，并且只在该函数内部有效。当函数执行完毕后，局部变量就会被销毁，全局变量仍然存在。

  – 同名局部变量的作用域不能重叠；

  – 变量名可以与函数名相同。[1]

- 数值常量

  integer_const 可以表示八进制、十进制、十六进制数字:

  integer_const → decimal_const
  | octal_const
  | hexadecimal_const

  decimal_const → nonzero_digit
  | decimal_const digit

  octal_const → 0 | octal_const octal_digit

  hexadecimal_const → hexadecimal_prefix hexadecimal_digit
  | hexadecimal_const hexadecimal_digit

  hexadecimal_prefix → '0x' | '0X'

  nonzero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  octal_digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

  hexadecimal_digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  | a | b | c | d | e | f | A | B | C | D | E | F

- 关键字

  void, int, const, Ident, if, while, break, continue, return, else, IntConst

- 运算符

  =, +, -, !, *, /, %, <, >, <=, >=, ==, !=, &&, ||

- 基本符号

  ;, [,], {, }, (,), //, /*, */

2.2.3 非终结符集合 V_N

非终结符即一些语法变量，是源程序到终结符之间的过渡。除终结符外其他均为非终结符，非终结符及其含义在下面的 CFG 表达式中。

2.2.4 开始符号 S

开始符号：CompUnit

2.2.5 表达式集合 P

- 编译单元

  CompUnit → CompUnit Decl | CompUnit | Decl | FuncDef | ε

  CompUnit 的顶层变量/常量声明语句、函数定义都不可以重复定义同名标识符，即便标识符的类型不同也不允许。

  CompUnit 的变量/常量/函数声明的作用域从该声明处开始到文件结尾。

- 声明

  Decl → ConstDecl|VarDecl

- 基本类型

  BType → 'int'

- 常量声明

  ConstDecl → 'const' BType ConstDefList ';'
  ConstDefList → ConstDefList, ConstDef | ConstDef

- 常数定义

  ConstDef → Ident Dim '=' ConstInitVal
  Dim → Dim '[' ConstExp ']' | ε

  ConstDef 用于定义常量。在 Ident 后、= 之前是可选的数组维度和各维长度的定义部分，在 = 之后是初始值。

  ConstDef 的数组维度和各维长度的定义部分不存在时，表示定义单个变量。此时 = 右边必须是单个初始数值。

  ConstDef 的数组维度和各维长度的定义部分存在时，表示定义数组。

  当 ConstDef 定义的是数组时，= 右边的 ConstInitVal 表示常量初始化器。全局常量数组的 ConstInitVal 中的 ConstExp 必须是常量表达式。局部常量数组的 ConstInitVal 中的 ConstExp 必须是能在编译时求值的 int 型表达式。[1]

- 常数初值

  ConstInitVal → ConstExp | "ConstValElement"
  ConstValElement → ConstValEnum | ε
  ConstValEnum → ConstValEnum, ConstInitVal | ConstInitVal

  ConstInitVal 初始化器必须是以下三种情况之一：

  – 一对花括号 {}，表示所有元素初始为 0；
  – 数组维数和各维长度完全对应的初始值；
  – 花括号中初始值少于对应维度元素个数，该维其余部分将被隐式初始化为 0。[1]

- 变量声明

  VarDecl → BType VarDefList ';'
  VarDefList → VarDefList, VarDef | VarDef

- 变量定义

  VarDef → Ident Dim | Ident Dim '=' InitVal
  Dim → Dim '['ConstExp']' | ε

  VarDef 用于定义变量。当不含有 = 和初始值时，其运行时实际初值未定义。

  VarDef 的数组维度和各维长度的定义部分不存在时，表示定义单个变量。存在时，和 ConstDef 类似，表示定义数组。[1]

- 变量初值

  InitVal → Exp |"ValElement"
  ValElement → ValEnum | ε
  ValEnum → ValEnum, InitVal | InitVal

- 浮点类型

  float-const → float-dec-const | float-oct-const | float-hex-const
  float-dec-const → int-dec-const node frac-dec-const
  float-oct-const → int-oct-const node frac-oct-const
  float-hex-const → int-hex-const node frac-hex-const
  node → .
  frac-dec-const → digit | frac-dec-const digit
  frac-oct-const → oct-digit | frac-oct-const oct-digit
  frac-hex-const → hex-digit | frac-hex-const hex-digit

- 函数定义

  FuncDef → FuncType Ident '('FuncFParamList')' Block

- 函数类型

  FuncType → 'void' | 'int'

- 函数形参表

  FuncFParamList → FuncFParams | ε
  FuncFParams → FuncFParams, FuncFParam | FuncFParam

- 函数形参

  FuncFParam → BType Ident OpArray
  OpArray → Array | ε
  Array → [ ] | [ ]Arrays;
  Arrays → [Exp]Arrays | [Exp]

- 函数实参表

  FuncRParams → FuncRParams,Exp | Exp

- 语句块

  Block → { OpBlockItems }
  OpBlockItems → BlockItems | ε
  BlockItems → BlockItems BlockItem | BlockItem

  Block 表示语句块。语句块会创建作用域，语句块内声明的变量的生命周期在该语句块内。

  语句块内可以再次定义与语句块外同名的变量或常量，其作用域从定义处开始到该语句块尾结束，它隐藏语句块外的同名变量或常量。[1]

- 语句块项

  BlockItem → Decl | Stmt

- 语句

  Stmt → LVal '=' Exp ';'
  | OpExp ';'
  | Block
  | 'if' (Cond) Stmt OpElse
  | 'while' (Cond) Stmt
  | 'break' ';'
  | 'continue' ';'
  | 'return' OpExp ';'

  OpExp → Exp | ε
  OpElse → 'else' Stmt | ε

  Stmt 中的 if 类型语句遵循就近匹配。

  单个 Exp 可以作为 Stmt。Exp 会被求值，所求的值会被丢弃。[1]

- 表达式

  Exp → AddExp

- 条件表达式

  Cond → LOrExp

- 左值表达式

  LVal → Ident OpArr
  OpArr → Arrays | ε

- 基本表达式

  PrimaryExp → '(' Exp ')' | LVal | Number

- 数值

  Number → IntConst

- 一元表达式

  UnaryExp → PrimaryExp | Ident '(' OpFuncRParams ')'
  | UnaryOp UnaryExp

- 单目运算符

  UnaryOp → '+' | '-' | '!'

- 加减表达式

  AddExp → MulExp
  | AddExp '+' MulExp
  | AddExp '-' MulExp

- 乘除模表达式

  MulExp → UnaryExp

  | MulExp '*' UnaryExp

  | MulExp '/' UnaryExp

  | MulExp '%' UnaryExp

- 关系表达式

  RelExp → AddExp

  | RelExp '<' AddExp

  | RelExp '>' AddExp

  | RelExp '<=' AddExp

  | RelExp '>=' AddExp

- 相等性表达式

  EqExp → RelExp

  | EqExp '==' RelExp

  | EqExp '!=' RelExp

- && 表达式

  LAndExp → EqExp

  | LAndExp '&&' EqExp

- || 表达式

  LOrExp → LAndExp

  | LOrExp'||'LAndExp

- 常量表达式

  ConstExp → AddExp

3 汇编编程

3.1 斐波那契数列

3.1.1 斐波那契 SysY 程序

```
#include <iostream>
using namespace std;
int main()
{
  int a, b, i, t, n;
  a = 0;
  b = 1;
```

```
  i = 1;
  cin >> n;
  cout << a << endl;
  cout << b << endl;
  while (i < n)
  {
    t = b;
    b = a + b;
    cout << b << endl;
    a = t;
    i = i + 1;
  }
  return 0;
}
```

3.1.2 斐波那契 arm 汇编程序

```
.arch armv7-a
.file "fib.cpp"
.text
.align 1
.global main
.syntax unified
.thumb
.thumb_func
.type main, %function
main:
  .fnstart
.LFB1719:
  @ 初始化局部变量
  @ a = 0
  @ b = 1
  @ i = 1
  push {r4, r7, lr} @ 保存寄存器r4、r7和lr到栈中
  .save {r4, r7, lr} @ 保存寄存器r4、r7和lr的状态
  sub sp, sp, #12 @sub指令分配12个字节的栈空间来存储局部变量
  setfp r7, sp, #0 @setfp和add指令设置栈帧指针r7，并将其初始化为栈顶地址。
  add r7, sp, #0 @ 将r7设置为当前栈顶地址

      @ 初始化局部整数变量a、b和i
  mov r4, #0 @ 将0赋值给r4
  mov r0, r4 @ 将r4的值存储到r0，a = 0
  str r0, [r7, #4] @ 将r0的值存储到栈上的偏移为4的位置，保存a的值
  mov r4, #1 @ 将1赋值给r4
  mov r0, r4 @ 将r4的值存储到r0，b = 1
  str r0, [r7, #8] @ 将r0的值存储到栈上的偏移为8的位置，保存b的值
  mov r4, #1 @ 将1赋值给r4
  mov r0, r4 @ 将r4的值存储到r0，i = 1
```

```
32    str r0, [r7, #12] @ 将r0的值存储到栈上的偏移为12的位置, 保存i的值
33
34          @ 调用cin操作, 将输入的值存储到变量n
35    mov r0, r7 @ 将栈帧指针r7的值存储到r0, 准备作为cin的this指针
36    ldr r4, =n @ 将n的地址加载到r4
37    ldr r0, [r4] @ 从n的地址加载值到r0
38    bl  _ZNSirsERi(PLT) @ 调用cin操作
39
40          @ 调用cout操作, 输出a的值
41    mov r4, [r7, #4] @ 从栈上的偏移为4的位置加载a的值到r4
42    ldr r0, =_ZSt4cout @ 将cout的地址加载到r0, 准备作为cout的this指针
43    bl  _ZNSt8ios_base4InitC1Ev(PLT) @ 调用cout的构造函数
44    ldr r0, [r7, #4] @ 从栈上的偏移为4的位置加载a的值到r0
45    ldr r1, =_ZSt4cout @ 将cout的地址加载到r1, 准备作为输出目标
46    bl  _ZNSt13basic_ostreamIcSt11char_traitsIcEE3putEc(PLT) @ 调用cout的put函数
47
48          @ 调用cout操作, 输出b的值
49    ldr r4, [r7, #8] @ 从栈上的偏移为8的位置加载b的值到r4
50    ldr r0, =_ZSt4cout  @ 将cout的地址加载到r0, 准备作为cout的this指针
51    bl  _ZNSt8ios_base4InitC1Ev(PLT) @ 调用cout的构造函数
52    ldr r0, [r7, #8]  @ 从栈上的偏移为8的位置加载b的值到r0
53    ldr r1, =_ZSt4cout  @ 将cout的地址加载到r1, 准备作为输出目标
54    bl  _ZNSt13basic_ostreamIcSt11char_traitsIcEE3putEc(PLT)  @ 调用cout的put函数
55
56    @ 开始while循环
57  .L2:
58  @ 从栈上的偏移为8的位置加载b的值到r0, 将b的值赋给t
59    ldr r0, [r7, #8] @ 从栈上的偏移为8的位置加载b的值到r0
60    str r0, [r7, #16] @ 将b的值保存到栈上的偏移为16的位置, 保存t的值
61
62   @ 计算新的b的值并输出
63    ldr r4, [r7, #8] @ 从栈上的偏移为8的位置加载b的值到r4
64    ldr r0, [r7, #4]  @ 从栈上的偏移为4的位置加载a的值到r0
65    add r0, r0, r4  @ 计算a + b, 并将结果存储到r0
66    str r0, [r7, #8]  @ 将r0的值存储到栈上的偏移为8的位置, 更新b的值
67
68  @ 调用cout操作, 输出b的值
69    ldr r4, [r7, #8]  @ 从栈上的偏移为8的位置加载b的值到r4
70    ldr r0, =_ZSt4cout  @ 将cout的地址加载到r0, 准备作为cout的this指针
71    bl  _ZNSt8ios_base4InitC1Ev(PLT) @ 调用cout的构造函数
72    ldr r0, [r7, #8] @ 从栈上的偏移为8的位置加载b的值到r0
73    ldr r1, =_ZSt4cout @ 将cout的地址加载到r1, 准备作为输出目标
74    bl  _ZNSt13basic_ostreamIcSt11char_traitsIcEE3putEc(PLT) @ 调用cout的put函数
75
76   @ 将t的值赋给a
77    ldr r0, [r7, #16] @ 从栈上的偏移为16的位置加载t的值到r0
78    str r0, [r7, #4]  @ 将r0的值存储到栈上的偏移为4的位置, 更新a的值
79
80   @ 更新循环变量i
```

```
81    ldr r4, [r7, #12]  @ 从栈上的偏移为12的位置加载i的值到r4
82    adds  r4, r4, #1  @ 将i加1
83    str r4, [r7, #12] @ 将r4的值存储到栈上的偏移为12的位置, 更新i的值
84
85   @ 检查是否需要继续循环
86    ldr r0, [r7] @ 从栈上的偏移为0的位置加载n的值到r0
87    ldr r4, [r7, #12] @ 从栈上的偏移为12的位置加载i的值到r4
88    cmp r4, r0  @ 比较i和n的值
89    blt .L2  @ 如果i < n, 则跳转到.L2继续循环
90
91    mov r0, #0  @ 设置返回值为0
92    mov r4, r0  @ 将0赋值给r4, 作为返回值
93    add r7, sp, #12  @ 恢复栈帧
94    mov sp, r7 @ 恢复栈指针
95    @ sp needed
96    pop {r4, r7, pc} @ 弹出保存的寄存器并返回
97  .L7:
98    .align  2
99    .size main, .-main
100 }
```

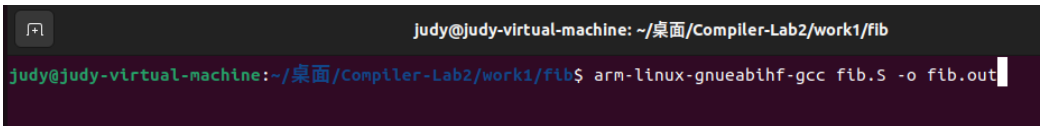
这段 ARM 汇编代码体现了以下 SysY 语言的特性:

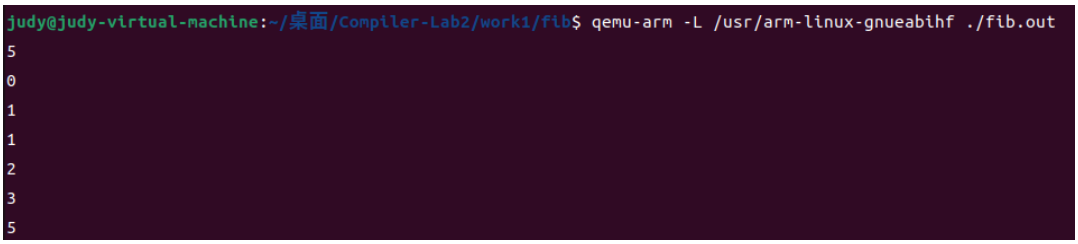
- 变量声明和初始化: 在 SysY 语言中, 变量可以在声明时进行初始化。在这段代码中, a、b、i 等局部变量都被初始化为特定的值。
- 数据类型: SysY 支持整数数据类型, 这在代码中体现为使用整数寄存器 (如 r0、r4) 来存储整数值。
- 输入输出操作: 代码中使用了 cin 和 cout 操作, 这是 SysY 语言中的输入和输出功能的对应。
- 赋值操作: 通过 mov 指令, 代码实现了变量之间的赋值操作。
- 条件判断和循环: 代码中使用了条件分支 (cmp 和 blt 指令), 以及循环结构 (while 循环), 这是 SysY 语言中支持的控制流结构。
- 函数调用: 代码中通过 bl 指令调用了其他函数, 这是 SysY 语言支持的函数调用特性。
- 栈操作: 代码中使用了栈来保存局部变量和寄存器状态, 这是 SysY 语言中常见的内存管理方式。

3.1.3 斐波那契测试

使用下面的命令进行测试

```
linux-gnueabihf-gcc fib.S -o fib.out
qemu-arm -L /usr/arm-linux-gnueabihf ./fib.out
```





3.2 矩阵相乘

3.2.1 矩阵相乘 SysY 程序

```
#include <iostream>

using namespace std;

int main()
{
    const int rows_A = 2;
    const int cols_A = 3;
    const int rows_B = 3;
    const int cols_B = 2;

    int A[rows_A][cols_A] = { {1, 2, 3}, {4, 5, 6} };
    int B[rows_B][cols_B] = { {1, 2}, {3, 4}, {5, 6} };
    int result[rows_A][cols_B];

    // 执行矩阵乘法
    for (int i = 0; i < rows_A; i++)
    {
        for (int j = 0; j < cols_B; j++)
        {
            result[i][j] = 0;
            for (int k = 0; k < cols_A; k++)
            {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    // 打印结果矩阵
    cout << "Matrix A:" << endl;
    for (int i = 0; i < rows_A; i++)
    {
        for (int j = 0; j < cols_A; j++)
```

```
        {
            cout << A[i][j] << " ";
        }
        cout << endl;
    }

    cout << "Matrix B:" << endl;
    for (int i = 0; i < rows_B; i++)
    {
        for (int j = 0; j < cols_B; j++)
        {
            cout << B[i][j] << " ";
        }
        cout << endl;
    }

    cout << "Result of matrix multiplication:" << endl;
    for (int i = 0; i < rows_A; i++)
    {
        for (int j = 0; j < cols_B; j++)
        {
            cout << result[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

3.2.2 矩阵相乘 arm 汇编程序

```
.arch armv7-a
.file "array_multi.cpp"
.text
.section .rodata
.align 2
.LC2:
    .ascii "Matrix A:\000" @ 字符串常量 "Matrix A:"
    .align 2
.LC3:
    .ascii " \000"
    .align 2
.LC4:
    .ascii "Matrix B:\000" @ 字符串常量 "Matrix B:"
    .align 2
.LC5:
    .ascii "Result of matrix multiplication:\000" @ 字符串常量 "Result of matrix
        multiplication:"
```

```
17      .align 2
18  .LC0:
19      .word 1
20      .word 2
21      .word 3
22      .word 4
23      .word 5
24      .word 6
25  .align 2
26  .LC1:
27      .word 1
28      .word 2
29      .word 3
30      .word 4
31      .word 5
32      .word 6
33  .text
34  .align 1
35  .global main
36  .syntax unified
37  .thumb
38  .thumb_func
39  .type main, %function
40  main:
41      .fnstart
42
43  .LFB1719:
44      @ args = 0, pretend = 0, frame = 120
45      @ frame_needed = 1, uses_anonymous_args = 0
46      push {r4, r5, r6, r7, lr} @ 保存寄存器r4, r5, r6, r7, lr到栈中
47      .save {r4, r5, r6, r7, lr} @ 保存寄存器 r4, r5, r6, r7, lr的状态
48      .pad #124 @ 分配栈帧, 大小为 124 字节
49      sub sp, sp, #124  @ 调整栈指针, 分配栈空间
50      .setfp r7, sp, #0  @ 设置帧指针 r7
51      add r7, sp, #0  @ r7 = sp
52      ldr r4, .L22 @ 将 .L22 地址加载到 r4 寄存器
53
54  .LPIC3:
55      add r4, pc @ 计算 .L22 地址偏移量
56      ldr r2, .L22+4 @ 将 .L22+4 地址加载到 r2 寄存器
57
58  .LPIC9:
59      add r2, pc @ 计算 .L22+4 地址偏移量
60      ldr r3, .L22+8  @ 将 .L22+8 地址加载到 r3 寄存器
61      ldr r3, [r2, r3] @ 从地址 [r2 + r3] 加载数据到 r3
62      ldr r3, [r3] @ 从地址 [r3] 加载数据到 r3
63      str r3, [r7, #116]  @ 存储 r3 到地址 [r7 + 116]
64      mov r3, #0  @ 将 0 存储到 r3 寄存器
65      movs r3, #2 @ 将 2 存储到 r3 寄存器
```

```
66      str r3, [r7, #36] @ 存储 r3 到地址 [r7 + 36]
67      movs r3, #3 @ 将 3 存储到 r3 寄存器
68      str r3, [r7, #40] @ 存储 r3 到地址 [r7 + 40]
69      movs r3, #3 @ 将 3 存储到 r3 寄存器
70      str r3, [r7, #44] @ 存储 r3 到地址 [r7 + 44]
71      movs r3, #2 @ 将 2 存储到 r3 寄存器
72      str r3, [r7, #48] @ 存储 r3 到地址 [r7 + 48]
73      ldr r3, .L22+12 @ 将 .L22+12 地址加载到 r3 寄存器
74
75  .LPIC0:
76      add r3, pc @ 计算 .L22+12 地址偏移量
77      add r5, r7, #68 @ r5 = r7 + 68, 用于存储局部变量
78      mov r6, r3 @ r5 = r7 + 68, 用于存储局部变量
79      ldmia r6!, {r0, r1, r2, r3} @ r5 = r7 + 68, 用于存储局部变量
80      stmia r5!, {r0, r1, r2, r3} @ 将加载的数据存储到 r5 地址
81      ldm r6, {r0, r1} @ 从 r6 地址加载 r0 和 r1 寄存器的数据
82      stm r5, {r0, r1} @ 将加载的数据存储到 r5 地址
83      ldr r3, .L22+16 @ 将 .L22+16 地址加载到 r3 寄存器
84
85  .LPIC1:
86      add r3, pc @ 计算 .L22+16 地址偏移量
87      add r5, r7, #92 @ r5 = r7 + 92, 用于存储局部变量
88      mov r6, r3 @ 将 r3 存储到 r6 寄存器
89      ldmia r6!, {r0, r1, r2, r3} @ 从 r6 地址加载多个寄存器的数据
90      stmia r5!, {r0, r1, r2, r3} @ 将加载的数据存储到 r5 地址
91      ldm r6, {r0, r1} @ 从 r6 地址加载 r0 和 r1 寄存器的数据
92      stm r5, {r0, r1} @ 将加载的数据存储到 r5 地址
93      b .L2 @ 跳转到标签 .L2
94
95  .L7:
96      b .L3 @ 跳转到标签 .L3
97
98  .L6:
99      ldr r3, [r7] @ 从地址 [r7] 加载数据到 r3
100     lsls r2, r3, #1 @ 左移 r3 寄存器的值, 结果存储到 r2
101     ldr r3, [r7, #4] @ 从地址 [r7 + 4] 加载数据到 r3
102     add r3, r3, r2 @ r3 = r3 + r2
103     lsls r3, r3, #2 @ 左移 r3 寄存器的值, 结果存储到 r3
104     adds r3, r3, #120 @ r3 = r3 + 120
105     add r3, r3, r7 @ r3 = r3 + r7
106     ldr r3, [r3, #-52] @ 从地址 [r3 - 52] 加载数据到 r3
107     mov r1, r3 @ 将 r3 的值存储到 r1
108     ldr r3, .L22+24 @ 将 .L22+24 地址加载到 r3 寄存器
109     ldr r3, [r4, r3] @ 从地址 [r4 + r3] 加载数据到 r3 寄存器
110     mov r0, r3 @ 将 r3 寄存器的值存储到 r0 寄存器
111     bl _ZNSolsEi(PLT) @ 调用函数 _ZNSolsEi(PLT) 处理输出
112     mov r2, r0 @ 将返回值存储到 r2 寄存器
113     ldr r3, .L22+32 @ 将 .L22+32 地址加载到 r3 寄存器
114
```

```
115 .LPIC4:
116     add r3, pc  @ 计算 .L22+32 地址偏移量
117     mov r1, r3 @ 将 r3 寄存器的值存储到 r1 寄存器
118     mov r0, r2  @ 将 r2 寄存器的值存储到 r0 寄存器
119     bl _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(PLT)  @
            调用函数输出字符串
120     ldr r3, [r7, #16]  @ 从地址 [r7 + 16] 加载数据到 r3 寄存器
121     adds r3, r3, #1 @ r3 = r3 + 1
122     str r3, [r7, #16]  @ 存储 r3 到地址 [r7 + 16]
123
124 .L9:
125     ldr r3, [r7, #16] @ 从地址 [r7 + 16] 加载数据到 r3 寄存器
126     cmp r3, #2 @ 比较 r3 和 2
127     ble .L10  @ 如果 r3 小于等于 2, 则跳转到 .L10
128     b .L2  @ 否则跳转到 .L2
129
130 .L5:
131     b .L3  @ 跳转到标签 .L3
132
133 .L4:
134     b .L11 @ 跳转到标签 .L11
135
136 .L3:
137     ldr r3, [r7, #4]  @ 从地址 [r7 + 4] 加载数据到 r3 寄存器
138     cmp r3, #1 @ 比较 r3 和 1
139     ble .L6  @ 如果 r3 小于等于 1, 则跳转到 .L6
140     ldr r3, [r7] @ 从地址 [r7] 加载数据到 r3 寄存器
141     adds r3, r3, #1  @ r3 = r3 + 1
142     str r3, [r7]  @ 存储 r3 到地址 [r7]
143
144 .L2:
145     ldr r3, [r7]  @ 从地址 [r7] 加载数据到 r3 寄存器
146     cmp r3, #1  @ 比较 r3 和 1
147     ble .L7 @ 如果 r3 小于等于 1, 则跳转到 .L7
148     ldr r3, .L22+20  @ 将 .L22+20 地址加载到 r3 寄存器
149
150 .LPIC2:
151     add r3, pc  @ 计算 .L22+20 地址偏移量
152     mov r1, r3 @ 将 r3 寄存器的值存储到 r1 寄存器
153     ldr r3, .L22+24  @ 将 .L22+24 地址加载到 r3 寄存器
154     ldr r3, [r4, r3]  @ 从地址 [r4 + r3] 加载数据到 r3 寄存器
155     mov r0, r3 @ 将 r3 寄存器的值存储到 r0 寄存器
156     bl _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(PLT)  @
            调用函数输出字符串
157     mov r2, r0 @ 将返回值存储到 r2 寄存器
158     ldr r3, .L22+28 @ 将 .L22+28 地址加载到 r3 寄存器
159     ldr r3, [r4, r3]  @ 从地址 [r4 + r3] 加载数据到 r3 寄存器
160     mov r1, r3 @ 将 r3 寄存器的值存储到 r1 寄存器
161     mov r0, r2 @ 将 r2 寄存器的值存储到 r0 寄存器
```

```
162     bl _ZNSolsEPFRSoS_E(PLT) @ 调用函数输出字符
163     movs r3, #0 @ 将 0 存储到 r3 寄存器
164     str r3, [r7, #12] @ 存储 r3 到地址 [r7 + 12]
165
166 .L8:
167     b .L11 @ 跳转到标签 .L11
168
169 .L10:
170     b .L5 @ 跳转到标签 .L5
171
172 .L1:
173     str r0, [sp, #116] @ 存储 r0 到地址 [sp + 116]
174     ldr r3, .L22+56 @ 将 .L22+56 地址加载到 r3 寄存器
175
176 .LPIC11:
177     add r3, pc @ 计算 .L22+56 地址偏移量
178     add r4, r4, r3 @ r4 = r4 + r3
179     add r4, r7, r4 @ r4 = r4 + r7
180     ldr r0, [sp, #116] @ 从地址 [sp + 116] 加载数据到 r0 寄存器
181     add sp, sp, #124 @ 回收栈空间
182     @ sp needed
183     .fnend
184 }
```

这段 ARM 汇编代码体现了以下 SysY 语言的特性：

- 循环结构：在代码中存在循环结构，例如.L6, .L7, .L2 等标签，这表示编译器能够识别和生成循环结构。
- 条件分支：存在条件分支指令，例如 cmp 和 ble，这些用于控制程序流程，根据条件执行不同的代码块。
- 函数调用：可以看到 bl 指令，用于调用其他函数，例如 _ZNSolsEi 和 _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc，这表示 SysY 编译器可以处理函数调用。
- 全局变量和字符串：在代码中存在对全局变量（例如.LC2, .LC4, .LC5）和字符串的处理，这说明编译器可以处理全局变量和字符串字面值。

3.2.3 矩阵相乘测试

使用下面的命令进行测试

```
linux-gnueabihf-gcc array_multi.S -o array_multi.out
qemu-arm -L /usr/arm-linux-gnueabihf ./array_multi.out
```

```
judy@judy-virtual-machine: ~/桌面/Compiler-Lab2/work1/array_multi
judy@judy-virtual-machine:~/桌面/Compiler-Lab2/work1/array_multi$ linux-gnueabihf-gcc array_multi.S -o array_multi.out
```

```
judy@judy-virtual-machine: ~/桌面/Compiler-Lab2/work1/array_multi
judy@judy-virtual-machine:~/桌面/Compiler-Lab2/work1/array_multi$ qemu-arm -L /usr/arm-linux-gnueabihf ./array_multi.out
Matrix A:
1 2 3
4 5 6
Matrix B:
1 2
3 4
5 6
Result of matrix multiplication:
22 28
49 64
```

4 思考

4.1 编译器转换 SysY 程序

4.1.1 Question

如果不是人“手工编译”, 而是要实现一个计算机程序（编译器) 来将 SysY 程序转换为汇编程序, 应该如何做 (这个编译器程序的数据结构和算法设计)?

4.1.2 Answer

一个完整的编译器的设计涉及词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成等多个阶段，并利用适当的数据结构和算法来实现这些功能。

- 词法分析（Lexical Analysis）
  - 数据结构：使用有限自动机（Finite Automaton）或正则表达式定义词法单元的模式。
  - 算法设计：根据模式匹配算法（如确定有穷自动机或正则表达式匹配算法），将源代码分解为词法单元。
- 语法分析（Syntax Analysis）
  - 数据结构：可以使用语法文法规则定义语法树的节点和生成规则。
  - 算法设计：使用递归下降（Recursive Descent）或者 LR 分析技术（如 LALR、LR(1) 等）来构建语法树。可以使用栈来管理分析过程中的状态和符号。
- 语义分析（Semantic Analysis）
  - 数据结构：构建符号表来管理变量、函数等的信息。
  - 算法设计：对语法树进行静态语义检查，包括类型检查、重定义检查、作用域检查等。可以根据语义规则进行相应的处理。

- 中间代码生成（Intermediate Code Generation）
  - 数据结构：使用合适的中间表示形式，如三地址码、四元式等。
  - 算法设计：基于语法树和符号表，将 SysY 程序转换为中间代码形式。可以考虑进行一些简单的优化，如常量折叠、公共子表达式消除等。
- 目标代码生成（Code Generation）
  - 数据结构：使用汇编代码生成的相关数据结构。
  - 算法设计：根据中间代码生成相应的汇编代码。根据目标平台的体系结构和指令集，生成对应的汇编代码。

  在上述过程中，合适的数据结构与算法设计是关键。
- 词法分析可以使用有限自动机或正则表达式匹配算法进行模式匹配。
- 语法分析可以使用递归下降或 LR 分析方法进行语法树的构建。
- 语义分析需要构建符号表来管理语义信息，并根据规则进行相应检查。
- 中间代码生成阶段可以使用合适的中间表示形式，以便后续优化与生成目标代码

5 总结

5.1 任务分工

本次实验小组组员均全程参与，阅读学习提供的参考资料，加之独立思考与合作讨论，共同完成了 CFG 对 SysY 语言子集的描述。两人均设计了斐波那契与矩阵相乘程序，并编写等价的 arm 汇编程序，将两人所得结果合并后进行改进优化，最终呈现效果如实验报告和源码链接所示。思考题也由两人共同讨论整理得出。经过本次实验，我们对编译器有了更深入的了解，掌握了编译器相关的重要知识，对未来在编译原理课程方面进一步的学习奠定下坚实的基础。

5.2 源码链接

https://github.com/jtysxtm/Compiler_Lab2

参考文献

[1] https://buaa-se-compiling.github.io/miniSysY-tutorial/miniSysY.html