



南開大學
Nankai University

南 开 大 学

计 算 机 学 院

编译原理作业报告

定义你的编译器 & 汇编程序

朱浩泽 1911530

年级：2019 级

专业：计算机科学与技术

指导教师：王刚

2021 年 10 月 11 日

摘要

本人未进行分组，所有任务均为独立完成。此次实验主要内容为对 SysY 语言特性进行分析和形式化定义并设计了上下文无关文法，以及对所设计的语言进行向 arm 汇编程序的手写转化。

关键字：SysY 语言、形式化定义、arm 汇编

目录

一、 SysY 语言特性和形式化定义	1
(一) SysY 语言特性	1
1. 关键字	1
2. 变量	1
3. 常量	2
4. 运算符和表达式	2
5. 语句	4
6. 函数	4
(二) 形式化定义	5
1. 变量声明	6
2. 常量声明	6
3. 表达式	6
4. 赋值表达式	6
5. 逻辑表达式	6
6. 关系表达式	7
7. 算数表达式	7
8. 系统操作	7
9. 语句	7
10. 循环语句	7
11. 分支语句	8
12. 跳转语句	8
13. 函数	8
14. 注释	8
15. 输入输出流	8
二、 汇编编程	9
(一) 一些基本操作	9
1. 关于函数的栈指针操作	9
2. I/O 操作	9
(二) 汇编程序编写	10
1. 斐波那契数列	10
2. 求平方	12

一、SysY 语言特性和形式化定义

(一) SysY 语言特性

Sysy 语言是 C 语言的一个子集，因此继承了 C 语言的语法定义和特性。由函数、常变量声明、语句、表达式等多种元素构成。接下来，我们将对这种语言的每个部分的特性进行具体分析。

1. 关键字

C++ 常用关键字如表1所示，每一个关键字在上下文无关文法中都会看作一个终结符，即语法树的叶结点。本实验将选取其中的一部分作为子集，构造 SysY 语言。

类型	关键字
数据类型相关	<i>int, bool, true, false, char, wchar_t, int, double, float, short, long, signed, unsigned</i>
控制语句相关	<i>switch, case, default, do, for, while, if, else, break, continue, goto</i>
定义、初始化相关	<i>const, volatile, enum, export, extern, public, protected, private, template, static, struct, class, union, mutable, virtual</i>
系统操作相关	<i>catch, throw, try, new, delete, friend, inline, operator, reinterpret_cast, typename</i>
命名相关	<i>using, namespace, typeof</i>
函数和返回值相关	<i>void, return, sizeof, typedef</i>
其他	<i>this, asm, _cast</i>

表 1: C++ 关键字

其中，由于功能相对简单，故为了满足最基本要求，我们为 SysY 语言选取的保留字如表2所示：

类型	关键字
数据类型相关	<i>int, char</i>
控制语句相关	<i>while, if, else, break, continue</i>
定义、初始化相关	<i>const</i>
系统操作相关	<i>new, delete</i>
命名相关	<i>using, namespace</i>
函数和返回值相关	<i>void, return</i>

表 2: SysY 语言关键字

2. 变量

C 语言中规定，将一些程序运行中可变的值称之为变量，与常量相对。在程序运行期间，随时可能产生一些临时数据，应用程序会将这些数据保存在一些内存单元中，每个内存单元都用一个标识符来标识。这些内存单元我们称之为变量，定义的标识符就是变量名，内存单元中存储的数据就是变量的值。^[1] 变量可以作左值，常量则只能作为右值。变量除了与常量相同的整型类

型、实型类型、字符类型这三个基本类型之外，还有构造类型、指针类型、空类型。由于在我们定义的 SysY 语言中，只会使用整型变量和字符变量两种变量类型，故其余几种类型不在此描述。

三种基本类型

整型变量 整型常量为整数类型 *int* 的数据。可分别如下表示为八进制、十进制、十六进制

- 十进制整型变量：0, 123, -1
- 八进制整型变量：0123, -01
- 十六进制整型变量：0x123, -0x88

实型变量 实型变量是实际中的小数，又称为浮点型变量。按照精度可以分为单精度浮点数 (float) 和双精度浮点数 (double)。浮点数的表示有三种方式如下：

- 指明精度的表示：以 f 结尾为单精度浮点数，如：2.3f；以 d 结尾为双精度浮点数，如：3.6d
- 不加任何后缀的表示：11.1, 5.5
- 指数形式的表示：5.022e+23f, 0f

字符变量 *char* 用于表示一个字符，表示形式为 '需要表示的字符常量'。其中，所表示的内容可以是英文字母、数字、标点符号以及由转义序列来表示的特殊字符。如 'a' '3' ',' '\n'。

变量的定义 用于为变量分配存储空间，还可为变量指定初始值。程序中，变量有且仅有一个定义。^[2] 变量有三个基本要素：变量名，代表变量的符号；变量的数据类型，每一个变量都应具有一种数据类型且内存中占据一定的储存空间；变量的值，变量对应的存储空间中所存放的内容。变量的定义可以以如下的形式：

```
1 type variable_list
```

在我们定义的 SysY 语言中，将支持整型变量（十进制）、字符型变量以及行主存储的整型一维数组类型。

3. 常量

C 语言中规定，将一些不可变的值称之为常量。常量可以分为整型常量、实型常量、字符常量这三种常量，其形式与整型变量、实型变量、字符变量基本相同，只不过在声明时必须初始化且在程序中不可以改变其值。在我们定义的 SysY 语言中，将定义整型常量（十进制）和字符型常量。

4. 运算符和表达式

在 C 语言中，运算符分为算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符、杂项运算符六大类。其中，我们所设计的 SysY 语言将定义以下运算符：

表达式 由运算分量和运算符按一定规则组成。运算分量是运算符操作的对象，通常是各种类型的数据。运算符指明表达式的类型；表达式的运算结果是一个值——表达式的值。出现在赋值运算符左边的分量为左值，代表着一个可以存放数据的存储空间；左值只能是变量，不能是常量或表达式，因为只有变量才可以带表存放数据的存储空间。出现在赋值运算符右边的分量为右值，右值没有特殊要求。

运算符	描述
+	把两个操作数相加
-	从第一个操作数中减去第二个操作数
*	把两个操作数相乘
/	分子除以分母
%	取模运算符，整除后的余数

表 3: 算术运算符

运算符	描述
==	检查两个操作数的值是否相等，如果相等则条件为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。

表 4: 关系运算符

运算符	描述
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。
	称为逻辑或运算符。如果两个操作数中有任何一个非零，则条件为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。

表 5: 逻辑运算符

运算符	描述
=	简单的赋值运算符，把右边操作数的值赋给左边操作数。

表 6: 赋值运算符

运算符	描述
&	返回变量的地址。
*	指向一个变量。

表 7: 复杂运算符

运算符优先级 运算符中优先级确定了表达式中项的组合，这会极大地影响表达式的计算过程以及结果。运算的优先顺序为：括号优先运算 → 优先级高的运算符优先运算 → 优先级相同的运算参照运算符结合性依次进行。当表达式包含多个同级运算符时，运算的先后次序分为左结合规则和右结合规则。其中左结合规则是从左向右依次计算，包括的运算符有双目的算术运算符、关系运算符、逻辑运算符、位运算符、逗号运算符；右结合规则是从右向左依次计算，包括的运算符有可以连续运算的单目运算符、赋值运算符、条件运算符。运算符优先级由高到低排列：后缀 → 一元 → 乘除 → 加减 → 移位 → 关系 → 相等 → 位与 → 位异或 → 位或 → 逻辑与 → 逻辑或 → 条件 → 赋值 → 逗号

5. 语句

在 C 语言中，语句分为说明语句、表达式语句、控制语句、标签语句、复合语句和块语句。在我们所定义的 SysY 语言中，我们将定义表达式语句和控制语句，其中控制语句分为分支语句、循环语句和转向语句。

表达式语句 任意有效表达式都可以作为表达式语句，其形式为表达式后面加上“;”。

分支语句 if 语句和 if……else 语句，由关键字 if 和 else 组成。其基本形式如下

```
1 if(expr){  
2     stmts  
3 }  
4 else{  
5     stmts  
6 }
```

循环语句 又称重复语句，用于重复执行某些语句。本实验的循环语句由 while 语句实现，基本形式如下

```
1 while{  
2     stmts  
3 }
```

转向语句 用于从循环体跳出的 break 语句；用于立即结束本次循环而去继续下一次循环的 continue 语句；用于立即从某个函数中返回到调用该函数位置的 return 语句。

6. 函数

函数的定义 函数是一组一起执行一个任务的语句。程序中功能相同，结构相似的代码段可以用函数进行描述。函数的功能相对独立，用来解决某个问题，具有明显的入口和出口。函数也可以称为方法、子例程或程序等等。

函数说明 C 语言中，函数必须先说明后调用。函数的说明方式有两种，一种是函数原型，相当于“说明语句”，必须出现在调用函数之前；一种是函数定义，相当于“说明语句 + 初始化”，可以出现在程序的任何合适的地方。在函数声明中，参数的名称并不重要，只有参数的类型是必需的。函数的声明形式如下所示：

```
1 return_type function_name(parameter list);
```

形式化定义 C 语言中，函数的形式化定义如下所示：

```

1 return_type function_name(parameter list)
2 {
3     body of the function
4 }
```

本实验定义的 SysY 语言的函数定义完全与此相同。

函数的参数 函数可以分为有参函数和无参函数。如果函数要使用参数，则必须声明接受参数值的变量，这些变量称为函数的形式参数。形式参数和函数中的局部变量一样，在函数创建时被赋予地址，在函数退出是被销毁。函数参数的调用分为传值调用和引用调用两种，传值调用是将实际的变量的值复制给形式参数，形式参数在函数体中的改变不会影响实际变量；引用调用是将形式参数作为指针调用指向实际变量的地址，当对在函数体中对形式参数的指向操作时，就相当于对实际参数本身进行的操作。

除此之外，函数还可以分为内联函数、外部函数等等，并还可以进行重载等操作。本次实验的 SysY 语言，只对函数最基本的功能进行实现。

（二）形式化定义

接下来，我们将采用 CFG 即上下文无关文法对 SysY 语言进行形式化定义。上下文无关文法由一个终结符号集合 V_T 、一个非终结符号集合 V_N 、一个产生式集合 \mathcal{P} 和一个开始符号 S 四个元素组成。在接下来的定义中，数位、符号和斜体字符串将被看作终结符号，斜体字符串将被看作非终结符号。若多个产生式以一个非终结符号为头部，则这些产生式的右部可以放在一起，并用 | 分割。

名称	符号	名称	符号
声明语句	<i>decl</i>	标识符	<i>id</i>
标识符列表	<i>idlist</i>	数据类型	<i>type</i>
表达式	<i>expr</i>	一元表达式	<i>unary_expr</i>
赋值表达式	<i>assign_expr</i>	逻辑表达式	<i>logical_expr</i>
算数表达式	<i>math_expr</i>	关系表达式	<i>relation_expr</i>
数字	<i>digit</i>	整数	<i>decimal</i>
符号和字母	<i>character</i>	常量定义	<i>const_init</i>
分配内存	<i>allocate</i>	回收内存	<i>recovery</i>
语句	<i>stmt</i>	循环语句	<i>loop_stmt</i>
分支语句	<i>selection_stmt</i>	跳转语句	<i>jmp_stmt</i>
函数定义	<i>funcdef</i>	函数参数	<i>para</i>
函数参数列表	<i>paralist</i>	函数名称	<i>funcname</i>
函数返回值	<i>re_type</i>		

表 8: 下文各符号含义

1. 变量声明

变量可以声明分为仅声明变量和声明变量且赋初值。(整型变量只支持十进制) 数组由指针实现。

$$\begin{aligned} idlist &\rightarrow idlist, id \mid id \\ type &\rightarrow int \mid char \\ decl &\rightarrow type \ idlist \mid \\ &\quad type \ id = logical_expr \mid \\ &\quad type \ id = unary_expr \mid \\ &\quad type \ *id = \&id \end{aligned}$$

2. 常量声明

常量包括整型常量（十进制）和字符型常量。

$$const_init \rightarrow \text{const } type \ id = unary_expr$$

3. 表达式

表达式可以分为一元表达式、赋值表达式、逻辑表达式、算数表达式、关系表达式。

$$expr \rightarrow unary_expr \mid assign_expr \mid logical_expr \mid math_expr \mid relation_expr$$

4. 赋值表达式

赋值表达式不能对常量进行赋值。

$$\begin{aligned} digit &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ decimal &\rightarrow digit \mid decimal \ digit \\ character &\rightarrow _ \mid a - z \mid A - Z \\ unary_expr &\rightarrow decimal \mid 'character' \mid id \\ assign_expr &\rightarrow id = unary_expr \mid \\ &\quad id = logical_expr \mid \\ &\quad id = funcnmae(paralist) \end{aligned}$$

5. 逻辑表达式

逻辑表达式包括逻辑与、逻辑或、逻辑非运算。

$$\begin{aligned} logical_expr &\rightarrow unary_expr \mid \\ &\quad !(logical_expr) \mid \\ &\quad logical_expr || logical_expr \mid \\ &\quad logical_expr \&\& logical_expr \end{aligned}$$

6. 关系表达式

关系表达式包括判断两个值是否相等或比较两值的大小。

$$\begin{aligned} \text{relation_expr} \rightarrow & \text{unary_expr} == \text{unary_expr} \mid \\ & \text{unary_expr} != \text{unary_expr} \mid \\ & \text{unary_expr} > \text{unary_expr} \mid \\ & \text{unary_expr} < \text{unary_expr} \mid \\ & \text{unary_expr} >= \text{unary_expr} \mid \\ & \text{unary_expr} <= \text{unary_expr} \end{aligned}$$

7. 算数表达式

算数表达式包括加、减、乘、除、取模、取负六种运算。

$$\begin{aligned} \text{math_expr} \rightarrow & \text{unary_expr} \mid \\ & - \text{unary_expr} \mid \\ & \text{math_expr} + \text{math_expr} \mid \\ & \text{math_expr} - \text{math_expr} \mid \\ & \text{math_expr} * \text{math_expr} \mid \\ & \text{math_expr} / \text{math_expr} \mid \\ & \text{math_expr} \% \text{math_expr} \end{aligned}$$

8. 系统操作

系统操作包括分配内存和回收内存。

$$\begin{aligned} \text{allocate} \rightarrow & \text{type} * \text{id} = \text{new type} [\text{id}] \mid \\ & \text{type} * \text{id} = \text{new type} [\text{decimal}] \end{aligned}$$

9. 语句

语句包括循环语句、分支语句、跳转语句。

$$\begin{aligned} \text{stmt} \rightarrow & \text{loop_stmt} \mid \text{selecion_stmt} \mid \text{jmp_stmt} \mid \\ & \text{expr}; \text{stmt} \mid \\ & \text{expr} \end{aligned}$$

10. 循环语句

循环语句利用 while 实现。

$$\text{loop_stmt} \rightarrow \text{while}(\text{expr}) \{ \text{stmt} \}$$

11. 分支语句

分支语句利用 if else 实现。

$$\begin{aligned} selection_stmt \rightarrow & \text{if}(\text{expr}) \{ \text{stmt} \} \mid \\ & \text{if}(\text{expr}) \{ \text{stmt} \} \text{ else } \{ \text{stmt} \} \end{aligned}$$

12. 跳转语句

跳转语句包括继续执行循环、退出循环和返回值。

$$\begin{aligned} jmp_stmt \rightarrow & \text{continue} \mid \\ & \text{break} \mid \\ & \text{return} \mid \\ & \text{return expr} \end{aligned}$$

13. 函数

函数的返回值有整型、字符型、指针型三种，参数有整型、字符型、指针型、引用四种。（数组由指针实现）

$$\begin{aligned} funcdef \rightarrow & re_type \text{ funcname}(\text{paralist}) \text{ stmt} \\ paralist \rightarrow & para, paralist \mid para \\ para \rightarrow & type \text{ id} \mid \\ & type * \text{ id} \mid \\ & type \& \text{ id} \\ re_type \rightarrow & type \mid type * \mid \text{void} \end{aligned}$$

14. 注释

本实验定义的 SysY 语言将使用 // 作为注释开始的符号，/* */ 将不被使用。在编译的预处理阶段，// 后到行末的内容将被删除。

15. 输入输出流

输入输出流将由 SysY 语言提供的 I/O 函数实现，在此不再定义。

二、 汇编编程

本人在 Linux ubuntu 4.15.0-142-generic 环境下，利用 vim 将 C++ 文件手动翻译成汇编文件。通过 `arm-linux-gnueabi-g++ 汇编文件名.S -o 目标文件名 -static` 指令生成可执行程序，再通过 `qemu-arm ./目标文件名` 指令进行运行调试，最终对 C++ 文件到 arm 源文件的转换有了一定的了解。手动编写的汇编文件已经上传于 gitlab，链接如下：

SSH: [git@gitlab.eduxiji.net:nku2021-anthony/compilers.git](https://gitlab.eduxiji.net:nku2021-anthony/compilers.git)

HTTP: <https://gitlab.eduxiji.net/nku2021-anthony/compilers.git>

(一) 一些基本操作

1. 关于函数的栈指针操作

在现今的计算机体系结构中，栈是向下增长的，即由大端地址增长到小端地址。在 arm 汇编中，fp 寄存器用作帧指针，sp 寄存器指向栈顶，在跳转语句调用子函数时会将当前的 PC 保存在 lr 寄存器中。当调用一个函数时，该函数首先将 fp 的当前值保存在堆栈上。然后，它将 sp 寄存器的值保存在 fp 寄存器中。然后递减 sp 寄存器来为本地变量分配空间。fp 寄存器用于访问本地变量和参数，局部变量位于帧指针的负偏移量处，传递给函数的参数位于帧指针的正偏移量。当函数返回时，fp 寄存器被复制到 sp 寄存器中，这将释放用于局部变量的堆栈，函数调用者的 fp 寄存器的值由 pop 从堆栈中恢复。[3]

1	<code>push {fp, lr} @ fp</code>	sp	fp	lr	pc	lr
2	<code>@</code>					
3	<code>pop {fp, lr}</code>					
4	<code>bx lr @ mov pc, lr</code>					

函数被调用时的基本框架

2. I/O 操作

已知在 arm 汇编架构中，关于 C++ 文件输入输出流 cin、cout 没有直接相对应的指令，所以我们会调用从 C 语言中继承的 scanf 和 printf 在其中使用。已知，在调用这两个函数时，r0 寄存器保存的是自定义字符串的地址，r1 寄存器保存的是“%d”中将要被替换的内容，具体代码如下：

1	<code>sub sp, sp, #4 @</code>	4
2	<code>ldr r0, =_cin</code>	
3	<code>mov r1, sp @ sp</code>	r1 scanf
4	<code>bl scanf</code>	
5	<code>ldr , [sp, #0] @ sp</code>	
6	<code>add sp, sp, #4 @</code>	
7		
8	<code>.data @</code>	
9	<code>_cin</code>	
10	<code>.asciz "%d"</code>	

arm 汇编语言中的输入流模版

```

1
2 ldr r0, =_bridge
3 mov r1,
4 bl printf
5
6 .data
7 _bridge:
8 .asciz "%d\n"

```

arm 汇编语言中的输出流模版

(二) 汇编程序编写

1. 斐波那契数列

```

1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int a, b, i, t, n;
6     a = 0;
7     b = 1;
8     i = 1;
9     cin >> n;
10    cout << "a:" << a << endl;
11    cout << "b:" << b << endl;
12    cout << "we are going to loop now! " << endl;
13    while (i < n)
14    {
15        t = b;
16        b = a + b;
17        cout << b << endl;
18        a = t;
19        i = i + 1;
20    }
21    return 0;
22 }

```

源程序

```

1 .arch armv7-a @
2 .arm
3 @r0          r1      printf
4 @
5 @
6 .text @
7 .global main
8 .type main, %function
9 main:
10 push {fp, lr} @ fp          sp          fp   lr          pc          lr
11 sub sp, sp, #4 @          4
12 ldr r0, =_cin
13 mov r1, sp @ sp          r1      scanf          n
14 bl scanf

```

```

15  ldr r6, [sp, #0] @    sp          n
16  add sp, sp, #4 @
17
18  @
19  @ldr r0, =_bridge3
20      @mov r1, r2
21      @bl printf
22
23  mov r4, #0 @a = 0
24  mov r5, #1 @b = 1
25  mov r7, #1 @i = 1
26  @r4    a    r5    b    r7    i    r6    n
27  ldr r0, =_bridge
28  mov r1, r4 @ r4    a    r1
29  bl printf @    a
30  ldr r0, =_bridge2
31  mov r1, r5 @ r5    b    r1
32  bl printf @    b
33      ldr r0, =_bridge4
34      bl printf
35
36  @
37  @ldr r0, =_bridge3
38      @mov r1, r6
39      @bl printf
40      @ldr r0, =_bridge3
41      @mov r1, r7
42      @bl printf
43
44  Loop:
45  @
46  @ldr r0, =_bridge4
47  @bl printf
48  @ldr r0, =_bridge3
49  @mov r1, r6
50      @bl printf
51  @ldr r0, =_bridge3
52      @mov r1, r7
53      @bl printf
54
55  cmp r6, r7
56  ble RETURN @    r7    r6    i    n
57  mov r8, r5 @t = b @r8
58  add r5, r4 @b = a + b
59  ldr r0, =_bridge3
60  mov r1, r5 @ r5    b    r1
61  bl printf @cout << b << endl;
62  mov r4, r8 @a = t
63  add r7, r7, #1 @i = i + 1
64  b Loop
65
66  RETURN:
67  pop {fp, lr} @
68  bx lr @return 0
69  .data @

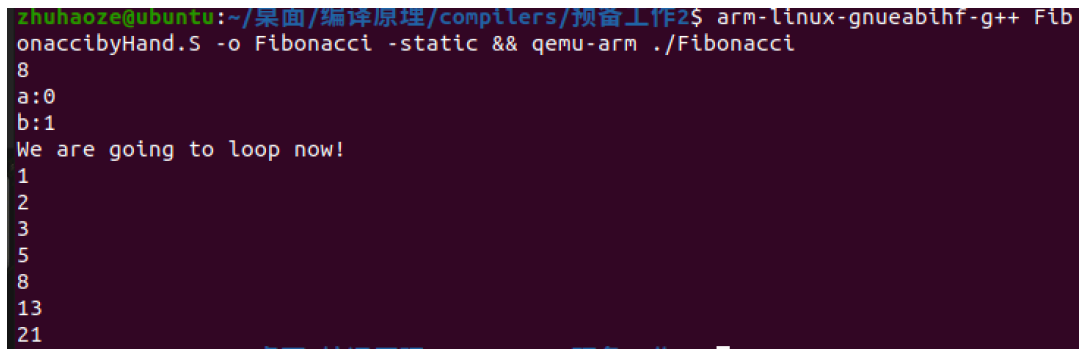
```

```

70 _cin:
71     .asciz "%d"
72
73 _bridge:
74     .asciz "a:%d\n"
75
76 _bridge2:
77     .asciz "b:%d\n"
78
79 _bridge3:
80     .asciz "%d\n"
81
82 _bridge4:
83     .asciz "We are going to loop now! \n"
84
85     .section .note.GNU-stack,"",%progbits @ do you know what's the use of this :-)
```

改写后的汇编代码

运行后的结果如图1所示。可以看出，手写汇编代码正确。



```

zhuhaoze@ubuntu:~/桌面/编译原理/compilers/预备工作2$ arm-linux-gnueabi-g++ FibonacciByHand.S -o Fibonacci -static && qemu-arm ./Fibonacci
8
a:0
b:1
We are going to loop now!
1
2
3
5
8
13
21
```

图 1: 斐波那契程序运行结果

2. 求平方

```

1  #include <iostream>
2  using namespace std;
3  int square(int a){
4      int m;
5      m = a * a;
6      return m;
7  }
8  int main()
9  {
10     int a, s_a;
11     cin >> a;
12     s_a = square(a);
13     cout << s_a << endl;
14     return 0;
15 }
```

源程序

```
1 .arch armv7-a
2 .arm
3
4 .text @
5 .global square
6 square: @function int square(int a)
7     str fp, [sp, #-4]! @pre-index mode, sp = sp -4, push fp
8     mov fp, sp
9     sub sp, sp, #8 @
10    str r0, [fp, #-8] @r0 = [fp, #-8] = a
11    mul r1, r0, r0
12    mov r0, r1
13    add sp, fp, #0
14    ldr fp, [sp], #4
15    bx lr
16
17 .text @
18 .global main
19 .type main, %function
20 main:
21     push {fp, lr}
22     sub sp, sp, #4
23     ldr r0, =_cin
24     mov r1, sp
25     bl scanf
26     ldr r0, [sp, #0] @          r0
27     add sp, sp, #4
28     bl square
29     mov r1, r0
30     ldr r0, =_cout
31     bl printf
32     mov r0, #0
33     pop {fp, lr}
34     bx lr
35
36 .data @
37 _cin:
38     .asciz "%d"
39
40 _cout:
41     .asciz "%d\n"
42
43 .section .note.GNU-stack,"",%progbits @ do you know what's the use of this :-)
```

改写后的汇编代码

运行后的结果如图2所示。可以看出，手写汇编代码正确。

```
zhuhaoze@ubuntu:~/桌面/编译原理/compilers/预备工作2$ arm-linux-gnueabi-g++ Funcb  
yHand.S -o Func -static && qemu-arm ./Func  
5  
25  
zhuhaoze@ubuntu:~/桌面/编译原理/compilers/预备工作2$ arm-linux-gnueabi-g++ Funcb  
yHand.S -o Func -static && qemu-arm ./Func  
20  
400  
zhuhaoze@ubuntu:~/桌面/编译原理/compilers/预备工作2$ arm-linux-gnueabi-g++ Funcb  
yHand.S -o Func -static && qemu-arm ./Func  
100  
10000  
zhuhaoze@ubuntu:~/桌面/编译原理/compilers/预备工作2$ arm-linux-gnueabi-g++ Funcb  
yHand.S -o Func -static && qemu-arm ./Func  
15  
225
```

图 2: 求平方程序运行结果

NIJUB

参考文献

- [1] auto7691. C 语言中的常量与变量. 2018.
- [2] CHENG Jian. C 语言中声明和定义详解. 2016.
- [3] greedyhao. 栈和帧指针使用方法. 2019.