



南開大學
Nankai University

计算机学院
编译原理实验报告

定义语言特性及汇编编程

姓名：李元志、周延霖

学号：2013600、2013921

专业：计算机科学与技术、信息安全

2022 年 10 月 12 日

摘要

本文介绍了 SysY 语言的十大特性，并介绍了相关的形式化定义。另外，本文基于 arm 汇编，实现了四个包括阶乘、数组求和等程序，分析汇编语言中的各个特性。SysY 语言的前五个特性，及其相关的形式化定义，阶乘和数组求和汇编程序由李元志完成。SysY 语言的后五个特性，及其相关的形式化定义，斐波那契数列和自定义程序由周延霖完成。

关键字：SysY 语言 形式化定义 arm 汇编

目录

1	SysY 语言特性	3
1.1	数据类型	3
1.2	变量	3
1.3	常量	3
1.4	运算符	4
1.5	判断	5
1.6	循环	6
1.7	函数	7
1.8	数组	8
1.9	指针	8
1.10	字符串	8
2	形式化定义	9
2.1	变量声明	9
2.2	常量声明	9
2.3	赋值表达式	10
2.4	复合语句	10
2.5	循环及分支语句	10
2.6	算术运算	10
2.7	逻辑运算	11
2.8	关系运算	11
2.9	函数定义	11
2.10	数组指针	11
3	ARM 汇编程序	12
3.1	程序设计思路	12
3.2	实验平台设置	12
3.3	实验方案设计	12
3.4	阶乘	12
3.5	斐波那契数列	14
3.6	数组求和	16
3.7	自定义程序	18
4	代码链接	20
5	总结与展望	20
5.1	总结	20
5.2	未来展望	20

1 SysY 语言特性

1.1 数据类型

数据类型指的是用于声明不同类型的变量或函数的一个广泛的系统。变量的类型决定了变量存储占用的空间，以及如何解释存储的位模式。

序号	数据类型	描述
1	基本类型	它们是算术类型，包括两种类型：整数类型和浮点类型
2	枚举类型	它们也是算术类型，被用来定义在程序中只能赋予其一定的离散整数值的变量。
3	void 类型	类型说明符 void 表明没有可用的值。
4	派生类型	它们包括：指针类型、数组类型、结构类型、共用体类型和函数类型。

表 1: 数据类型

1.2 变量

变量其实只不过是程序可操作的存储区的名称。每个变量都有特定的类型，类型决定了变量存储的大小和布局，该范围内的值都可以存储在内存中，运算符可应用于变量上。变量的名称可以由字母、数字和下划线字符组成。它必须以字母或下划线开头。

变量定义就是告诉编译器在何处创建变量的存储，以及如何创建变量的存储。变量定义指定一个数据类型，并包含了该类型的一个或多个变量的列表。

变量声明向编译器保证变量以指定的类型和名称存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。变量声明只在编译时有它的意义，在程序连接时编译器需要实际的变量声明。变量的声明有两种情况：

- 一种是需要建立存储空间的。例如：int a 在声明的时候就已经建立了存储空间
- 另一种是不需要建立存储空间的，通过使用 extern 关键字声明变量名而不定义它。例如：extern int a 其中变量 a 可以在别的文件中定义的。除非有 extern 关键字，否则都是变量的定义。

1.3 常量

常量是固定值，在程序执行期间不会改变。这些固定的值，又叫做**字面量**。常量可以是任何的基本数据类型，比如整数常量、浮点常量、字符常量，或字符串字面值，也有枚举常量。常量就像是常规的变量，只不过常量的值在定义后不能进行修改。

序号	常量类型	描述
1	整数常量	整数常量可以是十进制、八进制或十六进制的常量。
2	浮点常量	浮点常量由整数部分、小数点、小数部分和指数部分组成。
3	字符常量	字符常量是括在单引号中。
4	字符串常量	字符串字面值或常量是括在双引号””中的。

表 2: 常量类型

1.4 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号，提供了以下类型的运算符：

- 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

算术运算符：

序号	运算符	描述
1	+	把两个操作数相加。
2	-	从第一个操作数中减去第二个操作数。
3	*	把两个操作数相乘。
4	/	分子除以分母。
5	%	取模运算符，整除后的余数。
6	++	自增运算符，整数值增加 1。
7	--	自减运算符，整数值减少 1。

表 3: 算数运算符

关系运算符：

序号	运算符	描述
1	==	检查两个操作数的值是否相等，如果相等则条件为真。
2	!=	检查两个操作数的值是否相等，如果不相等则条件为真。
3	>	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。
4	<	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。
5	>=	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。
6	<=	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

逻辑运算符：

序号	运算符	描述
1	&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。
2		称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。
3	!	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

位运算符：

p	q	p&q	p q	p^q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

表 6: 位运算符

赋值运算符：

序号	运算符	描述
1	=	把右边操作数的值赋给左边操作数。
2	+=	检查两个操作数的值是否相等，如果不相等则条件为真。
3	-=	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。
4	*=	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。
5	/=	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。
6	«=	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。
7	»=	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。
8	&=	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。
9	⊆=	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。
10	=	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 7: 关系运算符

杂项运算符：

序号	运算符	描述
1	sizeof()	返回变量的大小。
2	&	返回变量的地址。
3	*	指向一个变量。
4	?:	条件表达式。

表 8: 杂项运算符

1.5 判断

判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。

序号	语句	描述
1	if 语句	一个 if 语句由一个布尔表达式后跟一个或多个语句组成。
2	if...else 语句	一个 if 语句后可跟一个可选的 else 语句，else 语句在布尔表达式为假时运行。
3	嵌套 if 语句	您可以在一个 if 或 else if 语句内使用另一个 if 或 else if 语句。
4	switch 语句	一个 switch 语句允许测试一个变量等于多个值时的情况。
5	嵌套 switch 语句	您可以在一个 switch 语句内使用另一个 switch 语句。

表 9: 判断语句

1.6 循环

循环作用 有的时候，我们可能需要多次执行同一块代码。一般情况下，语句是按顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。编程语言提供了更为复杂执行路径的多种控制结构。循环语句允许我们多次执行一个语句或语句组。

循环类型 在我们的 SysY 语言中，我们将提供以下几种循环类型，如表3.4所示：

循环类型	描述
while 循环	当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
for 循环	多次执行一个语句序列，简化管理循环变量的代码。
do...while 循环	除了它是在循环主体结尾测试条件外，其他与 while 语句类似。
嵌套循环	可以在 while、for 或 do..while 循环内使用一个或多个循环。

表 10: 循环类型描述

每个循环的语法如下：

while 循环

```

1 while( condition )
2 {
3     statement(s);
4 }
```

for 循环

```

1 for ( init; condition; increment )
2 {
3     statement(s);
4 }
```

do...while 循环

```

1 do
2 {
3     statement(s);
4 } while( condition );
```

嵌套循环

```

1 for ( initialization; condition; increment/decrement )
2 {
3     statement(s);
4     for ( initialization; condition; increment/decrement )
5     {
6         statement(s);
7         ... ..
8     }
```

```

9     ... ..
10 }

```

循环控制语句 在循环中，有一些特殊的关键词可以改变代码的执行顺序，通过它可以实现代码的跳转。我们将提供以下几种循环控制语句，如表3.1所示：

控制语句	描述
break 语句	终止循环或 switch 语句，程序流将继续执行紧接着循环或 switch 的下一条语句。
continue 语句	告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。

表 11: 循环控制语句

1.7 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，在函数退出时被销毁。

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,我们只对函数最基本的功能进行实现。**

1.8 数组

数组介绍 在 C 语言中支持数组数据结构,它可以存储一个固定大小的相同类型元素的顺序集合。数组是用来存储一系列数据,但它往往被认为是一系列相同类型的变量。所有的数组都是由连续的内存位置组成。最低的地址对应第一个元素,最高的地址对应最后一个元素。

数组声明 在我们所定义的 SysY 语言中要声明一个数组,需要指定元素的类型和元素的数量,如下所示:

```
1 type arrayName [ arraySize ];
```

arraySize 必须是一个大于零的整数常量, type 可以是任意有效的 C 数据类型。

数组初始化 对于数组的初始化,可以使用初始化语句,如下所示:

```
1 double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

访问数组元素 数组元素可以通过数组名称加索引进行访问。元素的索引是放在方括号内,跟在数组名称的后边。例如:

```
1 double salary = balance[9];
```

1.9 指针

指针概念 在我们定义的 SysY 语言中,和 C 语言一样,指针指的是内存地址,指针变量是用来存放内存地址的变量。就像其他变量或常量一样,必须在使用指针存储其他变量地址之前,对其进行声明。指针变量声明的一般形式为:

```
1 type *var_name;
```

type 是指针的基类型,它必须是一个有效的数据类型, var_name 是指针变量的名称。用来声明指针的星号 * 与乘法中使用的星号是相同的。所有实际数据类型,不管是整型、浮点型、字符型,还是其他的数据类型,对应指针的值的类型都是一样的,都是一个代表内存地址的长的十六进制数。不同数据类型的指针之间唯一的不同是,指针所指向的变量或常量的数据类型不同。

指针用法 使用指针时会频繁进行以下几个操作:定义一个指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。这些是通过使用一元运算符 * 来返回位于操作数所指定地址的变量的值。

1.10 字符串

和 C 语言中的定义类似,我们的 SysY 语言的字符串实际上是使用空字符 \0 结尾的一维字符数组。因此, \0 是用于标记字符串的结束。

空字符 (Null character) 又称结束符, 缩写 *NUL*, 是一个数值为 0 的控制字符, `\0` 是转义字符, 意思是告诉编译器, 这不是字符 0, 而是空字符。并且在我们的编译器中并不需要把 `null` 字符放在字符串常量的末尾, 此编译器会自动在初始化数组时, 把 `\0` 放在字符串的末尾。

2 形式化定义

名称	符号	名称	符号
数据类型	<code>type</code>	表达式	<code>expr</code>
标识符列表	<code>idlist</code>	算术运算	<code>math_expr</code>
标识符	<code>id</code>	逻辑运算	<code>logical_expr</code>
变量声明	<code>decl</code>	函数定义	<code>funcdef</code>
常量声明	<code>const_init</code>	函数名	<code>funcname</code>
单个数字	<code>number</code>	参数列表	<code>paralist</code>
数字	<code>digit</code>	指针声明	<code>point</code>
一元表达式	<code>unary_expr</code>	数组	<code>arr</code>
赋值表达式	<code>assign_expr</code>	数组列表	<code>arrlist</code>
复合语句	<code>composed_stmt</code>	数组声明	<code>array</code>
分支语句	<code>selection</code>	while 循环语句	<code>while_stmt</code>
for 循环语句	<code>for_stmt</code>		

表 12: 下文中各符号含义

2.1 变量声明

$$type \rightarrow \text{int}|\text{float}|\text{double}|\text{char}$$

$$idlist \rightarrow idlist, id|id$$

$$decl \rightarrow type\ idlist$$

2.2 常量声明

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

2.3 赋值表达式

$$\begin{aligned}
 digit &\rightarrow number\ digit \\
 number &\rightarrow 0|1|2|3|4|5|6|7|8|9 \\
 unary_expr &\rightarrow digit|id \\
 assign_expr &\rightarrow unary_expr = assign_expr \mid \\
 &\quad logical_expr
 \end{aligned}$$

2.4 复合语句

$$composed_stmt \rightarrow stmt \mid \{ composed_stmt; stmt; \}$$

2.5 循环及分支语句

$$\begin{aligned}
 selection &\rightarrow \mathbf{if}(expr)\ stmt \mid \\
 &\quad \mathbf{if}(expr)\ stmt\ \mathbf{else}\ stmt \\
 while_stmt &\rightarrow \mathbf{while}(expr)\ stmt \\
 for_stmt &\rightarrow \mathbf{for}(expr; expr; expr)\ stmt
 \end{aligned}$$

2.6 算术运算

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

2.7 逻辑运算

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

2.8 关系运算

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

2.9 函数定义

$$\begin{aligned} funcdef \rightarrow & type\ funcname(paralist)\ stmt \\ paralist \rightarrow & paralist, parade \mid parade \mid \epsilon \\ parade \rightarrow & type\ id \end{aligned}$$

2.10 数组指针

$$\begin{aligned} point \rightarrow & type * idlist \\ arr \rightarrow & id[digit] \\ arrlist \rightarrow & arr \mid arrlist, arr \\ array \rightarrow & type\ arrlist \end{aligned}$$

3 ARM 汇编程序

3.1 程序设计思路

对于阶乘程序，首先，定义全局变量作为阶乘的参数。其次，在主函数中对阶乘参数赋值，将阶乘封装为一个函数，通过调用函数来实现阶乘来实现阶乘计算。最后，将阶乘的结果打印输出。对于数组求和程序，定义全局变量作为数组。其次，在主函数中对数组初始化。然后，利用循环计算逐个遍历数组元素，计算数组元素之和。最后，将数组和打印输出。对于斐波那契数列程序，首先定义五个变量，在后续中给其赋值，其中 a, b 是循环迭代的值，i 是循环所用的计数器，n 是用户需要输入想让这个程序循环的次数，然后会打印一个带有作者标识的字符串，接下来在每次循环中打印 b 的值可以观察到当前的进度。对于自定义的程序，因为斐波那契数列程序中缺少对 if 判断语句和逻辑运算语言特性的操作，所以在自定义程序中将其体现。整体思路是判断一个数与 100 的大小，并最终输出比较后的结果。

3.2 实验平台设置

处理器	AMD Ryzen 5 4600U with Radeon Graphics 2.10GHz
Cache 大小	512KB
机带 RAM	4.0GB
系统类型	Ubuntu 16.04.7 LTS
内核版本	4.15.0-142-generic

表 13: 实验平台参数

如表13所示，展示了本次实验的相关实验配置。

3.3 实验方案设计

对于阶乘程序，首先，定义全局变量 i、n、f，以及 str0 表示格式化输出字符串。其次，在主函数中，加载全局变量到寄存器中，对其初始化。然后，传递阶乘函数所需的参数，并跳转到阶乘函数中。紧接着，在阶乘函数中开辟栈帧，保存变量值，通过一重循环计算得到结果并保存到 0 号寄存器中。最后，利用函数返回值调用 printf 函数，实现结果的输出。

对于数组求和程序，首先，定义全局变量 arr、sum，以及 str0 表示格式化输出字符串。其次，在主函数中，通过加载全局变量到寄存器中，实现全局变量的初始化。然后，通过 cmp 和 blt 指令实现一重循环，逐个遍历数组元素完成初始化。紧接着，同样采用 cmp 和 blt 指令实现一重循环，统计数组元素之和。最后，传递 printf 函数的参数，实现结果的输出。

对于斐波那契数列程序，首先定义五个变量，在后续中给其赋值，其中 a, b 是循环迭代的值，i 是循环所用的计数器，n 是用户需要输入想让这个程序循环的次数，然后会打印一个带有作者标识的字符串，接下来在每次循环中打印 b 的值可以观察到当前的进度。

对于自定义的程序，因为斐波那契数列程序中缺少对 if 判断语句和逻辑运算语言特性的操作，所以在自定义程序中将其体现。整体思路是判断一个数与 100 的大小，并最终输出比较后的结果。

3.4 阶乘

”阶乘”

```

1  .arch armv5t
2      .comm i,4
3      .comm n,4
4      .comm f,4
5      .text
6      .align 2
7      .section .rodata
8      .align 2
9  __str0:
10     .ascii  "%d\\0"
11     .text
12     .align 2
13
14     .global loop
15 loop:
16     @ 保存栈帧
17     str fp,[sp,#-4]!
18     mov fp,sp
19     @ 开辟栈帧, 保存变量
20     sub sp,sp,#16
21     str r0,[fp,#-8] @i
22     str r1,[fp,#-12] @n
23     str r2,[fp,#-16] @f
24 .L3:
25     cmp r1,r0
26     blt .L2 @branch if exit
27     @ 复合语句
28     mul r4,r0,r2 @f = f * i
29     mov r2,r4
30     add r0,r0,#1 @i = i + 1
31     b .L3
32 .L2:
33     @ 恢复栈帧
34     add sp,fp,#0
35     ldr fp,[sp],#4
36     @ 函数返回
37     bx lr
38     .global main
39 main:
40     push {fp,lr}
41     add fp,sp,#4
42     ldr r2,_bridge @r2 = &f
43     ldr r1,_bridge+4 @r1 = &n
44     ldr r0,_bridge+8 @r0 = &i
45     @ 赋值语句
46     mov r5,#1
47     mov r4,#4
48     mov r3,#2

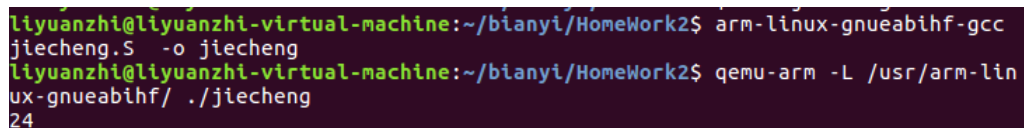
```

```

49  str r5,[r2] @f = 1
50  str r4,[r1] @n = 4
51  str r3,[r0] @i = 2
52  ldr r5,_bridge
53  ldr r4,_bridge+4
54  ldr r3,_bridge+8
55  @ 传递参数
56  ldr r2,[r5]
57  ldr r1,[r4]
58  ldr r0,[r3]
59  @ 函数跳转
60  bl loop
61  mov r1,r2
62  ldr r0,_bridge+12
63  bl printf
64  mov r0,#0
65  pop {fp,pc}
66 _bridge:
67  @ int
68  .word f @r2 and r5
69  .word n @r1 and r4
70  .word i @r0 and r3
71  @ string
72  .word _str0
73
74 .section .note.GNU-stack,"",%progbits

```

运行后的结果如图3.1所示。可以看出，手写汇编代码正确。



```

liyuanzhi@liyuanzhi-virtual-machine:~/bianyi/HomeWork2$ arm-linux-gnueabi-gcc
jiecheng.S -o jiecheng
liyuanzhi@liyuanzhi-virtual-machine:~/bianyi/HomeWork2$ qemu-arm -L /usr/arm-lin
ux-gnueabi/ ./jiecheng
24

```

图 3.1: 阶乘程序运行结果

3.5 斐波那契数列

斐波那契数列源程序

```

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 << endl;

```

```

11     cout << b << endl;
12     cout << "this is zyl's loop" << 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 }

```

依照斐波那契数列手写的汇编代码

```

1  .arch armv7-a @处理器架构
2  .arm
3  @r0是格式化字符串，r1是对应的printf对应的第二个参数
4  .text @代码段
5  .global main
6  .type main, %function
7  @主函数
8  main:
9      push {fp, lr}
10         @将fp的当前值保存在堆栈上，然后将sp寄存器的值保存在fp中，lr中存储的是pc的保存在lr中
11         @ 此处是获取输入的数据
12         sub sp, sp, #4 @在栈中开辟一块大小为4的内存地址，用于存储即将输入的数据
13         ldr r0, =_cin
14         mov r1, sp @将sp的值传输给r1寄存器，使scanf传入的值存储在栈上，即栈顶的值是n
15         bl scanf @执行输入操作
16         ldr r6, [sp, #0] @取出sp指针指向的地址中的内容，即栈顶中的内容（输入的n的值）
17         add sp, sp, #4 @恢复栈顶，释放内存空间
18         @此处是对变量的赋值操作
19         mov r4, #0 @a = 0
20         mov r5, #1 @b = 1
21         mov r7, #1 @i = 1
22         @r4中存a的值，r5中存b的值，r7中存i的值，r6中存n的值
23         ldr r0, =_bridge1
24         mov r1, r4 @将r4中的值即a的值赋予r1
25         bl printf @cout << a << endl;
26         ldr r0, =_bridge1
27         mov r1, r5 @将r5中的值即b的值赋予r1
28         bl printf @cout << b << endl;
29         ldr r0, =_bridge2
30         bl printf @输出自己加载的字符串，用来证实原创性
31         @此处是while循环
32         Loop:
33         cmp r6, r7
34         ble RETURN @比较r7和r6（即i和n）的大小用于跳转

```



```

34  mov r8, r5 @t = b @r8为临时变量的寄存器
35  add r5, r5, r4 @b = a + b
36  ldr r0, =_bridge1
37  mov r1, r5 @将r5中的值即b的值赋予r1
38  bl printf @cout << b << endl;
39  mov r4, r8 @a = t
40  add r7, r7, #1 @i = i + 1
41  b Loop
42
43 RETURN:
44  pop {fp, lr} @上下文切换
45  bx lr @return 0
46 .data @数据段
47 _cin:
48  .asciz "%d"
49
50 _bridge1:
51  .asciz "%d\n"
52
53 _bridge2:
54  .asciz "this is zyl's loop\n"
55
56 .section .note.GNU-stack,"",%progbits @ do you know what's the use of this :-)
```

运行后的结果如图3.2所示。可以看出，手写汇编代码正确。

```

zhouyanlin@ubuntu: ~/Desktop/ubuntu_git/preparation2
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/preparation2$ arm-linux-gnueabi-g++ main.S -o main -static
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/preparation2$ qemu-arm ./main
6
0
1
this is zyl's loop
1
2
3
5
8
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/preparation2$ qemu-arm ./main
10
13
21
34
55
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/preparation2$
```

图 3.2: 斐波那契程序运行结果

3.6 数组求和

”数组求和”

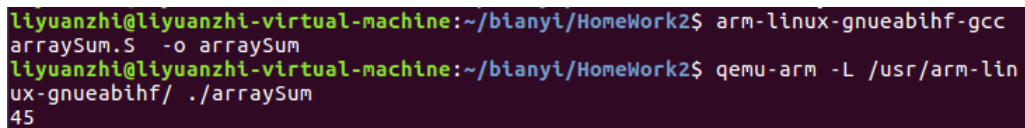
```
1      .arch armv5t
2      .comm arr,4
3      .comm sum,4
4      .text
5      .align 2
6      .section .rodata
7      .align 2
8      _str0:
9      .ascii "%d\\0"
10     .text
11     .align 2
12
13     .global main
14 main:
15     push {fp,lr}
16     add fp,sp,#4
17     @ 变量初始化
18     ldr r0,_bridge @r0 -> &arr
19     mov r1,#0 @r1 -> i
20     .loop1:
21     @ 判断循环条件
22     cmp r1,#10
23     blt .content1
24     b .exit1
25     .content1:
26     @ 数组初始化
27     str r1,[r0] @arr[i] = i
28     add r0,r0,#4 @arr -> arr+4
29     add r1,r1,#1 @i -> i+1
30     b .loop1
31     .exit1:
32
33     @ 变量初始化
34     ldr r0,_bridge @r0 -> &arr
35     mov r1,#0 @r1 -> i
36     ldr r4,_bridge+4
37     ldr r2,[r4]
38     mov r2,#0 @r2 -> sum
39     .loop2:
40     @ 判断循环条件
41     cmp r1,#10
42     blt .content2
43     b .exit2
44     .content2:
45     @ 逐个遍历数组累加
46     ldr r3,[r0]
47     add r2,r2,r3 @sum -> sum+arr[i]
48     add r0,r0,#4 @arr -> arr+4
```

```

49     add r1,r1,#1    @i -> i+1
50     b .loop2
51 .exit2:
52     str r2,[r4]
53
54     @ 打印数组结果
55     mov r1,r2
56     ldr r0,_bridge+8
57     bl printf
58     mov r0,#0
59     pop {fp,pc}
60
61 _bridge:
62     @ int*
63     .word arr
64     @ int
65     .word sum
66     @ string
67     .word __str0
68
69 .section .note.GNU-stack,"",%progbits

```

运行后的结果如图3.3所示。可以看出，手写汇编代码正确。



```

liyuanzhi@liyuanzhi-virtual-machine:~/bianyi/HomeWork2$ arm-linux-gnueabi-gcc
arraySum.S -o arraySum
liyuanzhi@liyuanzhi-virtual-machine:~/bianyi/HomeWork2$ qemu-arm -L /usr/arm-lin
ux-gnueabi/ ./arraySum
45

```

图 3.3: 数组求和运行结果

3.7 自定义程序

本人在汇编程序中主要撰写的语言特性包括 if 分支语句，以及 while/for 循环，支持算术运算（加减乘除、按位与或等）、逻辑运算（逻辑与或等）、关系运算（不等、等于、大于、小于等），由于在斐波那契数列中有一点语言特性并没有体现出来（比如逻辑运算和 if 分支语句），所以又撰写了一个较简单的自定义程序来作为对斐波那契程序中缺少的语言特性的补充，自定义程序如下：

自定义源程序

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int a;
7     cin >> a;
8     if (!(a > 100))
9         cout << "your number is below 100!\n";
10    else

```

```

11     cout << "your number is greater than 100!\n";
12     return 0;
13 }

```

自定义程序改写后的汇编代码

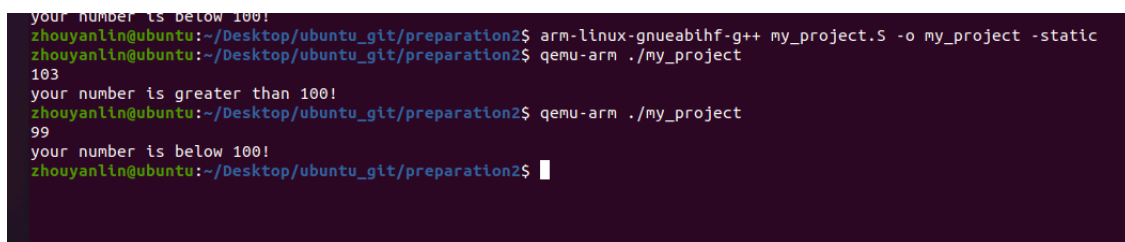
```

1  .arch armv7-a @处理器架构
2  .arm
3  @r0是格式化字符串，r1是对应的printf对应的第二个参数
4  .text @代码段
5  .global main
6  .type main, %function
7  @主函数
8  main:
9      push {fp, lr}
10         @将fp的当前值保存在堆栈上，然后将sp寄存器的值保存在fp中，lr中存储的是pc的保存在lr中
11         @此处是赋值操作
12         sub sp, sp, #4 @在栈中开辟一块大小为4的内存地址，用于存储即将输入的数据
13         ldr r0, __cin
14         mov r1, sp @将sp的值传输给r1寄存器，使scanf传入的值存储在栈上，即栈顶的值是a
15         bl scanf
16         ldr r6, [sp, #0] @取出sp指针指向的地址中的内容，即栈顶中的内容（输入的a的值）
17         add sp, sp, #4 @恢复栈顶，释放内存空间
18
19         @r6中存a的值
20         @接下来是判断操作
21         cmp r6, #0x64
22         ble COMPARE @比较0x64和r6（即100和a）的大小用于跳转
23         @输出"your number is greater than 100!\n"
24         ldr r0, __bridge3
25         bl printf
26         b RETURN @不论怎么样都跳转
27
28 COMPARE:
29         @输出"your number is below 100!\n"
30         ldr r0, __bridge2
31         bl printf
32
33 RETURN:
34         pop {fp, lr} @上下文切换
35         bx lr @return 0
36
37 .data @数据段
38 __cin:
39     .asciz "%d"
40
41 __bridge1:
42     .asciz "%d\n"

```

```
43 .asciz "your number is below 100!\n"  
44  
45 __bridge3:  
46 .asciz "your number is greater than 100!\n"  
47  
48 .section .note.GNU-stack,"",%progbits @ do you know what's the use of this :-)
```

运行后的结果如图3.4所示。可以看出，手写汇编代码正确。



```
your number is below 100!  
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/preparation2$ arm-linux-gnueabi-g++ my_project.S -o my_project -static  
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/preparation2$ qemu-arm ./my_project  
103  
your number is greater than 100!  
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/preparation2$ qemu-arm ./my_project  
99  
your number is below 100!  
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/preparation2$
```

图 3.4: 自定义程序运行结果

4 代码链接

<https://gitlab.eduxiji.net/nku-hlast/lab2/-/tree/master/lab02>

5 总结与展望

5.1 总结

首先，基于 SysY 语言，介绍其主要的十大特性，同时给出了与其相关的形式化定义。其次，基于 arm 汇编程序，实现了四个常见的程序，包括阶乘、数组求和等等。程序大致覆盖了所有的特性，同时本文以注释的形式给出了各个相关特性，及相关分析。

5.2 未来展望

本次是第二次实验，对编译方面的相关知识有了更深入、更全面的理解，对于各个部分所要实现的功能以及未来自己所要实现的语言特性有了一定的掌握，期望我们小组在这个学期可以取得一个满意的结果。