



南開大學
Nankai University

计算机学院
编译原理实验报告

预备工作 2

姓名：张书睿 谢子涵
学号：2010521 2010507
专业：计算机科学与技术

2022 年 12 月 25 日

目录

1 实验平台配置	2
2 第一部分——CFG 设计	2
2.1 分工情况	2
2.2 编译单元	2
2.3 声明	2
2.4 常量声明、常数定义与常量初值（包括数组）	2
2.5 基本类型	3
2.6 变量声明、定义和初值（包括数组）	3
2.7 函数定义，类型以及形参表	3
2.8 语句块、语句块项以及语句	3
2.9 算术表达式	4
2.10 逻辑表达式和关系表达式	4
3 第二部分——汇编程序编写	5
3.1 分工情况	5
3.2 斐波那契数列计算	5
3.3 组合数计算	7
3.4 判断质数合数	10
3.5 逆置数组	13
4 思考题——如何设计翻译 SysY 程序的编译器	16
4.1 数据结构设计	16
4.2 算法设计	16

1 实验平台配置

实验配置配置条件如下表1。

表 1: 实验条件配置

CPU	Intel(R) Core(TM) i7-10870H CPU @ 2.20GHz 2.21 GHz
OS	Ubuntu 20.04.5 LTS
IDE	CLion 2021.2.3
VTune	Intel VTune Profiler 2022.0

2 第一部分——CFG 设计

2.1 分工情况

本部分作业（SysY 上下文无关语法设计）由两人分析观察 EBNF 的产生式，张书睿同学负责前半部分 CFG 设计，谢子涵同学负责后半部分设计，共同讨论修改得到以下结果。

2.2 编译单元

$$CompUnit \rightarrow Decl \mid FuncDef \mid CompUnit Decl \mid CompUnit FuncDef$$

其中 $CompUnit$ 表示编译单元， $Decl$ 表示声明语句， $FuncDef$ 表示函数定义。

2.3 声明

$$Decl \rightarrow ConstDecl \mid VarDecl$$

其中 $ConstDecl$ 表示常量声明， $VarDecl$ 表示变量声明。

2.4 常量声明、常数定义与常量初值（包括数组）

$$ConstDecl \rightarrow \mathbf{const} \ BType \ ConstDefList;$$

$$ConstDefList \rightarrow ConstDefList , ConstDef \mid ConstDef$$

$$ConstDef \rightarrow \mathbf{Ident} \ ArrayConstIndex = ConstInitVal$$

$$ConstInitVal \rightarrow ConstExp \mid \{ ConstExpList \} \mid \{ ConstInitVal \} \mid ConstInitVal, \{ ConstExpList \}$$

$$ConstExpList \rightarrow ConstExp \mid ConstExpList , ConstExp$$

$$ArrayConstIndex \rightarrow [ConstExp] \mid ArrayConstIndex[ConstExp] \mid \epsilon$$

其中 $ConstDecl$ 表示常量声明语句； $ConstDefList$ 表示常量定义标识符列表； $ConstDef$ 表示 $ConstDefList$ 中的一个常量定义（包括数组和单个常量）； $ConstInitVal$ 表示一个常量的初值或一个常量数

组的初始化列表；ConstExpList 表示常量表达式列表；ConstExp 表示变量表达式；ArrayConstIndex 表示数组的维度定义，如 [m][n]； ϵ 为空字，是终结符。

2.5 基本类型

$$BType \rightarrow \text{int} \mid \text{float}$$

其中 BType 为基本类型，int 和 float 为具体的基本数据类型，为终结符。

2.6 变量声明、定义和初值（包括数组）

$$VarDecl \rightarrow BType \text{ VarDefList};$$

$$VarDefList \rightarrow VarDefList, VarDef \mid VarDef$$

$$VarDef \rightarrow \text{Ident} \text{ ArrayConstIndex} \mid \text{Ident} \text{ ArrayConstIndex} = \text{InitVal}$$

$$\text{InitVal} \rightarrow \text{ExpList} \mid \{ \text{InitVal} \} \mid \text{InitVal}, \{ \text{ExpList} \}$$

$$\text{ExpList} \rightarrow \text{Exp} \mid \text{ExpList}, \text{Exp}$$

其中，VarDecl 表示变量声明；VarDefList 表示变量定义标识符列表；VarDef 表示 varDefList 中的一个变量定义（包括数组和单个变量）；InitVal 表示一个变量的初值或一个变量数组的初始化列表；ArrayConstIndex 表示数组的维度定义，如 [m][n]，与 2.3 常量中相同；ExpList 表示变量表达式列表；Exp 表示变量表达式。

2.7 函数定义，类型以及形参表

$$FuncDef \rightarrow FuncType \text{ Ident } (\text{FuncFParams}) \text{ Block}$$

$$FuncType \rightarrow \text{void} \mid \text{int} \mid \text{float}$$

$$FuncFParams \rightarrow FuncFParam \mid FuncFParams, FuncFParam$$

$$FuncFParam \rightarrow BType \text{ Ident} \text{ ArrayConstIndex}$$

其中，FuncDef 是函数定义，FuncType 是函数类型，包括 int，void 和 float 三种，FuncFParams 是函数参数表，可以是数组形式。

2.8 语句块、语句块项以及语句

$$Block \rightarrow \{ \text{BlockItemList} \}$$

$$\text{BlockItemList} \rightarrow \text{BlockItem} \mid \text{BlockItemList} \text{ BlockItem}$$

$$\text{BlockItem} \rightarrow \text{Decl} \mid \text{Stmt}$$

$$\begin{aligned}
Stmt &\rightarrow LVal = Exp ; \mid Exp ; \mid Block \\
&\mid \text{if} (Cond) Stmt ElseStmt \\
&\mid \text{while} (Cond) Stmt \\
&\mid \text{break} ; \mid \text{continue} ; \\
&\mid \text{return} Exp ; \\
&\mid \epsilon \\
ElseStmt &\rightarrow \text{else} Stmt \mid \epsilon
\end{aligned}$$

其中 Block 是语句块, BlockItemList 是语句块项表, BlockItem 是语句块项, Decl 是定义语句, Stmt 是语句, 以上定义了 if,while,break,continue,return 等语句。

2.9 算术表达式

$$\begin{aligned}
ConstExp &\rightarrow AddExp \\
Exp &\rightarrow AddExp \\
AddExp &\rightarrow MulExp \mid AddExp + MulExp \mid AddExp - MulExp \\
MulExp &\rightarrow UnaryExp \mid MulExp * UnaryExp \mid MulExp / UnaryExp \mid MulExp \% UnaryExp \\
UnaryExp &\rightarrow PrimaryExp \mid \text{Ident} (FuncRParams) \mid UnaryOp UnaryExp \\
PrimaryExp &\rightarrow (Exp) \mid LVal \mid Number \\
LVal &\rightarrow \text{Ident} ArrayIndex \\
ArrayIndex &\rightarrow [Exp] \mid ArrayIndex[Exp] \mid \epsilon \\
Number &\rightarrow \text{IntConst} \mid \text{floatConst} \\
FuncRParams &\rightarrow Exp \mid FuncRParams , Exp \mid \epsilon \\
UnaryOp &\rightarrow + \mid - \mid !
\end{aligned}$$

其中, ConstExp 表示常量表达式; Exp 表示变量表达式; AddExp 表示加减表达式; MulExp 表示乘除模表达式; UnaryExp 表示一元表达式; PrimaryExp 表示基本表达式; LVal 表示左值表达式, 主要用于数组的取值, 其中 Ident 表示标识符, ArrayIndex 表示数组的下标, 如 [m][n]; Number 表示数值, 包括 int 型和 float 型数值; FuncRParams 表示函数实参表; UnaryOp 表示单目运算符, 包括 +、-、!。

2.10 逻辑表达式和关系表达式

$$\begin{aligned}
Cond &\rightarrow LOrExp \\
LOrExp &\rightarrow LAndExp \mid LOrExp \parallel LAndExp
\end{aligned}$$

$$LAndExp \rightarrow EqExp \mid LAndExp \ \&\& \ EqExp$$

$$EqExp \rightarrow RelExp \mid EqExp == RelExp \mid EqExp != RelExp$$

$$RelExp \rightarrow AddExp \mid RelExp < AddExp \mid RelExp > AddExp \mid RelExp <= AddExp \mid RelExp >= AddExp$$

其中，Cond 表示条件表达式；LOrExp 表示逻辑或表达式；LAndExp 表示逻辑与表达式；EqExp 表示相等性表达式；RelExp 表示关系表达式。

3 第二部分——汇编程序编写

3.1 分工情况

本部分汇编程序设计分工为：每人分别设计两个不同的汇编程序，以展示 arm 汇编程序的一些不同特性，前两个程序由张书睿同学设计编写，后两个程序由谢子涵同学设计编写。汇编程序代码与说明文档分别存放在 [zsr_Git](#) 和 [xzh_Git](#)。

3.2 斐波那契数列计算

本程序对应的 c 代码大致如下所示：

```

1  #include <stdio.h>
2  int fibo(int n){
3      if (n==1||n==0){
4          return 1;
5      }
6      return fibo(n-1) + fibo(n-2);
7  }
8  int main(){
9      int k = fibo(5);
10     printf("%d",k);
11     return 0;
12 }
```

```

1  .text
2  .section .rodata
3  _str:
4  .ascii "%d\012\000" @ "%d\n\0"
5  .align 2
6  .global fibo
7  fibo: @ function int fibo(int )
8  push {fp, lr}
9  add fp, sp, #8
10 sub sp, sp, #12
11 str r0, [fp, #-16] @ 保存参数n值
```

```

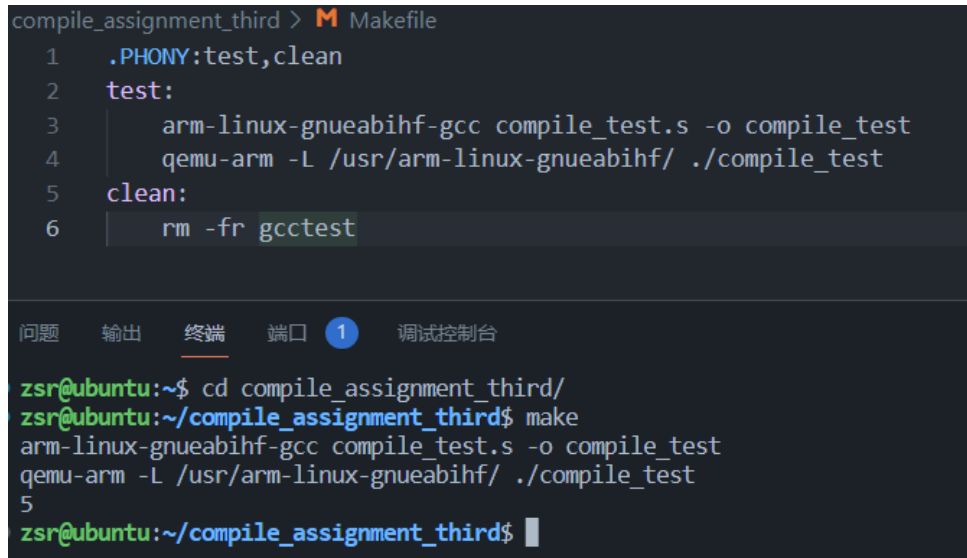
12    ldr r3, [fp, #-16]    @ 将r0值转移到r3中, 再进行比较
13    cmp r3, #1           @ 将n与1进行比较
14    beq .L1              @ 如果等于1, 则进入if语句块
15    cmp r3, #0           @ 再和0比较
16    bne .L2              @ 如果等于0, 则进入if语句块, 不等于0则跳出
17    .L1:                 @ if语句块
18    mov r3, #1           @ 返回值为1
19    b .L3                 @ 进入return部分
20    .L2:
21    sub r0, r3, #1
22    bl fibo               @ fibo(n-1):第一次递归
23    mov r4, r0            @ 保存第一次结果到r4
24    ldr r3, [fp, #-16]    @ 加载提前保存的参数n
25    sub r3, r3, #2
26    mov r0, r3
27    bl fibo
28    @ fibo(n-2):第二次递归, 为了防止寄存器调用冲突, 使用已经提前保存的数值
29    add r3, r4, r3        @ 计算两者和
30    .L3:
31    mov r0, r3            @返回值存入r0
32    sub sp, fp, #8
33    pop {fp, pc}
34
35    .align 2
36    .global main
37    main:
38    push {fp, lr}
39    add fp, sp, #4
40    sub sp, sp, #8
41    mov r0, #5             @ fibo函数, 使用参数常数5
42    bl fibo
43    mov r1, r0
44    ldr r0, _bridge
45    @ 将字符串常量地址作为第一个参数,
46    fibo函数返回值保存在r1中, 作为第二个参数
47    bl printf
48    mov r0, #0
49    sub sp, fp, #4
50    pop {fp, pc}
51
52    _bridge:
53    .word _str

```

```

54 .section .note.GNU-stack,"",%progbits
55 @ 保护代码，禁止生成可执行堆栈

```



```

compile_assignment_third > M Makefile
1 .PHONY:test,clean
2 test:
3     arm-linux-gnueabi-gcc compile_test.s -o compile_test
4     qemu-arm -L /usr/arm-linux-gnueabi/ ./compile_test
5 clean:
6     rm -fr gcctest

问题  输出  终端  端口 1  调试控制台

zsr@ubuntu:~$ cd compile_assignment_third/
zsr@ubuntu:~/compile_assignment_third$ make
arm-linux-gnueabi-gcc compile_test.s -o compile_test
qemu-arm -L /usr/arm-linux-gnueabi/ ./compile_test
5
zsr@ubuntu:~/compile_assignment_third$

```

图 3.1: 第一个汇编程序验证结果

本汇编程序计算的是斐波拉契数列的第五个数（5），验证结果如上图3.1所示。在本汇编程序中，体现了 CFG 设计中支持 SysY 语言的函数相关特性以及定义常量、变量相关特性。

3.3 组合数计算

本程序等价的 c 代码大致如下所示：

```

1  #include <stdio.h>
2  int factorial(int n)
3  {
4      int m = 1;
5      for (int i = 1; i <= n; i++){
6          m *= i;
7      }
8      return m;
9  }
10 int C(int n, int m)
11 {
12     return factorial(n) / (factorial(m)*factorial(n - m));
13 }
14
15 int main()
16 {
17     printf("%d\n",C(10,2));
18     return 0;

```


19 }

```

1  .text
2  .section  .rodata
3  _str:
4  .ascii  "%d\012\000"      @ "%d\n\0"
5
6  factorial: @ int factorial(int k) --> k!
7  str fp, [sp, #-4]
8  sub sp, sp, #4
9  mov fp, sp
10 sub sp, sp, #20           @ 开辟栈帧
11 str r0, [fp, #-16]       @ 提前将参数k保存在栈中
12 mov r3, #1               @ int m=1, 保存在r3寄存器
13 mov r4, #1               @ int i=1, 保存在r4寄存器
14 b .L1
15 .L0:
16 mul r3, r3, r4           @ m*=i
17 add r4, r4, #1           @ i++
18 .L1:
19 cmp r4, r0               @ m <= n, 则跳回
20 ble .L0
21 mov r0, r3               @ 保存返回值到r0
22 add sp, fp, #0
23 ldr fp, [sp], #4
24 bx lr
25
26 C: @ int C(int m, int n) --> C^m_n
27 push {fp, lr}
28 add fp, sp, #12
29 sub sp, sp, #8
30 str r0, [fp, #-16]       @ 提前将m, n的值保存在栈中
31 str r1, [fp, #-20]
32 ldr r0, [fp, #-16]
33 bl factorial             @ 计算m!
34 mov r5, r0               @
    使用r5保存中间值, 因为factorial函数没使用
35 ldr r0, [fp, #-20]
36 bl factorial             @ 计算n!
37 mov r6, r0               @
    使用r6保存中间值, 因为factorial函数没使用
38 ldr r2, [fp, #-16]

```

```
39  ldr r3, [fp, #-20]
40  sub r3, r2, r3
41  mov r0, r3
42  bl factorial                @ 计算(m-n)!
43  mov r3, r0
44  mul r3, r3, r6
45  mov r1, r3
46  mov r0, r5
47  bl __aeabi_idiv            @ 调用除法函数, 计算m!/(m-n)!/n!
48  sub sp, fp, #12
49  pop {fp, pc}
50
51  .global main
52  .type main, %function
53  main:
54  push {fp, lr}
55  add fp, sp, #4
56  mov r1, #2
57  mov r0, #10
58  bl C
59  mov r1, r0
60  ldr r0, _bridge
61  bl printf
62  mov r0, #0
63  pop {fp, pc}
64  _bridge:
65  .word _str
66  .section .note.GNU-stack,"",%progbits
67  @ 保护代码, 禁止生成可执行堆栈
```



```

M Makefile M X  compile_test M
compile_assignment_third > M Makefile
1  .PHONY:test,clean
2  test:
3      arm-linux-gnueabi-gcc compile_test2.s -o compile_test2
4      qemu-arm -L /usr/arm-linux-gnueabi/ ./compile_test2
5  clean:
6      rm -fr gcctest

问题  输出  终端  端口 1  调试控制台
• zsr@ubuntu:~$ cd compile_assignment_third/
• zsr@ubuntu:~/compile_assignment_third$ make
arm-linux-gnueabi-gcc compile_test.s -o compile_test
qemu-arm -L /usr/arm-linux-gnueabi/ ./compile_test
5
• zsr@ubuntu:~/compile_assignment_third$ make
arm-linux-gnueabi-gcc compile_test2.s -o compile_test2
qemu-arm -L /usr/arm-linux-gnueabi/ ./compile_test2
45
• zsr@ubuntu:~/compile_assignment_third$

```

图 3.2: 第二个汇编程序验证结果

本程序计算的是组合数 $C_m^n (m = 10, n = 2)$ 验证结果如上图3.2所示。在本汇编程序中，体现了 CFG 设计中支持 SysY 语言的函数相关特性、循环语句相关特性以及定义常量、变量相关特性。

3.4 判断质数合数

采用 arm 汇编语言实现了一个判断质数和合数的程序。输入数 a，如果是质数，则输出“Prime number”，如果是合数，则输出“Composite number”。程序体现的 SysY 语言特性有：全局变量、局部变量、函数、if 语句、return 语句、for 语句以及一些逻辑运算语句和算术运算语句。具体实现的功能为：

```

1  #include<stdio.h>
2  int a=0;
3  int isPrime(int n){
4      if(n==1)
5          return 0;
6      int i=2;
7      for(i=2;i<n;i++){
8          if(n%i==0)
9              return 0;
10     }
11     return 1;
12 }
13 int main(){
14     scanf("%d", &a);

```

```

15     if(isPrime(a)){
16         printf("Prime number\n");
17     }else
18         printf("Composite number\n");
19     return 0;
20 }

```

arm 汇编代码如下，具体的解释见代码中注释：

```

1     .comm a,4                @全局变量a
2
3     .align 2                 @以2^2字节对齐
4     .global isPrime         @将isPrime函数名添加到全局符号表中
5     .type isPrime,%function  @声明isPrime类型为函数
6 isPrime:
7 @ function int isPrime(int n) 判断n是否为质数,若是则返回1,否则返回0
8     push {fp, lr}           @将fp和lr压入栈
9     add fp, sp, #4           @设置新的fp
10    sub sp, sp, #16          @ sp=sp-16,为该函数分配栈帧空间
11    str r0, [fp, #-16]       @参数n入栈
12    cmp r0, #1               @ r0-1,并对相应的状态位置位
13    bne .L2                  @若n不等于1,则跳转到标签.L2
14    mov r0,#0                @若n等于1,则返回值为0,放到r0中
15    b isPrimeRn              @若n等于1,则跳转到标签isPrimeRn
16 .L2:
17    mov r1,#2                 @ r1=2,即变量int i=2
18    str r1,[fp, #-8]         @将变量i的值存到fp-8处
19 LOOP:
20    ldr r1, [fp, #-8]         @取出变量i的值到r1
21    ldr r2, [fp, #-16]       @取出n的值到r2
22    cmp r1, r2
23    blt .L3                  @若r1<r2(即i<n),就跳转到标签.L3
24    mov r0, #1               @若r1>=r2(即i>=n),就结束循环,返回值为1,存入r0
25    b isPrimeRn              @跳转到标签isPrimeRn
26 .L3:
27    mov r0,r2                 @ r0中存n的值,r1中存i的值
28    bl __aeabi_idivmod        @模运算
29    mov r3,r1
30    cmp r3,#0                 @比较n%i的结果是否等于0
31    bne .L4                  @如果不等于0,就跳转到标签.L4
32    mov r0,#0
33    b isPrimeRn              @跳转到标签isPrimeRn
34 .L4:

```

```

35     ldr r1, [fp, #-8]           @ 令 i++
36     add r1, r1, #1
37     str r1, [fp, #-8]
38     b LOOP                     @ 继续循环
39 isPrimeRn:
40     sub sp, fp, #4
41     pop {fp, pc}
42
43     .section    .rodata
44     .align     2
45 _str0:
46     .ascii     "%d\000"
47     .align     2
48 _str1:
49     .ascii     "Prime number\n\000"
50     .align     2
51 _str2:
52     .ascii     "Composite number\n\000"
53
54     .text
55     .align     2
56     .global   main
57     .type     main, %function
58 main:
59     push {fp, lr}
60     add fp, sp, #4
61     ldr r1, _bridge             @ r1=&a
62     ldr r0, _bridge+4           @ *r0="%d\000"
63     bl __isoc99_scanf           @ scanf("%d", &a);
64     ldr r2, _bridge             @ r2=&a
65     ldr r0, [r2]                @ r0=a
66     bl isPrime                  @ 调用函数 isPrime, 判断 a 是否为质数
67     cmp r0, #0                  @ 比较 r0 中的 isPrime 返回值是否为 0
68     beq .L5                     @ 若返回值为 0, 则跳转到标签 .L5
69     ldr r0, _bridge+8           @ *r0="Prime number\000"
70     bl printf
71     b END
72 .L5:
73     ldr r0, _bridge+12          @ *r0="Composite number\000"
74     bl printf
75 END:
76     mov r0, #0

```

```

77     pop {fp, pc}
78
79 _bridge:
80     .word a
81     .word _str0
82     .word _str1
83     .word _str2
84     .section .note.GNU-stack,"",%progbits

```

```

● zihan@ubuntu:~/workspace/compiler_h2$ arm-linux-gnueabi-gcc test1.s -o test1
● zihan@ubuntu:~/workspace/compiler_h2$ qemu-arm -L /usr/arm-linux-gnueabi/ ./test1
2
Prime number
● zihan@ubuntu:~/workspace/compiler_h2$ qemu-arm -L /usr/arm-linux-gnueabi/ ./test1
3
Prime number
● zihan@ubuntu:~/workspace/compiler_h2$ qemu-arm -L /usr/arm-linux-gnueabi/ ./test1
4
Composite number
● zihan@ubuntu:~/workspace/compiler_h2$ qemu-arm -L /usr/arm-linux-gnueabi/ ./test1
17
Prime number

```

图 3.3: 判断质数合数汇编程序验证结果

如图3.3所示, 该 arm 汇编程序能够正确编译并执行, 输入 2、3、4、17 均能正确判断是合数还是质数, 说明程序正确。

3.5 逆置数组

采用 arm 汇编语言实现了一个将数组元素逆置的程序。输入 10 个数, 为数组 a 赋值, 会将数组 a 中的元素逆置, 并再次输出数组 a 的元素进行验证。程序体现的 SysY 语言特性有: 数组、变量、if 语句、return 语句、for 语句、while 语句以及一些逻辑运算语句和算术运算语句。具体实现的功能为:

```

1  #include <stdio.h>
2  int main(){
3      int a[10], i;
4      for(i=0; i<10; i++){
5          scanf("%d", &a[i]);
6      }
7      i=0;
8      int j=9;
9      while (i<j)
10     {
11         int t=a[i];
12         a[i]=a[j];
13         a[j]=t;

```

```

14         i++;
15         j--;
16     }
17     printf("Array after rotation:\n");
18     for(i=0;i<10;i++){
19         printf("%d  ",a[i]);
20     }
21     printf("\n");
22     return 0;
23 }

```

arm 汇编代码如下，具体的解释见代码中注释：

```

1     .text
2     .section .rodata
3     .align 2
4     _str0:
5     .ascii "%d\000"
6     .align 2
7     _str1:
8     .ascii "Array after rotation:\n\000"
9     .align 2
10    _str2:
11    .ascii "%d \000"
12    .align 2
13    _str3:
14    .ascii "\n\000"
15    .data
16    array:.word 0,0,0,0,0,0,0,0,0,0    @定义一个长度为10的数组
17
18    .text
19    .align 2
20    .global main
21    .type main, %function
22    main:
23        stmfd sp!,{lr}
24        mov r4,#0                    @ r4保存变量i的值,初始化i=0
25        ldr r5, =array                @ 将数组a的首地址读取到r5中
26        b .L2
27    Loop1:
28        ldr r0, _bridge                @ *r0="%d\000"
29        mov r1,r5                      @
        令r1为此刻指向的数组元素的地址,作为scanf的第二个参数

```

```

30     add r5,#4           @ 给r5自增4,指向数组下一个元素对应的地址
31     bl scanf
32     add r4,#1           @ i++
33 .L2:
34     cmp r4,#10          @将r4中存的i和10比较
35     blt Loop1           @若r4<10,就跳转到标签Loop1,继续第一个for循环
36
37     mov r4,#0           @ r4保存变量i的值,重置i=0
38     mov r3,#9           @ r3保存变量j的值,初始化为9
39     ldr r0,=array        @ r0保存a[i]的地址,初始化为数组首地址
40     add r1,r0,#36        @ r1保存a[j]的地址,初始化为数组末尾地址
41     b .L3
42 Loop2:
43     ldr r2,[r0]          @ r2保存a[i]的值
44     ldr r6,[r1]          @ r6保存a[j]的值
45     str r2,[r1]          @ a[j]为原来a[i]的值
46     str r6,[r0]          @ a[i]为原来a[j]的值
47     add r0,#4           @ 给r0自增4,指向数组下一个元素对应的地址
48     sub r1,#4           @ 给r1自增4,指向数组前一个元素对应的地址
49     add r4,#1           @ i++
50     sub r3,#1           @ j--
51 .L3:
52     cmp r4,r3           @ 比较r4和r3,即比较i和j的大小
53     blt Loop2           @若r4<r3(即i<j),则跳转到标签Loop2,继续while循环
54
55     ldr r0,_bridge+4     @ *r0="Array after rotation:\n\000"
56     bl printf
57
58     mov r4,#0           @ r4保存变量i的值,重置i=0
59     ldr r5,=array        @
                           r5保存循环中当前数组元素的地址,重新赋值为数组a的首地址
60     b .L4
61 Loop3:
62     ldr r0,_bridge+8     @ *r0="%d \000"
63     ldr r1,[r5],#4       @将数组的首地址读取到r5中,在循环中将r5的值传给r1,并向后取址
64     bl printf
65     add r4,#1           @ i++
66 .L4:
67     cmp r4,#10          @将r4中存的i和10比较
68     blt Loop3           @若r4<10,就跳转到标签Loop3,继续第二个循环
69     ldr r0,_bridge+12    @ *r0="\n\000"

```



```

70     bl printf
71     mov r0, #0
72     ldmfd sp!,{lr}
73     mov pc, lr
74
75 _bridge:
76     .word _str0
77     .word _str1
78     .word _str2
79     .word _str3
80     .section .note.GNU-stack,"",%progbits

```

```

● zihan@ubuntu:~/workspace/compiler_h2$ arm-linux-gnueabi-gcc test2.s -o test2
● zihan@ubuntu:~/workspace/compiler_h2$ qemu-arm -L /usr/arm-linux-gnueabi/ ./test2
1 2 3 4 5 6 7 8 9 10
Array after rotation:
10 9 8 7 6 5 4 3 2 1

```

图 3.4: 逆置数组汇编程序验证结果

如图3.4所示, 该 arm 汇编程序能够正确编译并执行, 输入 1、2、3、4、5、6、7、8、9、10 为数组 a 赋初值后, 再次输出的数组 a 的元素内容的确是逆置后的结果, 说明程序正确。

4 思考题——如何设计翻译 SysY 程序的编译器

4.1 数据结构设计

- **抽象语法树 (AST):** 所设计编译器中最大最基础的数据结构。由于需要将 SysY 程序翻译为汇编程序, 需要首先通过语法分析与语义分析将程序转化为一个抽象语法树 (Abstract Semantic Tree, AST) 的数据结构中存储, 包含语义分析与语法分析结果。
- **标识符符表:** 程序定义标识符后, 需要对标识符名以及其值保存地址或寄存器 (如果有) 进行保存, 在后续操作中才可进一步调用。
- **函数节点与符表:** 由于需要对函数调用进行处理, 因此对每个函数都需要单独保存其信息, 包括其定义函数名、参数表等, 并且单独设置一个**函数符表**进行保存, 在调用时进行查表即可。

4.2 算法设计

该编译器的核心算法在于对于抽象语法树进行后序遍历的过程中, 遍历到子节点时同时翻译输出相应的汇编程序代码, 一些简单的记录与翻译规则如下所示:

- **常数:** 直接使用汇编指令中的立即数#x (x 为立即数, 视当时情况进行选择) 进行替代。
- **标识符与常量标识符:** 如果是常量标识符的定义语句, 则将读取的符号名保存在标识符表中, 并保存其地址, 翻译出的语句保存在.section .rodata下, 大致为type value, 并在汇编程序末用.word对地址进行保存, 后续操作中识别到该常量标识符时, 使用一个空闲的寄存器进行加载

使用；如果是普通标识符则不需单独保存常量地址，只需保存寄存器位置即可（寄存器位置也可变），后续使用标识符的操作中查表使用相应寄存器即可即可。

- **函数定义及调用：**如果是函数的定义语句，则提前对函数名与参数表进行保存，并翻译出相应的函数定义语句基本框架（函数名`function:`，压栈操作`push{fp,lr}`，退栈操作`pop{fp,pc}`等，如果有返回值，还需要记得将返回结果保存在`r0`寄存器），然后进一步翻译函数体中运算内容；对于函数的调用语句，则根据实参表提前将实参依次保存进相应的寄存器中，再翻译出`bl function`的指令对函数进行调用。
- **运算指令：**对于简单的运算指令，只需要查表得知操作数的保存位置，然后翻译出相应的汇编指令即可，如乘法操作对应`mul`指令等；值得一提的是，arm 指令集中对除法操作没有对应的汇编指令，但可以先将操作数值分别保存在`r0,r1`中，调用函数`__aeabi_idiv`进行除法操作即可。
- **循环语句：**循环语句的关键在于判断置位指令（`cmp`等）与跳转指令（`ble`等），只需将条件翻译为一个判断置位指令，在循环语句代码块上加上标签，再添加合适的跳转指令即可。