南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

编译系统原理

定义你的编译器、汇编编程并熟悉使用工具

汤清云 2013536 朱莞尔 2013289

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 10 月 16 日

摘要

SysY 语言是 C 语言的一个子集, 支持 int 类型和元素为 int 类型且按行优先存储的多维数组类型, 其 I/O 是以运行时库方式提供。而 arm 汇编语言是一种被广泛使用的汇编语言, 其指令集精简, 适用于很多处理器。本实验要完成的任务就是了解 SysY 语言特性, 给出其形式化定义, 描述 SysY 语言子集的 CFG 表示, 并设计尽可能全面地包含支持的语言特性的 SysY 程序, 编写等价的 ARM 汇编程序, 用汇编器生成可执行程序, 调试通过、能正常运行得到正确结果。

关键词：SysY ARM Compile CFG

目录

一、 引言

在实验一结束后，我们初步了解了一个编译器的大致流程是什么样子的，并且明白了 LLVM 中间语言的形式以及撰写方法。由此我们更进一步，延伸出了以下思考：

1.SysY 语言的特性能否有更数学语言化的描述？如果有的话该用什么实现？

2. 由上我们自然联想到 CFG，然而具体实现形式如何？

3. 在学习了 LLVM 中间代码后，机器如何实现从 c 语言到汇编语言的转换？

本实验即是基于以上思考所做出的实例探究。

二、 实验前期

(一) 实验要求

1. 基于“预备工作一”，确定本小组要实现的编译器支持哪些 SysY 语言特性，给出其形式化定义——学习教材第二章以及第二章讲义中的 2.2 节；参考 SysY 中巴克斯瑙尔范式定义，用上下文无关文法描述本小组的 SysY 语言子集。

2. 设计几个 SysY 程序（例如预备工作一给出的阶乘或者斐波那契程序），编写等价的 ARM 汇编程序，使用汇编器生成可执行程序，调试通过、能正常运行得到正确结果。

3.* 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是怎样的？【注：编译器不能只会翻译一个源程序，而是要有能力翻译所有合法的 SysY 程序。而穷举所有 SysY 程序是不可能的——因此实现每个语言特性的翻译即可；每个语言特性仍然有无穷多个合法的实例——因此需要符号化，即语法制导翻译，详见将以 2.8 节。】

4.* 鼓励尝试设计语法制导定义/翻译模式，实现简单的 SysY 程序到汇编程序的翻译，并且通过 Bison 进行实验。

(二) 实验分工

1. 共同讨论分析 SysY 语言所具有的特性，确定本次实验要实现的编译器所支持的 SysY 语言特性，给出其形式化定义，使用上下文无关文法描述 SysY 语言子集，并给出 SysY 语言的 CFG 描述。汤清云负责**函数、常变量声明、语句**部分的上下文无关文法转换；朱莞尔负责**表达式、标识符、数值常量**部分。

2. 以完全数程序为基础，在熟悉掌握 ARM 语言编写后对实验一中的程序进行汇编，共同讨论如何添加代码以尽可能最大程度体现出 SysY 的多种语言特性。

3. 确定 c 语言代码程序后，汤清云负责程序一中**内层 for 循环**的编写以及程序二中**内层 for 循环**和 **while、if 语句块的编写**；朱莞尔负责程序一中 **main 函数主体及外层 for 循环部分语句块编写**以及程序二中 **main 函数主体及外层 for 循环部分语句块编写**。

4. 共同完成文档编写、总结实验过程中遇到的困难和心得体会。

三、 实验过程

(一) 编译器所支持的 SysY 语言特性

2020 版 SysY 语言支持 int、void 数据类型，2022 版增加了对 float 类型数据的支持；支持变量声明；赋值语句、复合语句、if 分支语句、while/for 循环语句；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)；支持函数、数组指针等等。

(二) SysY 语言的形式化定义

·SysY 语言摘要

SysY 语言是编译系统设计赛要实现的编程语言。由 C 语言的一个子集扩展而成。每个 SysY 程序的源码存储在一个扩展名为 sy 的文件中。该文件中有且仅有一个名为 main 的主函数定义，还可以包含若干全局变量声明、常量声明和其他函数定义。SysY 语言支持 int/float 类型和元素为 int/float 类型且按行优先存储的多维数组类型，其中 int 型整数为 32 位有符号数；float 为 32 位单精度浮点数；const 修饰符用于声明常量。SysY 支持 int 和 float 之间的隐式类型，但是无显式的强制类型转化支持。

·上下文无关文法

上下文无关文法是一种用于描述完整有效的程序设计语言语法的表示方式。它涵盖四个组成部分：终结符号集、非终结符号集、产生式集合，以及开始符号。下面的部分使用上下文无关文法描述了我们需要实现的编译器所支持的 SysY 语言特性。

1. 编译单元

```
CompUnit → CompUnit SingleUnit | SingleUnit
SingleUnit → Decl | FuncDef
```

1. 一个 SysY 程序由单个文件组成，文件内容对应 EBNF 表示中的 CompUnit。在该 CompUnit 中，必须存在且仅存在一个标识为‘main’、无参数、返回类型为 int 的 FuncDef(函数定义)。main 函数是程序的入口点，main 函数的返回结果需要输出。

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

3. 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。

2. 声明

```
Decl → ConstDecl | VarDecl
```

3. 基本类型

```
BType → 'int' | 'float'
```

4. 常量

```
ConstDecl → 'const' BType ConstsDefs ';'
ConstsDefs → ConstDef | ConstsDefs ',' ConstDef
ConstDef → Ident ConstExps '=' ConstInitVal
ConstsExps → '[' ConstExp ']' | ConstsExps '[' ConstExp ']' | epsilon
ConstInitVal → ConstExp | '{' ConstInitVals '}'
ConstInitVals → epsilon | ConstInitVal | ConstInitVals ',' ConstInitVal
```

1. ConstDef 用于定义符号常量。ConstDef 中的 Ident 为常量的标识符，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。

2. 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。

3. 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。其语义和 C 语言一致，比如 [2][8/2][1*3] 表示三维数组，第一到第三维长度分别为 2、4、3，每维的下界从 0 编号。ConstDef 中表示各维长度的 ConstExp 都必须能在编译时求值到非负整数。

注意：SysY 在声明数组时各维长度都需要显式给出，而不允许是未知的。

4. 当 ConstDef 定义的是数组时，‘=’右边 d ConstInitVal 表示常量初始化器。ConstInitVal 中的 ConstExp 是能在编译时求值的 int/float 型表达式，其中可以引用已定义的符号常量。

5. ConstInitVal 初始化器必须是以下三种情况之一：

a) 一对花括号，表示所有元素初始为 0。

b) 与多维数组中数组维数和各维长度完全对应的初始值。

c) 如果花括号括起来的列表中的初始值少于数组中对应维的元素个数，则该维其余部分将被隐式初始化，需要被隐式初始化的整型元素均初始为 0。

5. 变量

```
VarDecl → BType VarDefs ';'
VarDefs → VarDef | VarsDefs ',' VarDef
VarDef → Ident ConstsExps | VarDefValue
VarDefValue → Ident ConnstsExps '=' InitVal
InitVal → Exp | '{' InitVals '}'
InitVals → epsilon | InitVal | InitVals ',' InitVal
```

1. 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。

2. 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，表示定义多维数组。（参见 ConstDef 的第 2 点）

3. 当 VarDef 含有‘=’和初始值时，‘=’右边的 InitVal 和 CostInitVal 的结构要求相同，唯一的不同是 ConstInitVal 中的表达式是 ConstExp 常量表达式，而 InitVal 中的表达式可以是当前上下文合法的任何 Exp。

4. VarDef 中表示各维长度的 ConstExp 必须是能求值到非负整数，但 InitVal 中的初始值为 Exp，其中可以引用变量.

5. 全局变量声明中指定的初值表达式必须是常量表达式。

6. 常量或变量声明中指定的初值要与该常量或变量的类型一致.

7. 未显式初始化的局部变量，其值是不确定的；而未显式初始化的全局变量，其（元素）值均被初始化为 0 或 0.0。

6. 函数

```
FuncType → 'int' | 'float' | 'void'
FuncDef → FuncType Ident '(' FuncFParams ')' Block
FuncFParams → epsilon | FuncFParam | FuncFParams ',' FuncFParam
FuncFParam → BType Ident Express
Express → epsilon | '[' ']' Exps
Exps → epsilon | '[' Exp ']' | Exps '[' Exp ']'
```

1. FuncFParam 定义一个函数的一个形式参数。当 Ident 后面的可选部分存在时，表示数组定义。

2. 当 FuncFParam 为数组定义时，其第一维的长度省去（用方括号 [ ] 表示），而后面的各维则需要用表达式指明长度，长度是整型常量。

3. 函数实参的语法是 Exp。对于 int/float 类型的参数，遵循按值传递；对于数组类型的参数，则形参接收的是实参数组的地址，并通过地址间接访问实参数组中的元素。

4. 对于多维数组，可以传递其中的一部分到形参数组中。

5. FuncDef 表示函数定义。其中的 FuncType 指明返回类型。

a) 当返回类型为 int/float 时，函数内所有分支都应当含有带有 Exp 的 return 语句。不含有 return 语句的分支的返回值未定义。

b) 当返回值类型为 void 时，函数内只能出现不带返回值的 return 语句。

6. FuncDef 中形参列表（FuncFParams）的每个形参声明（FuncFParam）用于声明 int/float 类型的参数，或者是元素类型为 int/float 的多维数组。FuncFParam 的语义参见前文。

7. 语句

```
Block → '{' BlockItems '}'
BlockItems → epsilon | BlockItem BlockItems
BlockItem → Decl | Stmt
Stmt → LVal '=' Exp ';'
        | Exp ';'
        | ';'
        | Block
        | 'if' '(' Cond ')' Stmt ElseStmt
        | 'while' '(' Cond ')' Stmt
        | 'break' ';'
        | 'continue' ';'
        | 'return' returnvalue ';'
ElseStmt → epsilon | 'else' Stmt
returnvalue → epsilon | Exp
```

- Stmt

1. Stmt 中的 if 类型语句遵循就近匹配。

2. 单个 Exp 可以作为 Stmt。Exp 会被求值，所求的值会被丢弃。

- LVal

1. LVal 表示具有左值的表达式，可以为变量或者某个数组元素。

2. 当 LVal 表示数组时，方括号个数必须和数组变量的维数相同（即定位到元素）。

3. 当 LVal 表示单个变量时，不能出现后面的方括号

- Exp 与 Cond

1. Exp 在 SysY 中代表 int 型表达式，故它定义为 AddExp；Cond 代表条件表达式，故它定义为 LOrExp。前者的单目运算符中不出现’!’，后者可以出现。

2. LVal 必须是当前作用域内、该 Exp 语句之前有定义的变量或常量；对于赋值号左边的 LVal 必须是变量。

3. 函数调用形式是 Ident ‘(’ FuncRParams ‘)’，其中的 FuncRParams 表示实际参数。实际参数的类型和个数必须与 Ident 对应的函数定义的形参完全匹配。

4. SysY 中算符的优先级与结合性与 C 语言一致，在上面的 SysY 文法中已体现出优先级结合性的定义。

8. 表达式

```
算数表达式: Exp -> AddExp
常量表达式: ConstExp -> AddExp （使用的所有 ident 必须为常量时）
加法运算符: AddOp -> '+ ' | '- '
乘法运算符: MulOp -> '* ' | '/ ' | '%'
AddExp -> MulExp | AddExp AddOp MulExp
MulExp -> UnaryExp | MulExp MulOp UnaryExp
一元表达式: UnaryExp -> PrimaryExp | UnaryOp UnaryExp | ident ' ( ) '
        | ident '( ' FuncRParams ') '
单目运算符: UnaryOp -> '+ ' | '- ' | ' ! '
函数实参表: FuncRParams -> Exp | FuncRParams ' , ' Exp
基本表达式: PrimaryExp -> '( ' Exp ') ' | LVal | int-const
左值表达式: LVal -> ident | ident ' [ ' Exp ' ] '
ConstLVal -> ident | ident ' [ ' ConstExp ' ] '
条件表达式: Cond -> LOrExp
逻辑或表达式: LOrExp -> LAndExp | LOrExp ' | | ' LAndExp
逻辑与表达式: LAndExp -> EqExp | LAndExp '&&' EqExp
相等性表达式: EqExp -> RelExp | EqExp '==' RelExp | EqExp '!= ' RelExp
关系表达式: RelExp -> RelExp '<' AddExp | RelExp '>' AddExp
        | RelExp '<=' AddExp | RelExp '>=' AddExp | AddExp
```

9. 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 ' | '_'
identifier - digit -> ' 0 ' | ' 1 ' | ' 2 ' | ' 3 ' | ' 4 ' | ' 5 ' | ' 6 '
        | ' 7 ' | ' 8 ' | ' 9 '
```

10. 常量

```
整型: integer -const -> decimal-const | octal-const | hexadecimal-const
十进制: decimal-const -> nonzero-digit | decimal-const digit
八进制: octal-const -> 0 | octal-const octal-digit
十六进制: 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'
```

(三) 自编 arm 语言汇编程序

接下来是本次实验第二部分的具体报告。在这一部分中我们将汇报自编程序以及转换所成的 arm 程序。[3]

1. SysY 语言特性

本次实验中我们共设计了两个程序，均基于下表中特性撰写。

1) 数据类型：int
2) 变量声明、常量声明，常量、变量的初始化
3) 语句：赋值（=）、表达式语句、语句块、if、while、return
4) 表达式：算术运算（+、-、、、%，其中 +、-都可以是单目运算符）、关系运算（==，>，<，>=，<=，!=）和逻辑运算（&&（与）、||（或）、!（非））
5) 注释
6) 输入输出（实现连接 SysY 运行时库，参见文档《SysY 运行时库》）

表 1: SysY 语言特性

2. SysY 程序设计示例一

·C 语言程序背景

如果一个数恰好等于它的真因子之和，则称该数为“完全数”，该代码完成了 1000 以内的完全数查找。

·C 语言代码

```c
#include<stdio.h>
int main() {
    int i, j;
    int sum;
    for (i = 1; i <= 1000; i++) {
        sum = 0;
        for (j = 1; j < i; j++)
            if (i % j == 0) {
                sum += j;
            }
        if (sum == i) {
            printf("%d\n", i);
        }
    }
```

```
}
```

·分工详情

程序设计完毕后，经过讨论，汤清云负责**内层 for 循环**编写；朱莞尔负责 **main 函数主体及外层 for 循环部分语句块编写**。

·ARM 程序编写结果

完全数 arm 程序

```
        .arch armv7-a
        .file   "exp2.c"
        .text
        .section        .rodata
        .align  2
.LC0:
        .ascii  "%d\012\000"
        .text
        .align  1
        .global main
        .syntax unified
        .thumb
        .thumb_func
main:
        push    {r4, lr}
        sub     sp, sp, #20                     @栈帧抬20
        add     r4, sp, #0                              @记录一下sp
        mov     r5, #1                                  @i=1
        str     r5, [r4, #4]                            @存入i
        b       .LC7                            @跳转到外层循环
.LC1:                                           @外层for循环内部执行体
        mov     r7, #0                          @sum = 0
        str     r7, [r4, #12]                           @存进sum
        mov     r6, #1                          @j = 1
        str     r6, [r4, #8]                            @存进j
        b       .LC4                            @跳转到内层循环
.LC2:
        ldr     r3, [r4, #4]                            @取出i 此处需要调用函数，故使
            用r3
        ldr     r1, [r4, #8]                            @取出j
        mov     r0, r3
        bl      __aeabi_idivmod(PLT)            @除法函数
        mov     r3, r1                                  @
        cmp     r3, #0                          @判断i % j是否为0
        bne     .LC3                            @不相等则跳转到LC3
        ldr     r6, [r4, #12]                           @r6=j
        ldr     r7, [r4, #8]                            @r7=sum
        add     r7, r7, r6                      @sum += j
        str     r7, [r4, #12]                           @存回sum
.LC3:
```

```
        ldr     r7, [r4, #8]                    @取出j
        add     r7, r7, #1              @j++
        str     r7, [r4, #8]
.LC4:
        ldr     r6, [r4, #8]                    @取出j
        ldr     r5, [r4, #4]                    @取出i
        cmp     r6, r5                  @比较i与j的大小
        blt     .LC2                     @小于则跳转到LC2进入if
        ldr     r7, [r4, #12]                   @取出sum
        ldr     r5, [r4, #4]                    @取出i
        cmp     r7, r5                  @比较sum与i的大小
        bne     .LC6                     @不相等则跳转到LC6
        ldr     r1, [r4, #4]
        ldr     r3, .LC9
.LC5:
        add     r3, pc
        mov     r0, r3
        bl      printf(PLT)             @输出sum
.LC6:
        adds    r5, r5, #1              @i++
        str     r5, [r4, #4]
.LC7:@外层循环判断条件
        ldr     r5, [r4, #4]                    @更新了i值，所以取出
        cmp     r5, #1000               @将i与1000比较
        ble     .LC1                     @小于则跳转到LC1
.LC8:@结束循环 程序结束
        add     r4, r4, #20                     @恢复一下栈帧
        mov     sp, r4
        pop     {r4, pc}                @return 0
.LC9:
        .word   .LC0-(.LC5+4)
```

·程序编译执行结果

```
root@LAPTOP-5B7ES80G:/home/exp2# arm-linux-gnueabihf-gcc arm_exp2.S -o test2 -static
root@LAPTOP-5B7ES80G:/home/exp2# ./test2
6
28
496
root@LAPTOP-5B7ES80G:/home/exp2# arm-linux-gnueabihf-gcc arm_exp2.S -o test2 -static
root@LAPTOP-5B7ES80G:/home/exp2# ./test2
6
28
496
```

图 1: 完全数运行结果

3. SysY 程序设计示例二

·C 语言程序介绍

C 语言程序背景：找出 0499 之间三位数字各不相同的三位数，以及它们的总个数。

·C 程序代码

```
#include<stdio.h>
int main()
{
        int sum;//变量声明
        sum=0;//变量初始化
        const int one=1;//常量声明与初始化
        int temp;
        int ans=scanf("%d",&temp);
        for(int i=0;i<5;i++)
        {
                for(int a=0;a<9;a++)
                {
                        int b=0;
                        while(b<9)
                        {
                                if(i!=a&&i!=b&&a!=b)
                                {
                                        printf("%d%d%d",i,a,b);
                                        putchar('\n');
                                        sum=sum+one*temp;
                                }
                                b=b+1;
                        }
                }
        }
        printf("%d",sum);
        return 0;
}
```

·分工详情

程序设计完毕后，经过讨论，汤清云负责**内层 for 循环**以及 **while、if 语句块的编写**；朱莞尔负责 **main 函数主体及外层 for 循环部分语句块编写**。

·ARM 程序编写

```
        @程序信息
    .arch armv7-a
        .file   "arm_exp.c"
        .text
    @常量声明
        .section        .rodata
        .align  2
.LC0:
        .ascii  "%d\000"    @ \000字符串结束
        .align  2
.LC1:
        .ascii  "%d%d%d\000"    @ \000字符串结束
```

```
    @代码段
        .text
        .align  1
        .global main
        .syntax unified
        .thumb
        .thumb_func
main:
    @栈帧调整
        push    {r4, r7, lr}
        sub     sp, sp, #36
        add     r7, sp, #0
    ldr r4, .LC14
.LC2:
        add     r4, pc                  @记录pc
        ldr     r3, .LC14+4              @r3=&sum
        ldr     r3, [r4, r3]
        ldr     r3, [r3]                @r3=sum
        str     r3, [r7, #28]
        mov     r3, #0                  @sum = 0
        str     r3, [r7, #4]
        mov     r3, #1                  @one = 1
        str     r3, [r7, #20]
        mov     r3, r7
        mov     r1, r3
        ldr     r3, .LC14+8
.LC3:
            add     r3, pc
        mov     r0, r3
        bl      __isoc99_scanf(PLT)     @输入temp的值
        str     r0, [r7, #24]
        mov     r3, #0                  @i = 0
        str     r3, [r7, #8]
        b       .LC11
.LC4:
        movs    r3, #0                  @a = 0
        str     r3, [r7, #12]
        b       .LC10
.LC5:
        movs    r3, #0                  @b = 0
        str     r3, [r7, #16]
        b       .LC9                    @跳转到while循环
.LC6:
        ldr     r2, [r7, #8]
        ldr     r3, [r7, #12]
        cmp     r2, r3                  @i与a比较
        beq     .LC8                    @相等则跳转到LC8
        ldr     r2, [r7, #8]
```

```
61          ldr     r3, [r7, #16]
62          cmp     r2, r3                  @i与b比较
63          beq     .LC8                    @相等则跳转到LC8
64          ldr     r2, [r7, #12]
65          ldr     r3, [r7, #16]
66          cmp     r2, r3                  @a与b比较
67          beq     .LC8                    @相等则跳转到LC8
68          ldr     r3, [r7, #16]
69          ldr     r2, [r7, #12]
70          ldr     r1, [r7, #8]
71          ldr     r0, .LC14+12
72  .LC7:
73          add     r0, pc
74          bl      printf(PLT)             @输出i, a, b
75          mov     r0, #10
76          bl      putchar(PLT)            @换行
77          ldr     r3, [r7]
78          ldr     r2, [r7, #20]
79          mul     r3, r2, r3              @one * temp
80          ldr     r2, [r7, #4]
81          add     r3, r3, r2              @sum = sum + one * temp
82          str     r3, [r7, #4]
83  .LC8:
84          ldr     r3, [r7, #16]
85          add     r3, r3, #1              @b = b + 1
86          str     r3, [r7, #16]
87  .LC9:
88          ldr     r3, [r7, #16]
89          cmp     r3, #8                  @b与8比较
90          ble     .LC6                    @b小于等于8则跳转到LC6
91          ldr     r3, [r7, #12]
92          add     r3, r3, #1              @b++
93          str     r3, [r7, #12]
94  .LC10:
95          ldr     r3, [r7, #12]
96          cmp     r3, #8                  @将a与8比较
97          ble     .LC5                    @a小于等于8则跳转到LC5
98          ldr     r3, [r7, #8]
99          add     r3, r3, #1              @a++
100         str     r3, [r7, #8]
101 .LC11:
102         ldr     r3, [r7, #8]
103         cmp     r3, #4                  @将i与4比较
104         ble     .LC4                    @i小于等于4则跳转到LC4
105         ldr     r1, [r7, #4]
106         ldr     r3, .LC14+16
107 .LC12:
108         add     r3, pc
```

```
109         mov     r0, r3
110         bl      printf(PLT)                 @输出sum
111         mov     r3, #0
112         mov     r0, r3
113         ldr     r3, .LC14+4
114         ldr     r3, [r4, r3]
115         ldr     r2, [r7, #28]
116         ldr     r3, [r3]
117         cmp     r2, r3
118         beq     .LC13
119 .LC13:
120         add     r7, r7, #36         @r7到旧栈顶
121         mov     sp, r7              @恢复sp
122         pop     {r4, r7,pc}                 @return 0
123         .align  2
124 .LC14:
125         .word   _GLOBAL_OFFSET_TABLE_-(.LC2+4)
126         .word   __stack_chk_guard(GOT)
127         .word   .LC0-(.LC3+4)
128         .word   .LC1-(.LC7+4)
129         .word   .LC0-(.LC12+4)
```

·程序编译执行结果

```
root@LAPTOP-5B7ES80G:/home/exp2# arm-linux-gnueabihf-gcc arm_exp.S -o test -static
root@LAPTOP-5B7ES80G:/home/exp2# ./test
1
012
013
014
015
016
017
018
021
023
024
025
026
027
028
```

图 2: 相符的三位数

四、 总结与思考

(一) 实验框架构造

在编写 ARM 实验中，我们遇到了许多困难。在学习完毕给出的资料后，开始进行编译前，我们选择使用逐步编译的方式来熟悉编译进程，比如首先构造最基本的框架：

简单代码-1

```
1 #include<studio.h>
2 int main()
```

```
{
    return 0;
}
```

接下来使用机器编译器对其编译，观察所需代码：

```
        .arch armv7-a
        .eabi_attribute 28, 1
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 2
        .eabi_attribute 30, 6
        .eabi_attribute 34, 1
        .eabi_attribute 18, 4
        .file   "exp1.c"
        .text
        .align  1
        .global main
        .syntax unified
        .thumb
        .thumb_func
        .fpu vfpv3-d16
        .type   main, %function
main:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 1, uses_anonymous_args = 0
        @ link register save eliminated.
        push    {r7}
        add     r7, sp, #0
        movs    r3, #0
        mov     r0, r3
        mov     sp, r7
        @ sp needed
        ldr     r7, [sp], #4
        bx      lr
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",%progbits
```

1. 可以看出在最基本的程序框架中，会有许多 eabi_attribute 的生成，然而经过实验和查阅相关资料 [5]，这些代码并没有任何含义，因此我们可以将它们删去。

2.main 函数部分最重要的是开辟栈帧，# 代表立即数，@ 后的内容均为注释，可以自行删去；程序默认使用寄存器 r7 存储栈帧位置。

3. 经过实践，size 及其之后部分可以删除，不影响程序正常运行。

再添加元素如下

```c
#include<stdio.h>
int main()
{
        int i,j;
        int sum;
        for(i=1;i<=1000;i++)
        {
        }
}
```

继续观察代码:

```
        .arch armv7-a
        .eabi_attribute 28, 1
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 2
        .eabi_attribute 30, 6
        .eabi_attribute 34, 1
        .eabi_attribute 18, 4
        .file   "exp2.c"
        .text
        .align  1
        .global main
        .syntax unified
        .thumb
        .thumb_func
        .fpu vfpv3-d16
        .type   main, %function
main:
        @ args = 0, pretend = 0, frame = 8
        @ frame_needed = 1, uses_anonymous_args = 0
        @ link register save eliminated.
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        movs    r3, #1
        str     r3, [r7, #4]
        b       .L2
.L3:
        ldr     r3, [r7, #4]
        adds    r3, r3, #1
        str     r3, [r7, #4]
.L2:
        ldr     r3, [r7, #4]
        cmp     r3, #1000
```

```
38          ble     .L3
39          movs    r3, #0
40          mov     r0, r3
41          adds    r7, r7, #12
42          mov     sp, r7
43          @ sp needed
44          ldr     r7, [sp], #4
45          bx      lr
46          .size   main, .-main
47          .ident  "GCC: (Ubuntu/Linaro 7.5.0-3ubuntu1~18.04) 7.5.0"
48          .section        .note.GNU-stack,"",%progbits
```

4. 以句点开头的指令实际上不是指令，而是指令; 它们是对汇编程序的提示，而不是 CPU 指令集的一部分。[1]

5. 在系统自动生成的情况下一般只使用寄存器 r3 完成各类取内容操作，在非特殊情况下不扩展寄存器使用数，节约空间 [4]。

在完成以上对代码的研究分析以及对 ARM 语言的研究后，我们就可以开始试着编写自己的 ARM 程序了。

本次实验中，在程序一中我们取用多个寄存器来存储变量以熟悉指令的使用；而在程序二中我们模仿了机器自动生成的代码，多次调用同一个寄存器 r3 以实现对不同变量的存取，减少了寄存器的使用，也加强了自身使用 ARM 语言编写的能力。

(二) ARM 学习总结

1. 关于寄存器

结合所给资料 [2]，我们了解到 arm 除却基于 ARMv6-M 和 ARMv7-M 的处理器外，有 30 个通用的 32 位寄存器。前 16 个寄存器可在用户级模式下访问;

R0-R12：在普通操作中可用于 store 临时值、指针 (store 器的位置) 等。例如，R0 在进行算术运算时可作为累加器，或用于 store 先前调用的函数的结果。R7 在处理系统调用时变得非常有用，因为它 store 了系统调用的编号，R11 帮助我们跟踪堆栈上的边界，作为框架指针 (将在后面介绍)。此外，ARM 的函数调用惯例规定，函数的前四个参数 store 在寄存器 r0-r3 中。

R13：SP(堆栈指针)。堆栈指针指向堆栈的顶部。堆栈是一个用于特定函数 store 的内存区域，在函数返回时被回收。因此，堆栈指针用于分配堆栈的空间，方法是用堆栈指针减去我们要分配的值 (以字节为单位)。换句话说，如果我们想分配一个 32 位的值，我们从堆栈指针中减去 4。(因为是以字节为单位，所以只要减去 4 即可)

R14：LR(链接寄存器)。当一个函数被调用时，链接寄存器被更新为内存地址，引用函数启动的下一条指令。这样做允许程序在” 子” 函数完成后返回到启动” 子” 函数的” 父” 函数。

R15：PC(程序计数器)。程序计数器根据所执行的指令的大小自动递增。这个大小在 ARM 状态下总是 4 字节，在 THUMB 模式下是 2 字节。当一个分支指令被执行时，PC 保存目标地址。在执行过程中，PC 在 ARM 状态下 store 当前指令的地址加 8(两条 ARM 指令)，在 Thumb(v1) 状态下 store 当前指令加 4(两条 Thumb 指令)。这与 x86 不同，x86 的 PC 总是指向下一条要执行的指令。

2. 关于 Thumb

ARM 版本的调用惯例比较混乱，而且并非所有的 ARM 版本都支持相同的 Thumb 指令集。其大致分为以下几种：

Thumb-1(16 位指令)：在 ARMv6 和早期体系结构中使用。

Thumb-2(16 位和 32 位指令)：通过增加更多的指令并允许它们的宽度为 16 位或 32 位 (ARMv6T2、ARMv7)，扩展了 Thumb-1。

ThumbEE：包括一些针对动态生成的代码 (在执行前不久或执行期间在设备上编译的代码) 的更改和增加。

3. 关于堆栈

一般来说，堆栈是程序/进程中的一个内存区域。[1] 当一个进程被创建时，这部分内存被分配。我们使用堆栈来存储临时数据，如一些函数的局部变量，环境变量，这有助于我们在函数之间的转换，等等。我们使用 PUSH 和 POP 指令与堆栈进行交互。

4. 关于条件执行和分支

1.ARM 使用条件来控制程序在运行时的流，通常是通过进行跳转（分支）或仅在满足条件时才执行某些指令。该条件被描述为 CPSR 寄存器中特定位的状态。这些位会根据某些指令的结果不时更改。例如，当我们比较两个数字并且它们被证明是相等的时，我们触发零位（$Z = 1$）。

2. 分支也可以有条件地执行，并在满足特定条件时用于分支到函数。让我们看一个非常简单的条件分支起诉 BEQ 的例子。这个程序集除了将值移动到寄存器中并在寄存器等于指定值时分支到另一个函数之外，不触发其他事件。

参考文献

[1] Arm 组装的旋风之旅. https://www.coranac.com/tonc/text/asm.htm.

[2] Arm 指令集相关内容. https://azeria-labs.com/writing-arm-assembly-part-1/.

[3] 所编写 arm 程序的 gitlab 链接. https://gitlab.eduxiji.net/wanerandtang.

[4] 树莓派中的 arm 汇编程序. https://thinkingeek.com/2013/01/10/arm-assembler-raspberry-pi-chapter-2/.

[5] 汇编程序用户指南. https://developer.arm.com/documentation/dui0473/m/overview-of-the-assembler/directives-that-can-be-omitted-in-pass-2-of-the-assembler.