南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理实验报告

预备工作二

姓名：张晏睿 莫骐骥

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

这篇报告介绍了我们小组的任务分工，计划要实现的编译器所要实现的 SysY 语言特性和其中一些语法特性的汇编代码实现，以及思考题的思路。在第一节中，介绍了本次任务的小组分工情况：张晏睿主要负责上下文无关文法设计中变量 & 常量 & 表达式 & 函数的部分，以及逻辑运算、if 分支、while 循环、函数的 ARM 代码编写，莫骐骥主要负责上下文无关文法设计中标识符 & 注释 & 语句块的部分，以及四则运算、位运算、数组、指针、for 循环 ARM 代码的编写，思考题的思路由二人讨论总结得出。之后在第二小节，用上下文无关文法对要实现的 SysY 语言特性给出了形式化描述。在第三小节中，给出了若干段简单的 C 程序代码，每段代码对应一个或多个语言特性，并在 C 代码的下方给出由自己尝试的与之对应的 ARM 汇编核心代码。在最后的第四小节中，介绍了将 C 代码翻译为中间代码的编译器程序的框架做出大致构思，针对编译的每一个中间过程的数据结构和算法给出我们的思路介绍。

关键字：SysY 语言特性　ARM 汇编　上下文无关文法　编译器

目录

一、 分工说明 **1**

二、 定义编译器 **2**
（一） 标识符 . . . 2
（二） 变量 & 常量 . . . 2
1. 变量声明 . . . 2
2. 变量初始化和定义 . . . 3
3. 常量声明定义及初始化 . . . 3
4. 数值常量 . . . 4
（三） 表达式 & 语句 . . . 5
1. 表达式 . . . 5
2. 语句、语句块 . . . 7
（四） 注释 . . . 7
（五） 函数 . . . 8

三、 汇编编程 **9**
（一） 运算 . . . 9
1. 四则运算 . . . 9
2. 位运算 . . . 10
3. 逻辑运算 & 条件运算 . . . 10
（二） 数组 & 指针 . . . 12
1. 指针 . . . 12
2. 数组 &for 循环 . . . 13
（三） 分支 & 循环 . . . 14
1. 跳转指令 . . . 14
2. if . . . 14
3. while&do while . . . 14
（四） 函数 . . . 15
1. 栈的使用 . . . 15
2. 库函数 . . . 15
3. 自定义函数 . . . 16
4. 栈传递参数 . . . 16

四、 思考题 **17**
（一） 设计思路 . . . 17
（二） 词法分析 . . . 17
（三） 语法分析 . . . 17
（四） 语义分析 . . . 18
（五） 中间代码生成及优化 . . . 18

一、 分工说明

任务	张晏睿	莫骐骥
CFG	变量、常量	标识符
	表达式	注释
	函数	语句块
ARM 汇编	逻辑运算	整数、浮点四则运算
	if 分支、while 循环	for 循环
	函数	数组、指针
思考题	共同总结	

表 1: 小组分工说明表

NKU

二、 定义编译器

了解你的编译器所支持的 sysY 语言特性，从中选取你要实现的部分定义为你编译器功能，使用上下文无关文法描述你所选取的 sysY 语言子集.

在本章节中，将基于将要实现编译器的 sysY 语言特性，给出其上下文无关文法的定义与描述。

(一) 标识符

标识符是程序中基本的组成单元，首先定义：

@非数字标识符字符

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

@数字

$$digit \longrightarrow 0|1|2|3|4|5|6|7|8|9$$

@标识符

$$id \longrightarrow id\text{-}nodigit \mid id\ \ id\text{-}nodigit \mid id\ \ digit$$

标识符可以由非数字字符开头，或由短标识符和数字、字母字符拼接而成，采用递归定义方式

(二) 变量 & 常量

1. 变量声明

@标识符列表

$$idlist \longrightarrow idlist,\ id \mid id$$

@基本数据类型

$$BType \longrightarrow int \mid float$$

@变量声明

$$VarDecl \longrightarrow BTypeidlist$$

- 声明 id 的列表采用右递归，便于使用超前预测法定义符号位置；
- 本次实现的编译器支持 int 和 float 两种基本数据类型

2. 变量初始化和定义

@变量初始化

$$InitVal \longrightarrow Exp \mid \{Initval\text{-}list\}$$

@变量列表

$$InitVal\text{-}list \longrightarrow InitVal \mid InitVal\text{-}listInitVal$$

@变量定义

$$VarDef \longrightarrow id \mid VarDef\ [\ ConstExp\ ] \mid VarDef = InitVal$$

- 变量初始化中，使用了变量列表，这是为了支持数组的初始化，例如 int a[3]={1,2,3};
- 变量的定义可以是单个 id、id 加中括号形成数组如 a[2]、或是带有初值的初始化定义

3. 常量声明定义及初始化

@常量声明

$$ConstDecl \longrightarrow const\ \ BType\ \ ConstDef \mid ConstDecl\ \ ConstDef$$

@常量定义

$$ConstDef \longrightarrow id = ConstInitVal \mid ConstExp = ConstInitVal$$

@常量初始化

$$ConstInitVal \longrightarrow ConstExp$$

@常量表达式

$$ConstExp \longrightarrow AddExp$$

@声明

$$Decl \longrightarrow ConstDecl \mid VarDecl$$

- 常量的声明、定义和初始化类似于变量，只不过它们属于不同的分类;
- 常量初始化使用了表达式，表达式将在后面定义
- 声明包括常量声明和变量声明

4. 数值常量

@整型常量

$$IntConst \longrightarrow decimalConst \mid octalConst \mid hexadecimalConst$$

@十进制

$$decimalConst \longrightarrow nonzero\text{-}digit \mid decimalConstdigit$$

@八进制

$$octalConst \longrightarrow \mathbf{0} \mid octalConstoctal\text{-}digit$$

@十六进制

$$hexadecimalConst \longrightarrow hexadecimal\text{-}prefixhexadecimal\text{-}digit \mid hexadecimalCon$$

$$hexadecimal\text{-}digit$$

@非零十进制数

$$nonzero\text{-}digit \longrightarrow \mathbf{1|2|3|4|5|6|7|8|9}$$

@八进制数

$$octal\text{-}digit \longrightarrow \mathbf{0|1|2|3|4|5|6|7}$$

@十六进制数

$$hexadecimal\text{-}digit \longrightarrow \mathbf{0|1|2|3|4|5|6|7|8|9|a|b|c|d|e|f|A|B|C|D|E|F}$$

@十六进制前缀

$$hexadecimal\text{-}prefix \longrightarrow \mathbf{'0x' \mid '0X'}$$

@浮点常量

$$FloatConst \longrightarrow digits\ '.'\ digits$$

@数字

$$digits \longrightarrow digit \mid digits\ digit$$

- 根据不同进制，使用不同的数字组合，来定义整型常量
- 浮点数由数字和小数点组成

(三) 表达式 & 语句

1. 表达式

@表达式

$$Exp \longrightarrow \quad | \ AddExp$$

@加减表达式

$$AddExp \longrightarrow MulExp \mid AddExp + MulExp \mid AddExp - MulExp$$

@乘除模表达式

$$MulExp \longrightarrow UnaryExp \mid MulExp * UnaryExp \mid MulExp / UnaryExp \mid MulE$$

$$\% \ UnaryExp$$

@基本表达式

$$PrimaryExp \longrightarrow (\ Exp\ ) \mid LVal \mid Number$$

$$LVal = id \mid LVal\ [\ Exp\ ]$$

$$Number \longrightarrow IntConst \mid floatConst$$

@一元表达式

$$UnaryOp \longrightarrow \ + \mid - \mid !$$

$$UnaryExp \longrightarrow PrimaryExp \mid id\ (\ FuncRParams\ ) \mid UnaryOp\ \ UnaryExp$$

@关系表达式

$$RelExp \longrightarrow AddExp \mid RelExp < AddExp \mid RelExp > AddExp$$

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

@相等性表达式

$$EqExp \longrightarrow RelExp \mid EqExp == RelExp \mid EqExp != RelExp$$

@逻辑与表达式

$$LAndExp \longrightarrow EqExp \mid LAndExp \ \&\&EqExp$$

@逻辑或表达式

$$LOrExp \longrightarrow LAndExp \mid LOrExp \mid\mid LAndExp$$

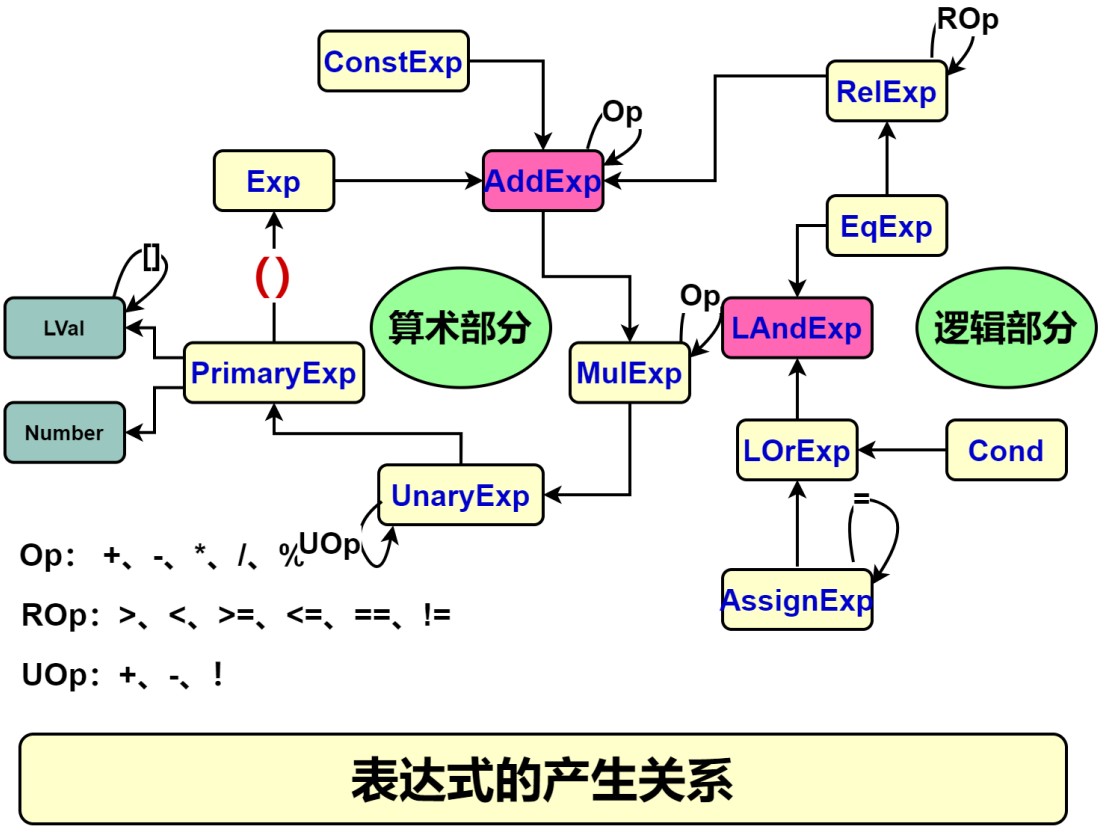
@条件表达式

$$Cond \longrightarrow LOrExp$$

@赋值表达式

$$AssignExp \longrightarrow id = AssignExp \mid LOrExp$$

使用图片直观表示不同表达式的联系:



- 算术部分有一个环，但是并非会无限递归，因为从 PrimaryExp 到 Exp 加括号改变了优先级
- 其他部分都是线性关系，有些 Exp 能够结合一些运算符递归定义

2. 语句、语句块

@语句

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

@语句块项

$$BlockItem \longrightarrow Decl \mid Stmt$$

@语句块

$$Block \longrightarrow \{\ \} \mid \{\ BlockItem\ \}$$

- 语句包括：左值赋值语句、if 语句、for 语句、while 语句、块语句、continue、break、return 语句、和其他表达式
- 语句块由语句块项和大括号组成，块项包括声明和语句

(四) 注释

@注释

$$Comments \longrightarrow LineComments \mid LinesComments$$

@单行注释

$$LineComments \longrightarrow /\ /\ str\ '\backslash n'$$

@多行注释

$$LinesComments \longrightarrow /*\ str\ */ \mid /*\ str\ '\backslash n'\ */$$

@注释内容

$$
\begin{aligned}
str \longrightarrow\ & \\
& \mid str\ \ id && \mid str\ \ number \\
& \mid str\ \ 0\ \ id && \mid str\ \ '\ ' \\
& \mid str\ \ ';' && \ldots (str + \text{各标点符号和空白符})
\end{aligned}
$$

- 注释分为单行注释和多行注释

- 单行注释由双斜线开始、换行符结束，多行注释以/* 开始，以 */结束
- 注释的字符串是任意的，包括标识符、数字、各种标点组成的串

（五） 函数

@函数类型

$$FuncType \longrightarrow \boldsymbol{void} \mid \boldsymbol{int} \mid \boldsymbol{float}$$

@函数定义

$$FuncDef \longrightarrow FuncType\ \ id\ \ (\ FuncFParams\ )\ \ Block$$

@函数参数

$$FuncFParam \longrightarrow BType\ \ id \mid FuncFParam\ \ [\ Exp\ ]$$

@参数列表

$$FuncFParams \longrightarrow FuncParam \mid FuncFParam\ ,\ FuncFParams$$

@编译单元

$$CompUnit \longrightarrow Decl \mid FuncDef \mid CompUnit\ \ Decl \mid CompUnit\ \ FuncDef$$

- 函数支持三种基本类型：空、整型、浮点型
- 函数参数可以是基本数据类型及其数组
- 单个参数构成参数列表，右递归方式定义
- 编译单元由函数和变量的定义、声明组成，采用递归定义

三、 汇编编程

设计几个尽可能全面地包含所支持的语言特性的 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.

(一) 运算

1. 四则运算

四则运算，即加减乘除，是任何一门编程语言中最基本的运算规则。整数和浮点数的四则运算汇编指令略有不同，整数四则运算汇编代码示例如下:

整数的加减乘除四则运算

```
1        movs    r3, #1          @r3 load operand1
2        str     r3, [r7]                @*r7->operand1
3        movs    r3, #2          @r3 load operand2
4        str     r3, [r7, #4]    @*(r7+4)->operand2
5        ldr     r2, [r7]                @*r2 = operand1
6        ldr     r3, [r7, #4]    @*r3 = operand2
7        add     r3, r3, r2              @add op1 op2
8        str     r3, [r7, #8]    @store add_result
9        sub     r3, r2, r3      @sub op1 op2
10       str     r3, [r7, #12]   @store sub_result
11       mul     r3, r2, r3              @mul op1 op2
12       str     r3, [r7, #16]   @store mul_result
13       ldr     r1, [r7, #4]
14       ldr     r0, [r7]
15       bl      __aeabi_idiv    @arm内核不支持除法,使用外部除法库
```

加减乘运算都有直接的指令支持:add、sub、mul,除法指令由于无法在单个 pipeline 中进行,因此没有直接的指令支持,可以通过重复减法或更有效地调用运行时库中的外部函数 __aeabi_idiv 来完成。

浮点数的四则运算汇编代码示例如下:

浮点数的加减乘除四则运算

```
1        mov     r2, #0
2        mov     r3, #0
3        movt    r3, 16400       @1000 0000 0010 000
4        strd    r2, [r7]
5        mov     r2, #0
6        movt    r3, 32768    @0100 0000 0000 0000
7        strd    r2, [r7, #8]
8        vldr.64 d6, [r7]
9        vldr.64 d7, [r7, #8]
10       vadd.f64        d7, d6, d7      @浮点加法
11       vsub.f64        d7, d6, d7      @浮点减法
12       vmul.f64        d7, d6, d7      @浮点乘法
13       vldr.64 d5, [r7]
14       vldr.64 d6, [r7, #8]
```

NKU

```
15          vdiv.f64        d7, d5, d6
```

movt(Move Top) 将立即数写入目标寄存器的高半字。它不会影响底半字的内容。strd(Store Register Dual) 将指定的偶数编号的寄存器以及指定的偶数编号的寄存器的后面紧跟接着后续奇数编号的寄存器的内容写入目标内存位置。但第一个字的地址必须双字对齐；即可以被 8 整除。VLDR 的语法为：$VLDR\{cond\}\{.size\}Fd,[Rn,\#offset]$, 其中.size 是指定数据大小的规范符，Fd 是要加载的扩展寄存器，可以是 D 或 S 寄存器。VADD.f64、vsub.f64、vmul.f64、vdiv.f64 是浮点数的加减乘除指令。浮点数的计算结果存储在 D 存储器中 (d6、d7)。

上面一段代码中两个参与计算的浮点数是 4.0 和 2.0，其在计算机中的浮点存储为：0100 0000 1000 0000 0000 0000 0000 0000 和 01000000000000000000000000000000。为了存储 4.0 和 2.0，需要初始化两个 S 寄存器，4.0 和 2.0 的低 16 位全 0，使用寄存器 r2 存储，寄存器 r3 的高半字先后用 movt 指令存入 4.0 和 2.0 的高 16 位，分别使用 strd 指令写入连续的 64 位内存中。

2. 位运算

C 语言直接提供的位运算符包括按位与“&”、按位或“|”、按位异或“^”、按位非“~”、左移“«”、右移”»“。

位运算汇编代码示例

```
1           mov     r3, #12         @operand1 = 12
2       ……
3           mov     r2, #34         @operand2 = 34
4       ……
5           orr     r3, r2, r3      @orr按位或
6       ……
7           and     r3, r3, r2      @and按位与
8       ……
9           mvn     r3, r3          @mvn将源寄存器每一位的反赋值给目的寄存器对应位
10      ……
11          lsl     r3, r3, #1      @lsl逻辑左移
12      ……
13          asr     r3, r3, #1      @asr算术右移
14      ……
15          eor     r3, r3, r2      @eor按位异或
```

对于左移来说，逻辑左移和算术左移是完全一样的，但是对于右移来说，算数右移的同时需要将符号位不停拷贝到最高位，而逻辑右移则在最高位填充 0 即可。$12_{10} = 00001100_2$,$34_{10} = 00100010_2$. 这两个数按位与的计算结果是 $00000000_2 = 0$, 按位或的计算结果是 $00101110_2 = 46_{10}$, 按位异或的计算结果是 $00101110_2 = 46_{10}$ 00101110_2 按位非的计算结果是 $11010001_2 = -13_{10}$,00001100_2 左移的结果是 $00011100_2 = 24_{10}$, $12_{10} = 00001100_2$ 右移的结果是 $00001110_2 = 6_{10}$。

3. 逻辑运算 & 条件运算

逻辑运算、条件运算主要用于控制程序流，包括：

- 与、或、非等基本指令
- 条件执行指令

NKU

- it 指令

下面分三部分来介绍:

逻辑与 (AND) 示例

```
@AND, 分别判断是否为0, 其中一个为0代表结果为0, 否则结果为1
    ldr r2, [r7, #4]            @从栈中加载第一个操作数到r2
    cmp r2, #0
    beq END                     @r2为0, 结果是0, 运算结束
    ldr r3, [r7, #8]            @第二个操作数r3
    cmp r3, #0
    beq END
    mov r3, #1                  @都不为0, 结果为1
END:
    mov r3, #0                  @至少一个为0, 结果为0
```

OR 也可以仿照 AND 进行编写，思路是类似的，流程如下:

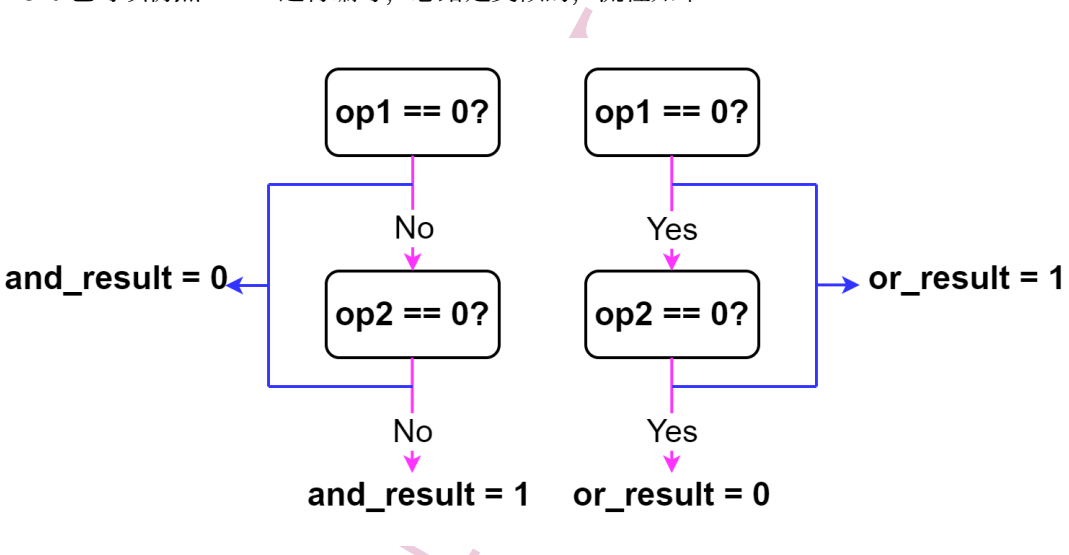


图 1: AND 与 OR 逻辑流图

对于 NOT，直接将操作数与 0 比较即可。

简单指令可以和指令后缀组合成复合指令、条件执行，此类指令优先检查 CPSR(程序状态寄存器) 各位的状态，再决定是否执行，如:

条件执行

```
cmp r0, #3
addlt r0, r0, #1        @只有当r0<3时, r0才自增1
```

条件执行指令常用后缀如下:

IT 指令常用于控制流分支，能简化汇编代码编写过程，如:

IT 指令

```
ite     gt              @ITE——If Then Else, gt——greater than
addgt   r3, #1          @如果大于条件成立, r3+=1
addle   r3, #2          @如果小于等于成立, r3+=2
```

inst-suffix	description	CPSR state
EQ	equal	Z==1
NE	not equal	Z==0
GT	greater than	(Z==0) && (N==V)
LT	less than	N!=V
GE	greater than or equal	N==V
LE	less than or equal	(Z==1) || (N!=V)

表 2: 常用指令后缀

除了 ITE 格式，还有 IT、ITT、ITTE、ITTEE 等格式，需要说明的是 if、then 和 else 的条件必须是严格相反的。

(二) 数组 & 指针

1. 指针

指针

```c
int main() {
    int* p1;
    int c = 3;
    p1 = &c;
    return 0;
}
```

指针汇编代码示例

```
        ldr     r2, .BRIDGE
.P:
        add     r2, pc, r2
        ldr     r3, .BRIDGE+4
        ldr     r3, [r2, r3]
        ldr     r3, [r3]
        str     r3, [fp, #-8]
        mov     r3, #3
        str     r3, [fp, #-16]      @*(fp-16) load 3
        sub     r3, fp, #16         @r3 <- addr(var c)
        str     r3, [fp, #-12]      @*(fp-12) <- addr(var c)
        ldr     r1, .BRIDGE+8
.CHECK:
        add     r1, pc
        ldr     r2, .L4+4
        ldr     r2, [r1, r2]
        ldr     r1, [r2]
        ldr     r2, [fp, #-8]
        eors    r1, r2, r1      @检查栈
        beq     .SUCCESS
```

NKU

```
        bl      __stack_chk_fail
.SUCCESS:
        pop     {fp, pc}
.BRIDGE:
        .word   _GLOBAL_OFFSET_TABLE_-(.P+8)
        .word   __stack_chk_guard(GOT)
        .word   _GLOBAL_OFFSET_TABLE_-(.CHECK+8)
```

指针的赋值需要将变量地址作为值放入内存中：先将变量的值放入内存（fp-16），然后将（fp-16）的地址放入（fp-12）从而完成对指针的赋值操作。

在使用到数组（指针）时，为了确保栈的安全性，往往需要在上下文和局部变量间插入数字监控栈是否被破坏，在程序结束之后使用异或指令 eor 检查其是否被修改过，如果被修改过需要触发库函数处理。在程序开始之前，先将数放在 fp-8 的位置，在程序结束之后，再对这个位置存放的数进行检查，从而确保栈的安全。

2. 数组 &for 循环

数组、for 循环语句

```
        ldr     r2, .BRIDGE
.START:
        add     r2, pc, r2              @get arr's addr
        ldr     r3, .BRIDGE+4
        ldr     r3, [r2, r3]
        ldr     r3, [r3]
        str     r3, [fp, #-8]
        @ init for loop expr1
        bl      .COND
.LOOP_BODY:
        @...
.COND:
        cmp     @...
        ble     .LOOP_BODY
        bl .CHECK_STACK
.CHECK_STACK:
        @...
        eors    r1, r2, r1
        beq     .END
        bl      __stack_chk_fail
.END:
        @POP STACK

.BRIDGE:
        .word   _GLOBAL_OFFSET_TABLE_-(.START+8)
        .word   __stack_chk_guard(GOT)
```

数组的本质是数组头指针和后面一串连续的地址空间，也需要对栈空间的安全性校验，其大致思路与上面指针的汇编代码一致。对于 for 循环的汇编代码，首先初始化 for 循环的表达式一，

NKU

然后跳转（bl）至条件判断语句段 COND，在 COND 段判断继续循环体 LOOP_BODY 或跳转结束循环，在结束循环后要对栈空间进行校验，成功后再跳入结束段（END）释放栈空间。

(三) 分支 & 循环

1. 跳转指令

有如下几种:

- B: 无条件跳转。
- BL: 链接跳转。跳转同时写 lr 寄存器，保存当前指令下一条指令的地址，常用于函数返回。
- BX/BLX: 跳转到目标寄存器或标签，可能会切换指令集。

跳转指令也能拼接前文提到的后缀，例如 beq、bne 等，实现条件执行

2. if

if 语句一般格式如下:

if 语句

```
@if
    cmp ...
    bxx ELSE
IF:
    @if block
    B END
ELSE:
    @else block
END:
```

首先用 cmp 更新 CPSR，之后用带条件后缀的 b 指令选择是否跳转到 ELSE 块，或继续执行 if 块，最后无条件跳转至 END

3. while&do while

二者结构上有细微区别

while、do...while 语句

```
@while
    B CONDITION
LOOP:
    @loop body
CONDITION:
    @update state

@do...while
LOOP:
    @loop body and update state
```

do...while 语句结构更清晰，可以只使用一个标签；而 while 由于要先判断，因此需要跳转和一个新的标签

ZJU

(四) 函数

1. 栈的使用

栈通过保存返回地址、帧指针来回溯，或用来传递参数等，因此函数调用中栈的使用至关重要。

在 armv5t 中，函数开始会 push fp，结束再恢复；在 armv7 中类似，先 push r7，再将 sp 保存至 r7，用 r7 来操作栈，保存局部变量等；而 sp 只是根据参数传递的需要而使用。

框架如下：

函数调用

```
@function start here
    push    {r7}
    sub sp, sp, #xx
    add r7, sp, #0          @r7 = sp
@function body...
    mov  r0, r3             @return value => r0
    adds r7, r7, #16        @restore stack
    mov  sp, r7             @restore sp
    pop {r7, pc}            @restore r7, pc(if has link)
```

2. 库函数

主要介绍 printf 和 scanf。

printf 函数调用

```
@printf
        .text
    .section    .rodata
    .align  2
.str:
    .ascii  "%d\000"                @printf第一个参数，字符串
    @...
    ldr r3, .BRIDGE
.PRINT:
    add r3, pc
    mov r0, r3
    bl  printf(PLT)
.BRIDGE:
    .word    .str-(.PRINT+4)
```

printf 中，需要解释的是 add r3, pc 这一点。

pc 保存的是.PRINT+4, 是因为无论是 armv7 的三级流水线还是 armv9 的五级流水线，一条指令处于执行阶段时，pc 寄存器已经更新到下下一条指令地址了 (pc+8)，又由于 pc 的更新与指令的执行是同步的，都在时钟上升沿，所以在执行 add 时 pc 应是.PRINT+4

在.BRIDGE 中 str 的地址减去 (.PRINT+4) 又加上了 pc 的 (.PRINT+4)，因此 r3 最终是 str 的地址，最后将 r3 放进 r0 最为第一个参数，使用 bl 调用 printf 函数

对于 scanf 函数调用也是相同过程，但是一般需要紧接一个 stack_check_fail 函数，来检查栈溢出。

NKU

3. 自定义函数

自定义的函数框架与库函数相同，局部变量用 sp 开辟栈空间并保存，函数调用初始和结束要分别 push r7 和 pop r7, pc，来维护栈空间的稳定性

4. 栈传递参数

当参数个数小于 4 个，依次使用 r0、r1、r2、r3 寄存器传参，而更多的参数需要操作 sp 用栈传参:

printf 函数调用

```
@function start here
    push    {r7, lr}
    sub     sp, sp, #32
    add     r7, sp, #8
@a lot of params
    movs    r3, #2              @第一个参数a=2
    str     r3, [r7, #4]        @存到栈中
    movs    r3, #3
    str     r3, [r7, #8]        @b=3 存到下一个位置
    ...
```

caller 中把参数依次入栈，callee 中依次从栈中取参数。

NKU

四、 思考题

如果不是人来手动编译 SysY 程序，而是设计一个计算机程序来完成从 SysY 程序到汇编代码的生成，应该如何做.

(一) 设计思路

高级语言生成汇编代码，对一个编译程序提出了如下要求：

- 能够识别关键字、用户自定义变量常量及函数，识别系统保留符号如 #、基本运算符号如 +、-、< 等，建立词法规则
- 对源程序语法结构有清晰的界定准则，能够将每个元素合并归类，用有限的类别和语法规则来表达无穷可能的源程序
- 分类后，能够将源程序拆解、重组成为层次清晰分明的结构化表示，从而进行进一步的语义分析
- 在前述分析基础之上，生成源程序的中间表示
- 在中间代码生成时，也可以提供一些优化策略，最终生成目标代码

所以这个程序应该包括如下部分：词法分析、语法分析、语义分析、中间代码生成及优化

(二) 词法分析

在词法分析过程中，最关键的是对词法记号的描述。一般情况下，编译系统使用正则文法来描述词法的规则，而对正则文法识别的工具就是有限自动机。通过有限自动机把词法记号识别出来，就完成了词法分析的工作。算法流程为从左至右逐个字符地对源程序进行扫描，产生一个个的单词符号，输出一个个由单词和词素构成的二元组，把作为字符串的源程序改造成为单词符号串的中间程序。

(三) 语法分析

一般情况下语法分析分为两种形式，一种是自顶向下的语法分析方法，另一种是自底向上的语法分析方法。自顶向下型语法分析算法是从语法分析树的根节点开始，使用最左推导的方法，推导构建完整的语法分析树，适用于 LL(k) 文法。LL(k) 文法的第一个 L 是输入从左到右（left to right），第二个 L 是最左推导（leftmost derivation），k 是前瞻符号（lookahead symbol）数量。LL(k) 算法的优点：

- 分析高效（线性时间）.
- 错误定位和诊断信息准确.

自底向上语法分析算法是从语法分析树的叶子节点开始，逐渐向上到达根节点，反向构造出一个最右推导序列，从而构建完整的语法分析树，适用于 LR(k) 文法。LR(k) 文法的 L 是输入从左到右，R 是反向最右推导（rightmost derivation in reverse），k 是前瞻符号数量。

在语法分析阶段需要构建一个十分重要的数据结构，即符号表。符号表贯穿整个编译过程，在有些情况下，词法分析器可以在它碰到组成一个词素的字符串时立刻创建一个对应的符号表条目。但更多情况下，只有语法分析器才能决定使用之前创建得符号表条目还是创建一个新的条

目，因此在语法分析阶段生成符号表更合适。符号表主要包括以下内容：变量记录哈希表、函数记录哈希表、串空间、函数调用实参变量记录链表。

变量声明或者定义时，编译器获取变量类型和名称信息，修改相关字段的内容，然后将信息插入符号表。函数声明时，编译器先插入函数记录到符号表，然后对参数声明处理方式是：先把参数变量记录信息存储在局部变量列表缓存中，若检测出是函数定义再把缓存的变量记录信息真正的插入符号表，否则清空缓冲区。函数定义时，编译器先将函数记录信息插入符号表，再将局部变量的定义依次插入符号表，并且记录函数内插入变量的个数，等到函数定义结束的时候将刚才插入的变量依次从符号表删除，最后清除缓冲区的变量记录，更新符号表。另外，在表达式解析的过程中会产生临时的局部变量，对其也当作正常的局部变量进行处理即可。

（四） 语义分析

语义分析：引用符号表内容，检查语义的合法性并引导代码生成例程。有了语法分析产生的符号表内容，语义处理可以通过查询符号表的信息来对已经声明的语法进行合法性的语义检查。当语义检查没有错误时就可以引导代码生成例程进行代码生成的工作。

（五） 中间代码生成及优化

中间代码生成：使用语法树和符号表，生成三地址码。

中间代码作为桥梁，需要易于翻译成目标机器上的目标代码，此外也可以在生成中间代码时作与机器无关的优化。

NKU