南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

编译系统原理实验报告

定义语言特性及汇编编程

王子莼 袁贞芷

年级：2019 级

专业：计算机科学与技术

指导教师：王刚

2021 年 10 月 12 日

摘要

本次实验是自己编写的编译器前的预备工作。首先选取了 C 语言子集 SysY 语言的部分特性并对它们进行了产生式定义。为了更好地了解编译器的工作，我们写了一些包含这些特性的 SysY 程序样例并将它们翻译为 arm 汇编程序。

关键字：SysY，编译器，arm 汇编

目录

一、 定义语言特性

(一) 终结符集合 V_T

该语言终结符包含:

1. 标识符 Ident
2. 数值常量 IntConst

(二) 非终结符集合 V_N

编译单元	CompUnit
声明	Decl
常量声明	ConstDecl
基本类型	BType
常数定义	ConstDef
常量初值	ConstInitVal
变量声明	VarDecl
变量定义	VarDef
变量初值	InitVal
函数定义	FuncDef
函数类型	FuncType
函数形参表	FuncFParams
函数形参	FuncFParam
语句块	Block
语句块项	BlockItem
语句	Stmt
表达式	Exp
条件表达式	Cond
左值表达式	LVal
基本表达式	PrimaryExp
数值	Number
一元表达式	UnaryExp
单目运算符	UnaryOp
函数实参表	FuncRParams
乘除模表达式	MulExp
加减表达式	AddExp
关系表达式	RelExp
相等性表达式	EqExp
逻辑与表达式	LAndExp
逻辑或表达式	LOrExp
常量表达式	ConstExp

(三) 开始符号 S

编译单元 CompUnit

(四) 产生式 P

1. 编译单元

1. 编译单元 CompUnit → [CompUnit] (Decl | FuncDef)
2. 声明 Decl → ConstDcel | VarDecl

2. 变量与常量

1. 常量声明 ConstDecl → ['**const**'BType ConstDef] | ConstDecl',' ConstDef
2. 基本类型 BType → '**int**'
3. 常数定义 ConstDef → Ident {'['ConstExp']'}' =' ConstInitVal
4. 常量初值 ConstInitVal → ConstExp | '{''}' | ''InitValList''
5. 变量声明 VarDecl → BType VarDef | VarDecl',' VarDef
6. 变量定义 VarDef → **Ident** {'['ConstExp']'} | **Ident** {'['ConstExp']'}' =' InitVal
7. 变量初值 InitVal → Exp | '{'[InitVal {',' InitVal}] | '}'

3. 语句

1. 语句块 Block → '{'{BlockItem}'}'
2. 语句块项 BlockItem → Decl | Stmt
3. 语句 Stmt → LVal '=' Exp ';' | [Exp] ';' | Block | '**if**' '(Cond ')' Stmt ['**else**' Stmt]
   | 'while' '(' Cond ')' Stmt
   | 'break' ';' | 'continue' ';'
   | 'return' [Exp] ';'

4. 表达式

1. 表达式 Exp → AddExp
2. 条件表达式 Cond → LOrExp
3. 左值表达式 LVal → **Ident** {'[' Exp ']'}
4. 数值 Number → IntConst
5. 一元表达式 UnaryExp → PrimaryExp | **Ident** '(' [FuncRParams]')' | UnaryOp UnaryExp
6. 单目运算符 UnaryOp → '+' | '−' | '!'
7. 函数实参表 FuncRParams → Exp { ',' Exp}

8. 乘除模表达式 MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
9. 加减表达式 AddExp → MulExp | AddExp ('+' | '−') MulExp
10. 关系表达式 RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
11. 相等性表达式 EqExp → RelExp | EqExp ('==' | '!=') RelExp
12. 逻辑与表达式 LAndExp → EqExp | LAndExp '&&' EqExp
13. 逻辑或表达式 LOrExp → LAndExp | LOrExp '||' LAndExp
14. 常量表达式 ConstExp → AddExp

5. 函数

1. 函数定义 FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
2. 函数类型 FuncType → '**void**' | '**int**'
3. 函数形参表 FuncFParams → FuncFParam { ',' FuncFParam}
4. 函数形参 FuncFParam → BType **Ident** ['[' ']' { '['Exp']'}]

二、 汇编编程

(一) SysY 代码

1. 全局变量

全局变量

```
const int x=5;
int n;
int y;
int z;
int array[5]={2,6,13,3,99};
```

2. 函数

函数

```
int func(int x, int y, int z)
{
    if (x<y && x<z){
        return y-x;
    }
    else if(x<y && x>=z){
        return x*y-z;
    }
    else{
        return z*x+y;
    }
}
```

3. 主函数

主函数

```
 int main()
{
    int sum=0;
    scanf("%d",&n);
    scanf("%d",&y);
    scanf("%d",&z);
    for(int i=1;i<=n;i++){
        sum+=i;
    }
    printf("%d\n",sum);
    int res=func(x,y,z);
    printf("result is %d\n",res);

    res=1;
```

```
    for(int i=0;i<5;i++){
        res*=array[i];
    }
    printf("%d",res);
    return 0;
}
```

(二) 汇编代码

1. 汇编知识

1. 函数调用：arm 汇编与 C 之间的函数调用需要符合 ATPCS，建议函数的形参不超过 4 个，如果形参个数少于或等于 4，则形参由 r0,r1,r2,r3 四个寄存器进行传递；若形参个数大于 4，大于 4 的部分必须通过堆栈进行传递。

   r0 用来存放函数的第一个参数，r1 用来存放第二个参数，r2 用来存放第三个参数，r3 用来存放第四个参数。其中 R0 还用来返回函数的调用结果，对应 C 函数里面的 return value 语句中的 value 存放在 r0 中。

2. 桥接: 连接全局变量的地址

3. 数组: 用“ldr r0, =array”指令可以将数组的地址放到 r0 寄存器中。在全局变量声明时，可以通过在 array 标签后连续的.word 进行数组中变量的初始化。

4. 循环与跳转: 用 loop 标识，设置一个计数器 i 每一次执行循环自增，用 cmp 指令和 blt，bne，bgt 条件跳转指令结合。可以利用 ldr 和 str 进行数组中相应地址的值的访问，详细见总结。

5. 输入与输出：调用库函数，传递的参数字符串，值，分别放在寄存器 r0 和 r1 里。

2. 代码

以下是我们编写的上述 SysY 程序的汇编代码，同时可以在我们的 gitlab 仓库中获取源文件。

汇编代码

```
    .arch armv5t

    .comm   y, 4
    .comm   n, 4
    .comm   z, 4
    .text
    .align  2
    .section    .rodata
    .align  2

    .global x
    .align      4
    .size       x, 4
x:
```

```
    .word       5

    .global array
    .align  4
    .size array, 20
array:
    .word   2
    .word   6
    .word   13
    .word   3
    .word   99

_str0:
    .ascii  "%d\n\0"
    .align 2

_str1:
    .ascii  "%d\0"
    .align  2

_str2:
    .ascii  "result is %d\n \0"
    .text
    .align  2

func:
    str     fp, [sp,#-4]!
    mov     fp, sp
    sub     sp, sp, #12

    str     r0, [fp, #-8]    @将[fp+8]处的值放进r0
    str     r1, [fp, #-12]
    str     r2, [fp, #-4]
    cmp     r0, r1           @比较x和y
    blt     .L2
    mul     r0, r2, r0       @x>=y,z*x
    add     r0, r0, r1       @z*x+y
    b       .L4
.L2:
    cmp     r0, r2       @比较x和z
    blt     .L3
    mul     r0, r1, r0   @x*y
    sub     r0, r0, r2   @x*y-z
    b       .L4
.L3:
     sub     r0, r1, r0      @x<z,y-x
.L4:
    add     sp, fp, #0
```

```
63      ldr     fp, [sp], #4
64      bx      lr
65
66      .global main
67  main:
68      push    {fp, lr}
69      add     fp, sp, #4
70      mov     r3 , #0          @初始化sum
71      ldr     r1 , _bridge+8
72      ldr     r0 , _bridge+16
73      bl      __isoc99_scanf   @输入n,r0是str1,r1是&n
74      ldr     r3, _bridge+8
75      ldr     r1, [r3]         @r1是n
76      mov     r2 , #1
77      mov     r3 , #0
78  loop:
79      add   r3 , r3 ,r2        @sum = sum + i
80      add r2 , r2 ,#1          @i++
81      cmp   r2 , r1            @判断 i 是否等于val
82      bgt   end                @若大于 跳转至END处
83      b     loop               @若小于等于 跳转至LOOP处进入下次循环
84  end:
85      ldr     r0, _bridge+12   @_str0
86      mov     r1, r3           @_要打印的放进r1
87      bl      printf
88      ldr     r1, _bridge+4    @r1=&y
89      ldr     r0, _bridge+16
90      bl      __isoc99_scanf   @输入y,r0是str1,r1是&y
91      ldr     r0, _bridge+16
92      ldr     r1, _bridge+24
93      bl      __isoc99_scanf    @输入z,r0是str1,r1是&z
94      ldr     r3, _bridge+4
95      ldr     r2, _bridge
96      ldr     r4, _bridge+24
97      ldr     r0, [r2]         @r0是x
98      ldr     r1, [r3]         @r1是y
99      ldr     r2, [r4]         @r2是z
100     bl      func             @调用func
101     mov     r1, r0
102     ldr     r0, _bridge+20
103     bl      printf           @r0是"result is %d", r1是result
104     mov     r0, #0
105
106     ldr     r0, =array       @把array[0]的地址存到r0里,没有=就是存值
107     mov     r1, #0           @初始化i
108     mov     r2, #1           @初始化结果mul
109 arrLoop:
110     ldr     r3,[r0],#4       @数组中的值拿出来存进r3,并将r0+4写入r0
```

```
111     mul     r2, r3, r2
112     add     r1, r1, #1
113     cmp     r1, #5
114     bne     arrLoop
115     ldr     r0, _bridge+12  @_str0
116     mov     r1, r2          @_要打印的放进r1
117     bl      printf
118     pop     {fp, pc}
119
120 _bridge:
121     .word   x
122     .word   y
123     .word   n
124     .word   _str0
125     .word   _str1
126     .word   _str2
127     .word   z
```

(三) 测试结果

调试指令

```
1 arm-linux-gnueabihf-gcc example.S -o example
2 qemu-arm -L /usr/arm-linux-gnueabihf/ ./example
```

对于函数的三种分支三种不同的运行结果如下图所示:

第一行是依次输入的三个值: 第一个数是累加循环的全局变量 n 的值, 后两个数是传进 func 函数的参数 y，z 的值。

第二行是输出的累加的结果。

第三行是输出的函数返回的结果。

第四行是数组 array 中的数累乘的结果。

分支情况 1 如图1所示，n=3，x=5，y=5，z=7，执行 z*x+y:

```
wzc@wzc-virtual-machine:~/文档/test5$ arm-linux-gnueabihf-gcc example.S -o examp
le
wzc@wzc-virtual-machine:~/文档/test5$ qemu-arm -L /usr/arm-linux-gnueabihf ./exa
mple
3 5 7
6
result is 40
46332
```

图 1: 累加结果为 6，函数结果为 40

分支情况 2 如图2所示，n=4，x=5，y=3，z=6，执行 z*x+y：

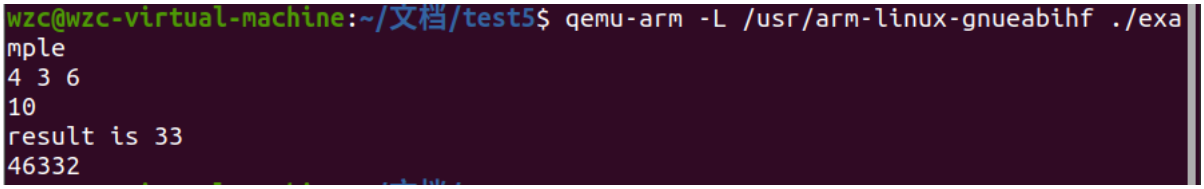

图 2: 累加结果为 10，函数结果为 33

分支情况 3 如图3所示，n=5，x=5，y=6，z=4，执行 x*y-z：

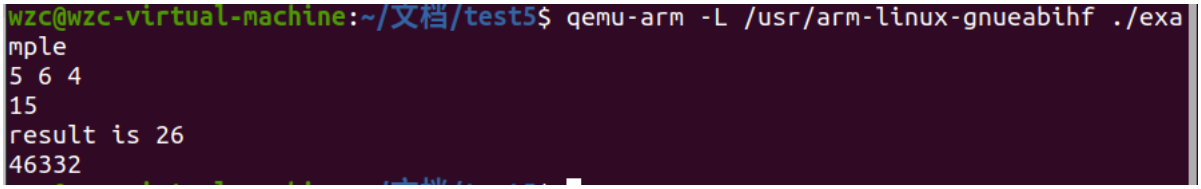

图 3: 累加结果为 15，函数结果为 26

三、 总结

(一) 分工

1. 代码部分

1. 函数：袁贞芷
2. 循环：王子纯
3. 输入输出：袁贞芷，王子纯
4. 变量定义：袁贞芷，王子纯
5. 调试：袁贞芷，王子纯

2. 报告部分

1. 摘要：袁贞芷
2. 产生式设计：袁贞芷，王子纯
3. 汇编知识：袁贞芷，王子纯
4. 感悟：袁贞芷，王子纯

(二) 感悟

在编写汇编函数的过程中，我们由易到难循序渐进学习 arm 汇编，从简单的"Hello world"开始，到对着实验指导书里的程序进行仿写。同时也不可避免的遇到许多 bug，查阅了不少的资料。最终我们的汇编程序里实现了变量声明与定义、函数调用、数组、循环与跳转与输入输出。

在编写函数的过程中，我对函数体中间的几条 ldr 与 str 指令仍有疑问，实验指导书给的样例程序中对参数进行了 str 指令，我查阅了一些资料发现在机器生成的汇编代码中，不仅要执行 str，还要执行 ldr 指令。但并没有详细的资料说明为什么要使用这些指令。一个猜想是为了保存传入的参数，但函数的栈在调用结束后被释放，仍然没有起到保存参数的作用。

ldr 指令的寻址方式比较灵活。在我的尝试下，指令“mov r3,[r1]”，会报错。必须更改为，“ldr r3,[r1]”。并且指令“ldr r3,[r1], #4”可以完成两个动作将存储器地址为 r1 的数据读入寄存器 r3，并将 r1+4 的值存入 r1。而指令“ldr r3,[r1,#4]”则是将 r1+4 的数据读入 r3。str 相关指令相似，只不过读取存储数据的方向相反。

在小组协作中，使用适当的工具十分重要。在本次实验中，我和队友研究并使用了 gitlab，每个人编写自己的代码，submit 并 merge，大大提高了工作的并行性。免除了传统的文件传输工作。并且在撰写实验报告的时候，我们也在 overleaf 上同步编辑，效率大大提高。