南開大學
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

编译系统原理实验报告

预备工作 2　定义编译器 &ARM 汇编编程

张丛　刘国民

年级：2021 级

专业：信息安全

指导教师：王刚

2023 年 10 月 10 日

摘要

本次实验我们参考 miniSysY 的全部文法，结合我们想要实现的 SysY 语言特性设计了上下无关文法。通过四元组对 CFG 进行了描述，同时为了熟悉目标代码——ARM 汇编语言，我们编写了斐波那契数列和阶乘程序的汇编代码，程序能够正确执行。其中 CFG 描述部分由刘国民和张丛同学共同完成，ARM 汇编部分刘国民同学负责斐波那契数列程序的编写，张丛同学负责阶乘程序的编写。

关键字：ARM 汇编；上下无关文法

目录

一、 定义编译器

(一) 编译器支持的 SysY 语言特性

基础 track:

- 数据类型：int
- 变量声明、常量声明，常量、变量的初始化
- 语句：赋值（=）、表达式语句、语句块、if、while、return
- 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））
- 注释
- 输入输出

竞赛 track:

- 数组
- 变量、常量作用域——在语句块中包含变量、常量声明，break、continue 语句
- 函数
- 代码优化

  -寄存器分配优化方法

  -基于数据流分析的强度削弱、代码外提、公共子表达式删除、无用代码删除等

设计 CFG 时我们将以上语言特性均考虑在内，在后续实验中会尝试实现所有语言特性

(二) CFG 描述 SysY 语言特性

我们利用上下无关文法对编译器所支持的 SysY 语言特性子集进行形式化定义，CFG 包括终结符集合 V_T，非终结符集合 V_N，开始符号 S 和产生式集合 P 四个部分。

1. 终结符集合 V_T

- 标识符

$$
\begin{aligned}
id \rightarrow & id\_nondigit \\
& | id\ id\_nondigit \\
& | id\ digit
\end{aligned}
$$

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

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

- 数值常量

$$
\begin{aligned}
integer\_const \rightarrow & decimal\_const \\
& |octal\_const \\
& |hex\_const
\end{aligned}
$$

$$
\begin{aligned}
decimal\_const \rightarrow & nonzero\_digit \\
& |decimal\_const\ digit
\end{aligned}
$$

$$\text{octal\_const} \rightarrow 0|octal\_const\ octal\_const$$

$$
\begin{aligned}
hex\_const \rightarrow & hex\_prefix\ hex\_gigit \\
& |hex\_const\ hex\_digit
\end{aligned}
$$

$$
\begin{aligned}
\text{hex\_prefix} &\rightarrow\ '0x'|'0X' \\
\text{nonzero\_digit} &\rightarrow 1|2|3|4|5|6|7|8|9 \\
\text{octal\_dugit} &\rightarrow 0|1|2|3|4|5|6|7
\end{aligned}
$$

$$
\begin{aligned}
hex\_digit \rightarrow & 0|1|2|3|4|5|6|7|8 \\
& |9|a|b|c|d|e|f \\
& |A|B|C|D|E|F
\end{aligned}
$$

- 运算符

{ +,-,*,/,%,=,!,>,<,>=,<=,==,!=,&&,|| }

- 关键字

{ void, int, if,else,while,break,const,return }

- 基本符号

{ ;, [,], (,), , , /*, */, // }

2. 非终结符集合 V_N

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

3. 开始符号 S

开始符号：CompUnit

4. 产生式集合 P

- CompUnit -> [CompUnit] (Decl | FuncDef)
- Decl -> ConstDecl | VarDecl
- ConstDecl -> 'const' BType ConstDef { ',' ConstDef } ';'
- BType -> 'int'
- ConstDef -> Ident { '[' ConstExp ']' } '=' ConstInitVal
- ConstInitVal -> ConstExp | '' [ConstInitVal { ',' ConstInitVal }] ''
- VarDecl -> BType VarDef { ',' VarDef } ';'
- VarDef -> Ident { '[' ConstExp ']' } | Ident { '[' ConstExp ']' } '=' InitVal
- InitVal -> Exp | '' [InitVal { ',' InitVal }] ''
- FuncDef -> FuncType Ident '(' [FuncFParams] ')' Block
- FuncType -> 'void' | 'int'
- FuncFParams -> FuncFParam { ',' FuncFParam }
- FuncFParam -> BType Ident ['[' ']' { '[' Exp ']' }]
- Block -> '' { BlockItem } ''
- BlockItem -> Decl | Stmt

- Stmt -> LVal '=' Exp ';' | [Exp] ';' | Block | 'if' '(' Cond ')' Stmt ['else' Stmt] | 'while' '(' Cond ')' Stmt | 'break' ';' | 'continue' ';' | 'return' [Exp] ';'
- Exp -> AddExp
- Cond -> LOrExp
- LVal -> Ident '[' Exp ']'
- PrimaryExp -> '(' Exp ')' | LVal | Number
- UnaryExp -> PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp
- UnaryOp -> '+' | '-' | '!' // 注：保证'!' 仅出现在 Cond 中
- FuncRParams -> Exp ',' Exp
- MulExp -> UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
- AddExp -> MulExp | AddExp ('+' | '−') MulExp
- RelExp -> AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
- EqExp -> RelExp | EqExp ('==' | '!=') RelExp
- LAndExp -> EqExp | LAndExp '&&' EqExp
- LOrExp -> LAndExp | LOrExp '||' LAndExp
- ConstExp -> AddExp // 在语义上额外约束这里的 AddExp 必须是一个可以在编译期求出值的常量

二、 ARM 汇编编程

ARM 汇编程部分我们以斐波那契数列和阶乘求解为例，编写了 SysY 程序和对应的 ARM 汇编程序。熟悉了源语言和目标语言的语言特性，为后续实现编译器打好基础。

(一) 斐波那契数列

SysY 程序与 C 程序类似，具体代码如下：

1. 斐波那契数列 SysY 程序

斐波那契数列 SysY 程序

```
#include<stdio.h>
int main()
{
  int a, b, i, t, n;
  a = 0;
  b = 1;
  i = 1;
  printf("Please␣enter␣the␣number␣of␣items␣in␣the␣Fibonacci␣sequence:");
```

```
    scanf("%d",&n);
    printf("The result is:\n%d\n",b);
    while (i < n){
    t = b;
    b = a + b;
    printf("%d\n",b);
    a = t;
    i = i + 1;
    }
    return 0;
}
```

由于 SysY 语言是 C 语言的子集，故源代码采用.c 格式保存。通过以下命令编译链接后，程序可以正确执行。

```
gcc fib.c -o fib
.\fib
```

2. 斐波那契 ARM 汇编程序

汇编代码如下所示，其中注释以 @ 开头

斐波那契数列 ARM 汇编代码

```
.arch armv5t                @表示使用 ARMv5t 指令集


@ 数据段
.comm   n, 4                @ 未初始化变量
.data
    a:   .word 0            @ arm架构下word表示32bits,与x86(16bits)相区别
    b:   .word 1
    i:   .word 1
    t:   .word 0


@ 只读数据段
.rodata:
    .align 2
info:
    .asciz  "Please enter the number of items in the Fibonacci sequence:"
    .align 2
input:
    .asciz "%d"
    .align 2
output1:
    .asciz "The result is:\n%d\n"
    .align 2
output2:
    .asciz "%d\n"
```

```
    .align 2


@ 代码段
.text
    .align 2
    .global main
main:
    push  {fp,lr}                @ 从左到右压入栈中，作用是保存返回地址
        和栈基地址
    add   fp, sp, #4             @ 开辟函数栈帧
    ldr   r0, _bridge+20         @ 传入参数
    bl    printf                 @ 调用输出函数


    ldr   r1, _bridge            @ r1=&n
    ldr   r0, _bridge+24         @ r0=input
    bl    __isoc99_scanf         @ scanf("%d",&n)

    ldr   r0, _bridge+4
    ldr   r1, [r0]               @ r1=b
    ldr   r0, _bridge+28         @ r0=output1
    bl    printf

LOOP:
    ldr   r0, _bridge+8
    ldr   r1, [r0]               @ r1=i

    ldr   r0, _bridge
    ldr   r2, [r0]               @ r2=n

    cmp   r1, r2
    bge   L1                     @ 退出循环

    ldr   r0, _bridge+4
    ldr   r3, [r0]               @ r3=b

    ldr   r0, _bridge+32
    str   r3, [r0]               @ t=r3

    ldr   r0, _bridge+12
    ldr   r4, [r0]               @ r4=a
    add   r3,r3,r4               @ r3=r3+r4
    ldr   r0, _bridge+4
    str   r3,[r0]                @ b=r3

    ldr   r1,[r0]                @ r1=b
    ldr   r0,_bridge+16          @ r0=output2
```

```
    bl    printf

    ldr   r0,_bridge+32
    ldr   r1,[r0]                    @ r1=t

    ldr   r0,_bridge+12
    str   r1,[r0]                    @ a=r1

    ldr   r0,_bridge+8
    ldr   r2,[r0]                    @ r2=i
    add   r2,r2,#1                   @ r2=r2+1
    str   r2,[r0]                    @ i=r2
    b     LOOP


L1:
    mov   r0, #0
    pop   {fp, pc}                   @ return 0

_bridge:
    .word n
    .word b
    .word i
    .word a
    .word output2
    .word info
    .word input
    .word output1
    .word t

    .section .note.GNU-stack,"",%progbits
```

编写完汇编代码后，通过交叉编译将汇编代码编译链接成可执行文件：

```
arm-linux-gnueabihf-gcc -c fib.S -o fib.o
arm-linux-gnueabihf-gcc fib.o -o fib
```

打开文件后进行测试，程序成功执行且结果正确，如下图所示：

```
liu1114@liu1114-virtual-machine:~$ arm-linux-gnueabihf-gcc -c fib.S -o fib.o
liu1114@liu1114-virtual-machine:~$ arm-linux-gnueabihf-gcc fib.o -o fib
liu1114@liu1114-virtual-machine:~$ ./fib
Please enter the number of items in the Fibonacci sequence:12
The result is:
1
1
2
3
5
8
13
21
34
55
89
144
```

图 1: fib.S

(二) 阶乘

1. 阶乘 SysY 程序

sysY 语言与 C 语言类似。在阶乘的 sysY 程序中，尽可能地包含了支持的语言特性，设置有全局变量、分支、循环、函数调用等等。代码如下:

```
#include<stdio.h>

int n=0;  //全局变量

int factorial(int n) {
    int re=1;

    if(n>=1)          //分支
    {
        while(n)      //循环
        {
            re*=n;
            n--;
        }
    }
    return re;
}

int main() {
    scanf("%d",&n);
    int result = factorial(n);    //函数调用
    printf("%d\n",result);
```

```
    return 0;
}
```

2. 阶乘 ARM 汇编程序

在 Ubuntu 中编写写好 factorial.S 文件后，使用命令:

```
arm-linux-gnueabihf-gcc factorial.S -o fac
```

得到可执行程序 fac。
然后使用命令:

```
qemu-arm -L /usr/arm-linux-gnueabihf/ ./fac
```

运行可执行文件可进行阶乘求解。如图:

```
root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# ls
factorial.S
root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# arm-linux-gnueabihf-gcc factorial.S -o fac
root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# qemu-arm -L /usr/arm-linux-gnueabihf/ ./fac
7
result is 5040
root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# ls
fac  factorial.S
root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2#
```

图 2: fact.S

分别验证 0! 和 12!，结果无误:

```
root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# qemu-arm -L /usr/arm-linux-gnueabihf/ ./fac
0
```

图 3: fac0

```
root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# qemu-arm -L /usr/arm-linux-gnueabihf/ ./fac
12
result is 479001600
```

图 4: fac12

arm 汇编代码如下:

```
    .arch armv7-a          @架构
    .comm a,4              @全局变量a,占4字节
    .text
    .align 2
    .section .rodata
    .align 2
_str0:                     @字符串str0和str1
    .ascii "%d\0"
    .align 2
_str1:
    .ascii "result is %d\n"
    .text
```

```
13      .align 2
14
15      .global fac
16  fac:                        @ 函数 int factorial(int n)
17      str fp,[sp,#-4]!        @表示将寄存器 fp 压入栈中, 同时更新 sp 指针
18      mov fp,sp               @将 sp 指针设置为当前栈指针
19      sub sp,sp,#12           @开辟栈空间
20      str r0,[fp,#-12]        @ r0 = n
21      mov r8,#1               @保存结果
22      cmp r0,#1
23      blt .end_fac
24
25  .loop:                      @循环
26      mul r8,r0
27      sub r0,#1
28      cmp r0,#0
29      bne .loop
30      b .end_fac
31
32  .end_fac:
33      add sp,fp,#0            @恢复栈空间, 恢复寄存器
34      ldr fp,[sp],#4
35      bx lr                   @返回
36
37      .global main
38  main:
39      push {fp,lr}
40      add fp,sp,#4
41      ldr r1,_bridge          @*r1=n
42      ldr r0,_bridge+4        @*r0="%d\n"
43      bl __isoc99_scanf
44      ldr r3,_bridge          @r3=&n
45      ldr r0,[r3]             @r0=n
46      bl fac
47      mov r1,r8
48      ldr r0,_bridge+8        @"result is %d\n"
49      bl printf
50      mov r0,#0
51      pop {fp, pc}
52
53  _bridge:
54      .word a
55      .word _str0
56      .word _str1
57
58      .section .note.GUN-stack,"",%progbits     @避免运行时出现的一些问题
```

参考文献