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

吴晨宇 郭坤昌

年级：2020 级

专业：信安 & 计科

指导教师：王刚

2022 年 10 月 14 日

摘要

本工作由吴晨宇和郭坤昌合作完成。报告首先说明二人在工作中的详细分工，接下来具体介绍设计的 SysY 语言子集特性和 Arm 汇编编程的详细过程。

关键字：SysY 语言子集, Arm 汇编

目录

一、 分工

本次工作由吴晨宇（学号 2012023）和郭坤昌（学号 2012522）合作完成。具体分工为：

1. 两人经过讨论，共同完成要实现的 SysY 语言子集特性选择、设计和编写。

2. Arm 汇编部分，针对 Fibonacci 函数，吴晨宇从基于 for 循环的实现进行实验，郭坤昌从基于递归的实现进行实验，相互总结交流，并各自编写实验结果。

实验过程的所有文件保存在gitee和gitlab 希冀平台中。

二、 要实现的 SysY 语言子集特性

我们要实现的 SysY 语言子集由如下部分构成：

1. 终结符：标识符（Ident）、数字（Number）。

2. 非终结符：编译单元（最顶层抽象概念，亦为开始符号）、变量声明（VarDecl）、函数声明（FuncDecl）等一系列抽象概念。

3. 产生式：表示非终结符之间、非终结符与终结符之间关系，如函数声明由函数返回值类型（Fun 从 Type）、函数名（即标识符）、参数列表（VarList）、语句块（Block）组成。

以下为基于上下文无关文法的形式化定义及相应注释：

(一) 开始符号

编译单元 CompUnit 为开始符号。整个编译单元由变量声明和函数声明组成。

CompUnit→ $CompUnit\ Decl \mid \epsilon$

$Decl \rightarrow VarDecl \mid FuncDecl$

(二) 变量声明

变量声明定义为可一次声明多个同一类型的变量，并对变量用表达式赋值。变量类型包括整型和单精度浮点型。

VarDecl→ $VarType\ VarDefList';'$

$VarType \rightarrow' int' \mid 'float'$

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

$VarDef \rightarrow Ident\ '='\ \ Exp$

(三) 函数声明

函数声明由函数返回值类型（包括空类型、整型、单精度浮点型）、函数名、形参列表、语句块组成。

FuncDecl → $FuncType\ Ident\ '('\ VarDefList\ ')'\ Block$

$FuncType \rightarrow' void' \mid 'int' \mid 'float'$

(四) 语句块

语句块由大括号将变量声明、函数调用、语句包含。其中，语句由语句块、赋值语句、条件分支语句、循环语句和无条件跳转语句组成。

Block $\rightarrow$' {' $BlockItem$ '}'
$BlockItem \rightarrow VarDecl \mid Ident$ '(' $VarDeclList$ ')' ';' $\mid Stmt$
$Stmt \rightarrow Ident$ ' =' exp ';'
$\mid Block$
$\mid$ 'if' '(' $Cond$ ')' $Stmt$ 'else' $Stmt$
$\mid$ 'while' '(' $Cond$ ')' $Stmt$
$\mid$ 'break' ';'
$\mid$ 'continue' ';'
$\mid$ 'return' Exp ';'

(五) 表达式

表达式由如下的关系表达式和运算表达式组成。

Cond -> LOrExp
LOrExp -> LAndExp | LAndExp '||' LOrExp
LAndExp -> EqExp | LAndExp '' EqExp
EqExp -> RelExp | EqExp ('==' | '!=') RelExp
RelExp -> AddExp | RelExp ('>' | '<' | '>=' | '<=') AddExp
AddExp -> MulExp | AddExp ('+' | '-') MulExp
MulExp -> UnaryExp | MulExp ('*' | '/') UnaryExp
Exp -> AddExp
AddExp -> MulExp | AddExp ('+' | '-') MulExp
MulExp -> UnaryExp | MulExp ('*' | '/') UnaryExp
UnaryExp -> PrimaryExp | UnaryOp UnaryExp
PrimaryExp -> '(' Exp ')' | Var | Number
UnaryOp -> '+' | '-' | '!'

(六) 终结符

终结符由数字和标识符组成。

数字给出类似 Fortran 中数字的定义，即支持 "$[0-9]+$" 和 "$[0-9]+$" 的形式，使用正则表达式表示为:

$Number \rightarrow [-+]?([0-9]*?[0-9]+|[0-9]+)(E(+|-)?[0-9]+)?$

标识符表示为:

$Ident \rightarrow [a-zA-Z\_][a-zA-Z0-9\_]*$

三、 Arm 汇编编程

针对如上设计的 SysY 语言子集，我们对求 Fibonacci 数列的函数，从 for 循环和递归形式分别编写 arm 汇编程序并检验输入结果的正确性。

(一) 基于 for 循环的 Fibonacci 函数

1. SysY 程序

fib 的 SysY 程序

```
#include<stdio.h>
int a = 0, b = 1, n;
int main()
{
        int i, t;
        printf("%d\n%d\n",a,b);
        i = 1;
        while (i < n)
        {
                t = b;
                b = a + b;
                printf("%d\n",b);
                a = t;
                i = i + 1;
        }
        return 0;
}
```

这个程序使用了全局变量与一个简单的循环，并调用了 printf() 函数来显示输出，SysY 库函数中的 putin() 与其类似。

2. ARM 汇编程序

SysY 程序的 ARM 汇编代码 fib.S

```
        .arch armv7-a
    .section .data
    .global a
    .type a, %object
    .size a, 4
a:
    .word 0
    .global b
    .type b, % object
    .size b, 4
b:
    .word 1
    .global n
    .type n, % object
    .size n, 4
n:
    .word 10
str1:
    .ascii "%d\n%d\n\0"
str2:
    .ascii "%d\n\0"
    .section .text
    .global fibonacci
```

```
24      .type fibonacci, %function
25  fibonacci:
26      push {lr}
27      ldr r1, _bridge
28      ldr r1, [r1 ]
29      mov r4, r1                  @a,获得a的内存地址
30      ldr r2, _bridge+4
31      ldr r2, [r2 ]
32      mov r5, r2                  @b
33      ldr r0, _bridge+12          @str1
34      push {r1, r2}
35      bl printf                   @printf(" %d\n%d\n" , a, b)
36      pop {r1, r2}
37      ldr r6, _bridge+8           @n
38      ldr r6, [r6 ]
39      mov r7, #0x0                @i
40  .l1:
41      mov r8, r5                  @t=b
42      add r5, r4, r5              @b=a+b
43      ldr r0, _bridge+16          @str2
44      mov r1, r5
45      push {r0, r1 ,r2, r3}
46      bl printf                   @printf("%d\n",b+a)
47      pop {r0, r1, r2, r3}
48      mov r4, r8                  @a=t
49      add r7, r7, #1              @i++
50      cmp r7, r6
51      blt .l1
52      pop {pc}
53      .global main
54      .type main, %function
55  main:
56      push {lr}
57      bl fibonacci
58      mov r0, #0
59      pop {pc}
60  _bridge:
61      .word a     @0
62      .word b     @4
63      .word n     @8
64      .word str1  @12
65      .word str2  @16
```

fib.S 对应上面的 SysY 程序，实现一个 fibonacci 数列的输出。使用了 ARM 汇编中的全局变量、局部变量，程序中还涉及了算术运算、while 循环、复合语句、赋值语句、变量声明等语言特性，使用了 C 语言中的 printf 函数来展示输出，这是因为 SysY 库 IO 函数的 putint 的使用和 printf 函数非常类似，通过函数的调用可以理解 ARM 汇编的栈结构和函数传参等问题。

编写一个简单的汇编代码要先清楚各个段的定义。数据段、常量数据段、代码段这几个是常

用的段。最简单的汇编编程我们需要定义变量、使用变量。如果是全局变量或者常量定义在数据段（.data 为初始化数据段，.bss 为未初始化数据段，.rodata 为常量数据段，之后桥接（_bridge）这些数据进行使用。如果是在函数中使用局部变量，那么可以直接使用寄存器或者使用栈来达到目的，在简单编程中，10 多个寄存器其实够用了。

3. 程序运行

使用 arm-linux-gnueabihf-gcc fib.S -o fibo -static 生成可执行文件 fib，使用命令：qemu-arm ./fib 运行可执行文件。输出结果为：0 1 1 2 3 5 8 13 21 34 55 89，即 n+2 个 Fibonacci 数列。

(二) 基于递归的 Fibonacci 函数

基于 Fibonacci 数列的特点，设计如下的基于递归的 c 函数代码。

递归求 Fibonacci 数列的 C 语言代码

```c
#include <stdio.h>

int fib(int n){
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
}

int main(){
    return fib(10);
}
```

使用 32 位 Arm 指令，编写并注释得到如下对应的 Arm 汇编程序。

递归求 Fibonacci 数列的 Arm 汇编程序

```
.global main

fib:
        push    {r4, r11, lr}           @ 保护现场
        sub sp, sp, #8                  @ 分配一定的栈空间
        add     r11, sp, #0             @ 保存栈顶指针
        str     r0, [r11, #4]           @ 存n
        cmp     r0, #1                  @ if (n<2) return n; else return fib(
            n-1)+fib(n-2);
        bgt     fib_recur
        b   fib_end
fib_recur:                              @ 递归调用
        ldr     r3, [r11, #4]           @ r3 = n
        sub     r3, r3, #1              @ r3 = n-1
        mov     r0, r3                  @ r0 = n-1
        bl      fib                     @ r0 = fib(n-1)
        mov     r4, r0                  @ r4 = fib(n-1)
        ldr     r3, [r11, #4]           @ r3 = n
        sub     r3, r3, #2              @ r3 = n-2
        mov     r0, r3                  @ r0 = n-2
```

```
20          bl      fib                 @ r0 = fib(n-2)
21          mov     r3, r0              @ r3 = fib(n-2)
22          add     r3, r3, r4          @ r3 = fib(n-1)+fib(n-2)
23  fib_end:                            @ 结束
24          mov     r0, r3              @ r0 = fib(n)
25          add     r11, r11, #8
26          mov     sp, r11             @ 恢复栈顶指针
27          pop     {r4, r11, pc}       @ 恢复现场
28
29  main:
30          push {r11, lr}              @ 保护现场
31          add r11, sp, #0             @ 保存栈顶指针
32          mov r0, #10                 @ r0 = n = 10
33          bl  fib                     @ r0 = fib(n)
34          pop {r11, pc}               @ 恢复现场
```

编写该汇编代码的困难在于正确分配栈空间，并及时储存中间结果。编写过程中有如下问题和收获：

1. 暂存 fib(n-1) 的 r4 寄存器在之后再次被使用，参与计算 fib(n-2)，因此需要将该中间结果保留在栈中。如图1所示

```
fib:
    push    {r4, r11, lr}   @ 保护现场
    sub sp, sp, #8          @ 分配一定的栈空间
    add r11, sp, #0         @ 保存栈顶指针
```

图 1: 调用函数前保存中间计算结果

2. 函数调用前需要分配一定栈空间，将帧指针寄存器（fr 即 r11）和链接寄存器 (lr) 入栈以保护现场。调用结束后，恢复帧指针寄存器，将帧指针寄存器和指令寄存器出栈，实现恢复到函数调用前运行位置。

3. 函数调用一般使用 4 个寄存器作为实参传递，超过 4 个参数时需要借助栈进行参数传递。寄存器 0 作为函数返回值的寄存器。若返回值不为 32 位，还需要借助其他寄存器辅助保存返回值。

最终计算 fib(10) 并通过主函数返回，结果如图2所示，运算正确。

```
(base) bill@bill-Lenovo-V15-IWL:~/Desktop/2022/lab2/program/asm (copy)$ make all
arm-linux-gnueabihf-as main.s -o main.o
main.s: Assembler messages:
main.s: Warning: end of file not at end of a line; newline inserted
arm-linux-gnueabihf-gcc main.o -o main
qemu-arm -L /usr/arm-linux-gnueabihf/ ./main
make: *** [Makefile:28: all] Error 55
```

fib(10)

图 2: 调用函数前需要保存中间计算结果

对于该部分 Arm 汇编的编写使用，参照了arm system call table和arm-asm tutorial。