预备工作 2——定义你的编译器 & 汇编编程

杨侯哲 李煦阳
2020 年 10 月
费迪
2021 年 9 月
张书睿 贺祎昕
2023 年 9 月

目录

1 实验描述

基于“预备工作 1”，继续：

1. 确定你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 2 章及第 2 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你的 SysY 语言子集。
2. 设计几个 SysY 程序（如“预备工作 1”给出的阶乘或斐波那契），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。这些程序应该尽可能全面地包含你支持的语言特性。

思考：

- 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？
- 注意：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（无穷无尽）是不可能的，怎么办？搞定每个语言特性如何翻译即可！

  每个语言特性仍然有无穷多个合法的实例（a=1，b=2.0，…），怎么办？符号化——语法制导翻译！参见讲义 2.8 节。

1.1 实验要求

要求：

- 确定上机大作业分组，后续实验中不再调整。
- 撰写研究报告（要求同前），编写的 ARM 汇编要给出 GitLab 项目链接。
- 在报告中用一个小节明确、详细地指出两人的分工情况。注意，不能是“A 负责代码，B 负责文档”！一方面，两人都要进行设计、实现工作，不能只撰写文档；另一方面，不能这么粗略地描述任务，要详细描述两人各负责哪部分（CFG）设计、各负责哪部分（ARM 汇编）编程，以后作业要求相同。
- 不要用 GCC 等编译器生成 C 程序对应的汇编程序直接交上来！可学习 GCC 生成的其他 C 程序的汇编程序，仿照着编写自己 SysY 程序的汇编程序。

期望： 鼓励有余力的同学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，并通过 Bison 进行实验。

2 参考流程

2.1 定义你的编译器

这一部分作业的主要要求是了解你的编译器所支持的 SysY 语言特性，如支持何种数据类型 (int 等)，支持变量声明，赋值语句，复合语句，if 分支语句，以及 while/for 循环，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。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。

2.1.1 上下文无关文法

上下文无关文法是一种用于描述程序设计语言语法的表示方式。一般来说，一个上下文无关文法（context-free grammar）由四个元素组成：

（1）一个终结符号集合 V_T，它们有时也称为“词法单元”。终结符号是该文法所定义的语言的基本符号的集合。

（2）一个非终结符号集合 V_N，它们有时也称为“语法变量”。每个非终结符号表示一个终结符号串的集合。

（3）一个产生式集合 $\mathscr{P}$，其中每个产生式包括一个称为产生式头或左部的非终结符号，一个箭头，和一个称为产生式体或右部的由终结符号及非终结符号组成的序列。产生式主要用来表示某个语法构造的某种书写形式。如果产生式头非终结符号代表一个语法构造，那么该产生式体就代表了该构造的一种书写方式。

（4）指定一个非终结符号为开始符号 S。

因此，上下文无关文法可以通过 (V_T,V_N,$\mathscr{P}$,S) 这个四元式定义。在描述文法时，我们将数位、符号和黑体字符串看作终结符号，将斜体字符串看作非终结符号，以同一个非终结符号为头部的多个产生式的右部可以放在一起表示，不同的右部之间用符号 | 分隔。

上下文无关文法无论是对课程理论内容的学习还是之后实践上机作业都是十分重要的，希望同学们能够认真学习并扎实掌握。

2.1.2 CFG 描述 SysY 语言特性举例

CFG 设计部分可以参考BUAA-MiniSysY中的详细设计。

1. 变量声明

$$type \rightarrow \textbf{int} \mid \textbf{float} \mid \textbf{double} \mid \textbf{char}$$

$$idlist \rightarrow idlist, \textbf{id} \mid \textbf{id}$$

$$decl \rightarrow type\ idlist$$

其中 id 表示标识符，type 代表变量类型，idlist 代表标识符列表，decl 代表声明语句。

2. 赋值语句

$$digit \rightarrow number\ digit$$

$$number \rightarrow \textbf{0}|\textbf{1}|\textbf{2}|\textbf{3}|\textbf{4}|\textbf{5}|\textbf{6}|\textbf{7}|\textbf{8}|\textbf{9}$$

$$unary-expr \rightarrow\ digit|\textbf{id}$$

$$assign-expr \rightarrow unary-expr = assign-expr | logical-expr$$

其中 num 代表数字字符，digit 代表数字，logical-expr 为逻辑表达式，unary-expr 为一元表达式，assign-expr 为赋值表达式，这里最后一个产生式第二个右部为逻辑表达式的原因是因为逻辑表达式的逻辑与和逻辑或优先级高于赋值表达式但却低于关系表达式的比较和算术表达式的各种运算。同学们写 CFG 时需格外注意优先级对 CFG 所带来的影响，经典例子为加减和乘除的 CFG（可查看龙书 P30）。

3. 循环语句及分支语句

$$stmt \rightarrow \textbf{if}\ (expr)\ stmt\ \textbf{else}\ stmt$$

$$stmt \rightarrow \textbf{while}\ (expr)\ stmt$$

$$stmt \rightarrow \textbf{for}\ (expr; expr; expr)\ stmt$$

其中 stmt 为语句，expr 为表达式。

4. 函数定义

$$funcdef \rightarrow type\ funcname(paralist)\ stmt$$

$$paralist \rightarrow paralist,\ paradef \mid paradef \mid \epsilon$$

$$paradef \rightarrow type\ \textbf{id}$$

其中 paralist 代表参数列表，paradef 代表参数声明，funcname 代表函数名，funcdef 代表函数声明语句。

2.2 汇编编程

我们用下面 C 代码为例来介绍 arm 汇编编程。

```c
#include<stdio.h>

int a = 0;
int b = 0;

int max(int a, int b) {
    if(a >= b) {
        return a;
    } else {
        return b;
    }
}

int main() {
    scanf("%d %d", &a, &b);
    printf("max is: %d\n", max(a, b));
    return 0;
}
```

你可以在这里，或者这里（简易版）获得你需要了解的 arm 架构的知识。它们可能包括，区分 arm 与 thumb 模式（我们应不会使用 thumb 模式）、理解 arm 架构各寄存器（与各状态位）的含义、了解要使用的指令集、理解函数栈是如何增长的，和一些必要的汇编代码编写技巧。

上一任笔者估计，若您对 arm 完全不了解，那么大概需要 3 小时的专注时间以理解你需要的全部知识。

代码示例

```
        .arch armv5t
        @ comm section   save global variable without initialization
        .comm    a, 4    @global variables
        .comm    b, 4
        .text
        .align    2
        @ rodata section  save constant
        .section    .rodata
        .align    2
    _str0:
        .ascii    "%d %d\0" @\000 is also one representation for `null character`
        .align    2
    _str1:
        .ascii    "max is: %d\n"
        @ text section   code
        .text
        .align    2

        .global    max
    max:  @ function  int max(int a, int b)
        str    fp, [sp, #-4]!   @ pre-index mode, sp = sp - 4, push fp
        mov    fp, sp
        sub    sp, sp, #12      @ allocate space for local variable
        str    r0, [fp, #-8]    @ r0 = [fp, #-8] = a
        str    r1, [fp, #-12]   @ r1 = [fp, #-12] = b
        cmp    r0, r1
        blt    .L2
        ldr    r0, [fp, #-8]
        b    .L3
    .L2:
        ldr    r0, [fp, #-12]
    .L3:
        add    sp, fp, #0
        ldr    fp, [sp], #4     @ post-index mode, pop fp, sp = sp + 4
        bx    lr                @ recover sp fp pc
        @ do you know the difference between `bx` and `bl`?
        @ and if max function is non-leaf, what should we do with the `lr` register?
        .global    main
    main:
        push    {fp, lr}
        add    fp, sp, #4
        ldr    r2, _bridge      @ r2 = &b
        ldr    r1, _bridge+4    @ r1 = &a
        ldr    r0, _bridge+8    @ *r0 = "%d %d\000"
        bl    __isoc99_scanf    @ scanf("%d %d", &a, &b)
        ldr    r3, _bridge+4    @ r3 = &a
        ldr    r0, [r3]         @ r0 = a
        ldr    r3, _bridge      @ r3 = &b
        ldr    r1, [r3]         @ r1 = b
        bl    max
        mov    r1, r0           @ r1 = r0
        ldr    r0, _bridge+12   @ *r0 = "max is: %d\0"
```

```
53          bl    printf            @ printf("max is: %d", max(a, b));
54          mov    r0, #0
55          pop    {fp, pc}         @ return 0
56
57      _bridge:
58          .word    b
59          .word    a
60          .word    _str0
61          .word    _str1
62
63          .section    .note.GNU-stack,"",%progbits  @ do you know what's the use of this :-)
```

代码说明

这其中有一系列的指令是编译器指令，作用是告知编译器要如何编译，通常以. 开始，其他指令则为汇编指令。

对每个函数的声明，观察可以发现一般首先为

```
.text
.global functionname
.type functionname, %function
```

即声明为代码段，将函数名添加到全局符号表中，声明类型为函数。

对全局变量与常量的声明，示例中已经给得比较详细，另外对于数组的使用，可以看到在声明时是毫无特殊的，而在使用时地址则为 *varname+offset*，其中偏移量即为数据类型大小乘个数。

另外值得说明的是，在进行函数调用时，一般前四个函数参数使用 r0-r3 号寄存器进行传参，其余参数压入栈中进行传参，往往按照从右至左的顺序逐个压栈；在函数返回时，一般来讲默认将函数的返回值放到 r0 寄存器中。

我们可以发现汇编与 C 的不同：汇编的语言要素就是“标签”（指示地址）、寄存器移动/计算指令。尤其标签的灵活使用：上述汇编代码中利用 `_bridge` 标签，“桥接”了在 C 代码中隐性的全局变量的地址。

某前辈曾说，编程语言理论，建立在“组合”之上——组合意味着复用，意味着抽象。在理解汇编代码时，我们希望将“理解其抽象、其作为整体的语义”作为思考目标（操作系统课或许会接触一些乍一看难理解汇编代码）；在编写程序时，也常常是自顶向下的思维过程。

代码逐行解析

1. 定义目标架构：`.arch armv5t` 表示使用 ARMv5t 指令集。
2. 定义数据区：`.comm a, 4` 定义了一个全局变量 a，大小为 4 字节。`.comm b, 4` 定义了一个全局变量 b，大小为 4 字节。
3. 定义文本区：`.text` 表示接下来是代码区。
4. 对 rodata 数据区进行对齐：`.align 2` 表示下一个数据项的地址应该是对齐的，且偏移量应为 2 字节。
5. 定义 rodata 数据区：`.section .rodata` 表示接下来是 rodata 数据区。
6. 定义常量字符串：`.ascii "%d %d\0"` 定义了一个包含两个整数格式化字符串的常量字符串，用于在后续的`scanf`和`printf`函数中使用。

7. 定义文本区：`.text` 表示接下来是代码区。
8. 定义全局变量 max：`.global max` 表示将变量 max 声明为全局变量。
9. max 函数实现：
   - 保存寄存器：`str fp, [sp, #-4]!` 表示将寄存器 fp 压入栈中，同时更新 sp 指针。
   - 设置寄存器：`mov fp, sp` 表示将 sp 指针设置为当前栈指针。
   - 分配栈空间：`sub sp, sp, #12` 表示为局部变量分配 12 字节的栈空间。
   - 保存参数：`str r0, [fp, #-8]` 表示将参数 a 压入栈中，偏移量为-8。
   - 保存参数：`str r1, [fp, #-12]` 表示将参数 b 压入栈中，偏移量为-12。
   - 比较参数：`cmp r0, r1` 表示比较参数 a 和 b 的大小。
   - 分支：`blt .L2` 表示如果 a 小于 b，则跳转到标签.L2。
   - 返回最大值：`ldr r0, [fp, #-8]` 表示将参数 a 从栈中弹出，并赋值给寄存器 r0。
   - 分支：`b .L3` 表示如果 a 大于等于 b，则跳转到标签.L3。
   - 返回最大值：`ldr r0, [fp, #-12]` 表示将参数 b 从栈中弹出，并赋值给寄存器 r0。
   - 恢复栈空间：`add sp, fp, #0` 表示将栈指针恢复到原始值。
   - 恢复寄存器：`ldr fp, [sp], #4` 表示从栈中弹出 fp 寄存器，并更新 sp 指针。
   - 返回：`bx lr` 表示恢复 pc 指针，并返回。
10. main 函数实现：
    - 保存寄存器：`push {fp, lr}` 表示将寄存器 fp 和 lr 压入栈中，并更新 sp 指针。
    - 定义局部变量：`add fp, sp, #4` 表示为局部变量分配 4 字节的栈空间。
    - 加载常量字符串：`ldr r2, _bridge` 表示将常量 b 的地址赋值给寄存器 r2。
    - 加载常量字符串：`ldr r1, _bridge+4` 表示将常量 a 的地址赋值给寄存器 r1。
    - 加载常量字符串：`ldr r0, _bridge+8` 表示将常量字符串 _str0 的地址赋值给寄存器 r0。
    - 调用函数：`bl __isoc99_scanf` 表示调用scanf函数读取两个整数参数 a 和 b。
    - 加载参数 a：`ldr r3, _bridge+4` 表示将常量 a 的地址赋值给寄存器 r3。
    - 加载参数 b：`ldr r1, [r3]` 表示将 r3 地址处的值（即 a 的值）赋值给寄存器 r1。
    - 调用函数：`bl max` 表示调用max函数计算 a 和 b（分别存在 r0，r1）的最大值，返回结果在 r0（默认返回 r0）。
    - 保存结果：`mov r1, r0` 表示将结果赋值给寄存器 r1。
    - 加载常量字符串：`ldr r0, _bridge+12` 表示将常量字符串地址 _str1 赋值给寄存器 r0。
    - 调用函数：`bl printf` 表示调用printf函数输出，r0 为字符串参数，r1 为 max 结果，两者作为函数调用的参数。
    - 返回 0：`mov r0, #0` 表示将 0 赋值给寄存器 r0，作为 main 函数的返回值。
    - 恢复寄存器：`pop {fp, pc}` 表示将 fp 寄存器弹出并恢复，并更新 pc 指针。
11. 定义_bridge的符号表项

- `.word b`：这是一个引用，表示符号b的 32 位 (.word) 地址。
- `.word a`：表示符号a的地址。
- `.word _str0`：表示符号_str0的地址。
- `.word _str1`：表示符号_str1的地址。
- `.section .note.GNU-stack,"",%progbits`：这一行代码是用于 GNU Assembler（GAS）的，通常出现在汇编语言源文件中。它用于指定一个特殊的节（section）`.note.GNU-stack`。这个节没有实际的代码或数据，而是用于向链接器和操作系统传递有关程序堆栈的信息。具体来说，`.note.GNU-stack`用于控制生成的可执行文件的堆栈是否是可执行的。在默认情况下，Linux 操作系统通常会将堆栈标记为不可执行，这有助于阻止堆栈执行攻击。这是一个安全性措施，旨在防止攻击者通过注入可执行代码到程序堆栈来攻击程序。

代码测试 当你写完汇编程序（比如 `example.S`）后，使用下述指令即可测试它。

```
arm-linux-gnueabihf-gcc example.S -o example
qemu-arm ./example
```

当然，你可以让测试过程更“自动化”些，将它加入 Makefile，并利用管道测试默认样例、生成结果。[1]。

```
1 .PHONY:test,clean
2 test:
3     arm-linux-gnueabihf-gcc example.S -o example.out
4     qemu-arm ./example.out
5 clean:
6     rm -fr example.out
```

[1]若您还没有配置好 arm 环境，请参考《编译器开发环境》实验指导。