预备工作 2——定义你的编译器 & 汇编编程

杨侯哲 李煦阳

October 2020

目录

1 实验描述

基于“预备工作 1”，继续

1. 设计你的源语言：你所使用的编译器支持哪些主要的 C(C++) 语言特性？在此基础上定义你的编译器支持的 C 语言子集——学习教材第 2 章以及第 2 章讲义中的 2.2 节，用上下文无关文法描述你的 C 语言子集。你应该尽可能参考SysY的 C 语言定义、在本学期大作业中实现其中特性。
2. 理解你的目标语言：对某个 C 程序 (如“预备工作 1”给出的阶乘或斐波那契)，编写等价的 (x86 或 arm 的) 汇编程序，用汇编器生成可执行程序，调试通过、能正常运行。这个程序也应该包含尽可能全面的语言特性。

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 C 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？其正确性如何定义、如何验证？

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 C 程序。

穷举所有 C 程序是不可能的，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, $\cdots$)，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：

- 撰写研究报告 (要求同前)，要包含你对源语言、目标语言的理解、你对编译过程的理解
- 不要用 gcc 生成 C 程序对应的汇编程序直接交上来，可用 gcc 生成其他 C 程序的汇编程序，仿照着编写自己这个 C 程序的汇编程序。
- 程序包含的语言特性尽可能全面，比如输入输出、全局变量、子过程调用。

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 C 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 C（C++）语言特性，如支持何种数据类型 (int,double 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。从中选取重要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 C 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 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构造的某种书写形式。如果产生式头非终结符号代表一个构造，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,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 C 语言特性举例

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

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

$$paralist \rightarrow paralist, paradef | paradef | \epsilon$$

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

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

指导书中汇编代码具有与以下 c 代码相同的功能。

```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;
}
```

2.2.1 x86 架构汇编编程

这一部分需要一些简单的汇编语言知识，如 movl，pushl 等等指令的运用，我们作业用的 GCC 编译器默认会将指令编译为 AT&T 格式，同学们可以通过翻译样例代码到汇编代码来尝试理解其中的语言格式。如果之前同学们学习过 intel 格式的汇编指令，实际上两者之间的差异并不大，想要详细了解的同学可以看这里或自行查阅。

另外，如果想要完整了解需要用到的汇编指令的话，可以查看这里。

代码示例

```
# 函数 max
    .text
    .globl  max
    .type   max, @function
max:
# if(a >= b)
    movl    4(%esp), %eax
    cmpl    8(%esp), %eax
    jl  L2
# return a;
    movl    4(%esp), %eax
    jmp L3
L2:
# return b;
    movl    8(%esp), %eax
L3:
    ret

# bss 段 存储全局变量
    .bss
# comm 声明未初始化的数据区域 zero 声明初始化的数据区域
# comm 用法
#   .comm a,4
#   .comm b,8
    .align  4 # 令数据地址按 4 对齐
a:
    .zero   4
    .align  4
b:
    .zero   4
# 开辟数组
    .align  4
c:
```

```
34      .zero   8
35
36  # rodata 段 存储常量
37      .section    .rodata
38  STR0:
39      .string "%d %d"
40  STR1:
41      .string "max is: %d\n"
42
43  # 主函数
44      .text
45      .globl  main
46      .type   main, @function
47  main:
48  # scanf("%d %d", &a, &b);
49      pushl   $b  # 从右向左压入参数
50      pushl   $a
51      pushl   $STR0
52      call    scanf
53      addl    $12, %esp # 不再需要参数
54  # printf("max is: %d\n", max(a, b));
55      movl    b, %edx
56      movl    a, %eax
57      pushl   %edx
58      pushl   %eax
59      call    max
60      addl    $8, %esp
61      pushl   %eax
62      pushl   $STR1
63      call    printf
64      addl    $8, %esp
65  # return 0;
66      movl    $0, %eax
67      ret
68  # 可执行堆栈段 不清楚具体含义的话建议保留
69      .section    .note.GNU-stack,"",@progbits
```

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，示例中已经给的比较详细，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，一般来讲默认将函数的返回值放到 eax 寄存器中，而在进入函数时候对于多个参数往往按照从右至左的顺序逐个压栈。

代码测试 编写完汇编代码之后当然要经过测试，将汇编代码通过 GCC 编译器编译为可执行文件，而后观察是否可以运行即可，简单来说通过如下命令

```
gcc main.s -m32 -o main.out # m32 表示按照 32 位架构编译
qemu-i386 main.out # 32 位程序直接运行需要一系列库，可以通过 qemu 模拟出 i386 架构来测试程序
```

当然如果愿意的话你也可以利用自动化的 Makefile 来进行测试，在这里我给出了利用 gcc 获得参考的编译代码的指令，里面参数的作用同学们可以自行查询并进行修改尝试。

```
.PHONY:test,all,clean,clean-all
test:
    gcc main.s -m32 -o main.out
    qemu-i386 main.out
all:$(subst .c,.s,$(wildcard *.c))
%.s:%.c
    gcc $< -m32 -std=c99 -S -o $@ -O0 -fno-asynchronous-unwind-tables -fno-builtin
    ↪  -fno-common -fno-ident -finhibit-size-directive -fno-pie -march=i386
clean-all:$(subst .c,-del,$(wildcard *.c)) clean
clean:
    rm -fr main.out
%-del:%.c
    rm -fr $(basename $<).s
```

2.2.2 arm 架构汇编编程

你可以在这里获得你需要了解的 arm 架构的全部知识。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。

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

```
1         .arch armv5t
2         .comm    a, 4    @global variables
3         .comm    b, 4
4         .text
5         .align    2
6
```

```
7         .section    .rodata
8         .align    2
9     _str0:
10        .ascii    "%d %d\0" @\000 is also one representation for `null character`
11        .align    2
12    _str1:
13        .ascii    "max is: %d\n"
14        .text
15        .align    2
16
17        .global    max
18    max:  @ int max(int a, int b)
19        str    fp, [sp, #-4]!  @ `push fp` along with `modifying sp`!
20        mov    fp, sp   @ mov is actually an simplification for add
21        sub    sp, sp, #12 @ allocating space for 3 things...
22        str    r0, [fp, #-8] @ r0 = [fp, #-8] = a
23        str    r1, [fp, #-12] @ r1 = [fp, #-12] = b
24        cmp    r0, r1
25        blt    .L2
26        ldr    r0, [fp, #-8]
27        b    .L3
28    .L2:
29        ldr    r0, [fp, #-12]
30    .L3:
31        add    sp, fp, #0
32        ldr    fp, [sp], #4
33        bx    lr @ recover sp and fp and pc
34        @ do you know the difference between `bx` and `bl`?
35        @ and if max function is non-leaf, what should we do with the `lr` register?
36        .global    main
37    main:
38        push    {fp, lr}
39        add    fp, sp, #4
40        ldr    r2, _bridge    @ r2 = &b
41        ldr    r1, _bridge+4 @ r1 = &a
42        ldr    r0, _bridge+8 @ *r0 = "%d %d\000"
43        bl    __isoc99_scanf @ scanf("%d %d", &a, &b); what if we have more arguments?
44        ldr    r3, _bridge+4 @ r3 = &a
45        ldr    r0, [r3] @ r0 = a
46        ldr    r3, _bridge @ r3 = &b
47        ldr    r1, [r3] @ r1 = b
48        bl    max
49        mov    r1, r0 @ r1 = r0
50        ldr    r0, _bridge+12 @ *r0 = "max is: %d\0"
51        bl    printf @ printf("max is: %d", max(a, b));
52        mov    r0, #0
53        pop    {fp, pc} @ return 0
54
55    _bridge:
56        .word    b
57        .word    a
58        .word    _str0
59        .word    _str1
60
61        .section    .note.GNU-stack,"",%progbits  @ do you know what's the use of this :-)
```

代码说明：

代码中已尽可能详细的注释了每个语句在 c 代码中的等价对应，因此不再对具体语句做解释。

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。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]

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

[1] 若您还没有配置好 arm 环境，请参考 `util` 实验指导。