编译原理研讨课实验 PR003 实验报告

一、任务说明

在前面的实验中，我们已经完成了 CACT 编译器的词法、语法分析工作，完成了语法树的构建。本次实验中，需要分析源代码，生成中间代码，并最终生成 RISC-v 汇编代码，要求能对测试用 CACT 代码编译出能正确执行的汇编代码。

二、成员组成

吕恒磊，康家锐，王东宇

三、实验设计

1. 设计思路

本次实验主要内容分为生成中间代码和生成汇编代码两部分，下面将分别介绍其实现思路。

中间代码我们采用三地址代码，在通过遍历语法树，结合我们在语法中设计的 SDT，递归地生成，以四元式的形式进行存储。我们首先要在语法树的各个节点设置必要的属性，用于保存我们需要传递的信息，然后根据设计好的 SDT 在语法树合适的位置添加操作，结合理论课中讲解的翻译方法，最终将语法树完善。在我们的设计中，中间代码的生成与语法分析是同时进行的。由于这两种功能的实现使用了两套不会相互干扰的逻辑，他们不会造成编译器的紊乱。

生成汇编代码部分，我们把三地址代码作为输入，并划分为多个部分翻译。首先是程序开头的全局变量、常量的声明，之后是各个函数的翻译，最后会附加浮点常量的保存。由于每个函数结构相似，并且各个函数的代码不会相互干扰，我们会以函数为单位逐个翻译。翻译的主体部分会根据输入的三地址代码操作符的不同分别进行处理，最终生成对应的汇编代码。至于具体的翻译方法，我们参考了 RISC-V 手册，针对不同类型的三地址代码生成对应的汇编代码序列。我们安装了交叉编译环境，自行设计样例代码进行编译比对，保证翻译合法。最后，为了保证编译器生成的汇编代码的正确性，我们还安装了 spike 和 pk 模拟 RISC-V 的运行环境。

2. 实验实现

在三地址代码的具体实现中，我们设计了如下数据结构，用以储存三地址代码，保存必要的信息，用于汇编代码的翻译。

```
class TACoperand
{
    public:
    int type;           // 类型
    void* data;         // 数据
    std::string temp;   // 变量暂存名
    int size;           // 类型宽度
    bool imm;           // 是否为立即数，默认为1，变量为0


    TACoperand(){};

    void init(int type, bool imm)…

    int copyImm(void* src)…

    int copyName(std::string name)…
};
```

图 1

上图中的数据结构用来存放三地址代码中单个操作数的信息，并带有部分相关的函数。

我们为每个操作数保存了数据类型、变量名、宽度、立即数标志等信息。由于数据类型不同，数据使用无类型指针存储，数据存储的地址会在调用 init 函数时根据数据类型宽度分配。copyImm 函数针对立即数类型的操作数使用，可以把数值存放在 data 域中。copyName 函数针对变量操作数使用，可以把变量名存在 temp 域中。

```
// 四元式
struct TACline
{
    TACop op;
    TACoperand arg1;
    TACoperand arg2;
    TACoperand result;
};
```

图 2

上图中数据结构用来存放三地址代码中的一行，包括操作符、两个操作数、结果。操作符包括空操作符、赋值、数组访问、各种双目运算符、各种单目运算符、各种跳转标志、设置参数、调用函数、函数返回等，共 20 多种。两个操作数和运算结果都用上面定义的 TACoperand 保存，在部分操作下可能这三者不全有效，会根据操作符的不同进行处理。例如，在加法语句中两个操作数和结果都会被用到，而赋值语句中只会用到结果 result 和第一个参数 arg1。

```
class threeAdressCode
{
    public:
    int num_lines;      // 三地址码行数
    int cur_line;
    TACline* lines;     // 存放三地址代码

    threeAdressCode(int num = 1024)…

    void copyline(TACline *line1, TACline *line2){…

    void addline(TACline line)…

    void addlines(TACline lines[], int num)…

    // 合并两个三地址代码段
    void codecat(threeAdressCode code)…

    void printOperand(TACoperand arg){…

    void printTAC(){…
};
```

图 3

上图中的数据结构用来存放一段三地址代码，其中可能包含若干行。num_lines 域保存该数据结构当前已分配的空间可容纳行数。我们在构造函数中会默认申请 1024 行，后续运行中如果溢出，则会动态地分配空间，保证能容纳下所有的三地址代码。cur_lines 域中保存该数据结构当前已存储三地址代码行数，lines 域就是真正储存三地址代码的数组指针。关于后续的几个函数，copyline 用于复制一个三地址行的内容，会在 addline 和 addlines 中被调用，后面这两个函数用于向数据结构中添加单个或多个三地址代码行。codecat 函数

用于合并两个三地址代码段，即将参数中的三地址代码复制到本数据结构尾部。两个 print 函数分别用于打印某个操作数的信息，以及本数据结构中保存的全部三地址代码，可以直观地检查我们生成的三地址代码的内容，便于检错。

中间代码的生成与 SDT 密切相关。我们设计的 SDT 仿照理论课实现，主要内容包括常量、变量的声明，函数的定义的处理，各种语句的处理。各种语句的处理是最为复杂的一部分，不仅要处理种类纷杂的不同语句类型，还要考虑与后续生成汇编代码部分的对接，让生成的三地址代码正确简洁，同时能更容易地继续翻译。例如，

```
    a = b + c;  // a, b, c 均为标量
=>  t0 = b;
          t1 = c;
          t2 = t0 + t1;
          a  = t2;

    a = b + c;  // a, b, c 均为数组, 长度为 n
=>   int i = 0;
    while(i < n)
    {
        a[i] = b[i] + c[i];
        i = i + 1;
    }
=> 再按规则生成三地址代码
```

通过研究 g++交叉编译器生成的汇编代码，并参考 RISC-V 手册，我们得知 RISVC-V 代码应大致分为全局变量、常量声明，函数定义，以及附在末尾的浮点常量存储三部分。在生成汇编代码的时候，考虑到各个函数结构相似，并且内容不会相互干扰，我们以函数为单位分别进行翻译（通过 genFunc 函数），再额外设置两个函数对全局变量部分以及浮点数常量部分进行输出。在对一个函数进行翻译时，以一个循环中嵌套一个 switch 为主体，依次把三地址代码中的每一行代码根据 op 进行判别，输出对应的指令段。

函数间的参数传递方面，我们按照 RISC-v 标准函数调用 API 进行，在调用时通过设置参数指令将对应数据放入特定的几个寄存器，在被调用函数的翻译时，我们从语法树中获取其参数信息，并传入 genFunc 函数中，从而生成相应的取参数指令，将各个参数依次从特定寄存器中取出。

翻译的一个具体要点是空间的分配，我们也设置了一个数据结构和两个相关函数来进行空间的分配、记录及查询。

```
9   struct address
10  {
11      list_head list;
12      std::string id;
13      int off;
14      int type;//1是寄存器，2内存，3全局
15      std::string addr;
16      int width;
17  };
```

图 4

这个数据结构通过 list 头进行连接，方便进行遍历查询，主要存储变量名、变量种类、地址信息、偏移量和位宽。address 特别采取了 string 类来存储，方便返回一个可以直接到 fprintf 里能用的串。查找工作由 lookup 函数实现，其会遍历所有 address 结构直到找到 id 相符的返回对应的 addr，如果没有找到则会返回“notfound”以标识。

另外翻译部分在维护的表还有下图中的浮点数常量表和函数参数表，分别用来记录生成最后的浮点数常量部分和在函数间进行参数传递。

```
19  struct floatc
20  {
21      list_head list;
22      std::string label;
23      int type;
24      void * data;
25  };
26
27  struct param{
28      int btype;
29      std::string name;
30  };
31
32  struct paramList{
33      int numParams;
34      struct param * p;
35  };
```

图 5

3. 其他

我们的编译器会将生成的汇编文件放在根目录下的 output 目录中。

四、总结

1. 实验结果总结

我们已经对测试样例进行了验证，可以正确生成.s 文件，生成的文件也可以通过 riscv-isa-sim 和 riscv-pk 生成可执行文件并正确运行。因此样例中涉及到的内容我们的编译器已经都可以实现了。此外，我们也自己设计了一些测试样例，与 g++交叉编译器得到的汇编代码对比，再模拟运行，基本能确保编译器工作的稳定性和正确性。

2. 分成员总结

康家锐：

通过这次实验，或者说三次全部的实验，我们确实完成了编译的全部过程，对编译该如何进行，有了一个完整的认识。虽然付出了不少的努力（cact 确实工作量相当大），但是确实得说也有着相当的收获，一些理论课上认识不足的地方，在具体的思考过代码以后，都有了更为准确的认识。实验过程中，我们也遇到了很多问题，不止是一些细节处的小 bug 很多，比较大的实现方向也有修改或补充，不过好在最后都解决了。

王东宇：

编译原理的实验课上我们确确实实地完成了一个编译器。尽管它只能处理语法逻辑简单的 CACT 语言，并且它的性能也较低，但我们完整地完成了编译器设计的全部过程，并把理论课上学到的知识应用到了实践中。同时，我们在设计编译器的过程中也能体会到理论课上所学到的编译器设计方法的初衷，能略略体会到前人的心路历程，加深对编译原理这门课的理解。这样想来，与 CACT 相伴的那些无眠的夜也不枉辛苦一遭。

吕恒磊：

CACT 实验三的工作量较大，超出了之前的预期，时间分配因此出现了一些问题。在本次实验中，我负责将源代码转换成三地址代码的部分，主要难点在于设计 SDT 和代码实现。SDT 大部分可以仿照课本对应内容，但是有些部分由于文法不同，需要自己设计 SDT。其次是代码实现工作量较大，并且调试较为繁琐，各个节点之间属性的传递内容繁杂，容易出错。总而言之，本实验加深了我对编译原理理论课知识的理解，并且锻炼了工程能力，十分有收获。