重庆大学编译原理课程实验报告

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | | **20计算机科学与技术（卓越）01** | | **姓名** | **黄昊** |
| **实验题目** | **编译器设计与实现** | | | | |
| **实验时间** | **2023/06/08** | | **实验地点** | **DS3301** | |
| **实验成绩** |  | | **实验性质** | **□验证性 □设计性 ■综合性** | |
| 教师评价：  □算法/实验过程正确；□源程序/实验内容提交 □程序结构/实验步骤合理；  □实验结果正确； □语法、语义正确； □报告规范；  其他：  评价教师签名： | | | | | |
| 一、实验目的  以系统能力提升为目标，通过实验逐步构建一个将类C语言翻译至汇编的编译器，最终生成的汇编代码通过GCC的汇编器转化为二进制可执行文件，并在物理机或模拟器上运行。实验内容还包含编译优化部分，帮助深入理解计算机体系结构、掌握性能调优技巧，并培养系统级思维和优化能力。 | | | | | |
| 二、实验项目内容  本次实验将实现一个由 SysY (精简版 C 语言，来自 <https://compiler.educg.net/>) 翻译至 RISC-V 汇编的编译器，生成的汇编通过 GCC 的汇编器翻译至二进制，最终运行在模拟器 qemu-riscv 上  实验至少包含四个部分: 词法和语法分析、语义分析和中间代码生成、以及目标代码生成，每个部分都依赖前一个部分的结果，逐步构建一个完整编译器  **实验一**：词法分析和语法分析，将读取源文件中代码并进行分析，输出一颗语法树  **实验二**：接受一颗语法树，进行语义分析、中间代码生成，输出中间表示 IR (Intermediate Representation)  **实验三**：根据 IR 翻译成为汇编  **实验四(可选)**：IR 和汇编层面的优化 | | | | | |
| 1. 实验内容实现 2. **实现哪些内容**  * 实验一与实验二的内容完全实现（能够通过所有测试点） * 实验三实现了大部分内容，但没有处理参数超过八个参数寄存器(a0-a7)的情况（不考虑浮点数的那8个，而且只有一个测试点），浮点数有处理但是没过对应测试点。以及剩下存在的未知bug导致两个测试点不能通过过。 * 对于优化方案，有尝试采取最简单的策略（寄存器当缓存用），寄存器和栈空间中数据换入换出的逻辑主要由expireRegData和loadMemData两个函数控制。  1. **IR库的使用，如何使用静态库链接，如何使用源代码来构建库？结合CMakelist说明。**   考虑cpp生成可执行文件的四个过程：预处理，编译，汇编，链接。如果采用源代码构建库，那么修改CMakeList.txt如下：  *# link\_directories(./lib)*  *# --------------------- from src ---------------------*  aux\_source\_directory(./src/ir IR\_SRC)  add\_library(IR ${IR\_SRC})  aux\_source\_directory(./src/tools TOOLS\_SRC)  add\_library(Tools ${TOOLS\_SRC})  也就是不使用添加库的构建规则link\_directories，而是加入源代码。那么，此时src下的ir和tools下面的源文件将会参与整个构建过程；  如果使用静态库链接，则修改如下：  link\_directories(./lib)  *# --------------------- from src ---------------------*  *# aux\_source\_directory(./src/ir IR\_SRC)*  *# add\_library(IR ${IR\_SRC})*  *# aux\_source\_directory(./src/tools TOOLS\_SRC)*  *# add\_library(Tools ${TOOLS\_SRC})*  此时，源代码将不会参与构建过程，而是由预先编译好（即实验提供的库）后生成的静态库libIR.a和libTools.a在整个项目构建的链接阶段参与项目构建。   1. **在IR中你如何处理全局变量的，这样的设计在后端有什么好处？后端中如何处理全局变量？**   对于全局变量的识别，关注该文法：  CompUnit -> (Decl | FuncDef) [CompUnit]  AST的根就是CompUnit。考虑scope\_stack的动态变化：识别到Block节点压栈，退出Block节点出栈。那么，我们完全可以在最开始压一个栈，之后处理的时候，如果栈大小为1，那么就可以判断这是个全局作用域。之后在这个节点处理ir。  等到整个AST分析结束后。所有用于全局变量初始化的ir将会被放入一个函数(globalFunc)来处理全局变量初始化，并且全局变量作用域的变量将会放入ir::program.globalVal的结构中。在main函数的第一条指令插入t0, call global的ir。这样做的好处是可以将全局变量初始化的操作作为一个函数来处理，简化了实现。同时也为后端的优化提供了可能性。  后端中处理全局变量的方式：一个是保留了call global的操作（如果有更好的优化也可以对这个函数作特殊处理，毕竟用rv指令作初始化也会存在花销），另外一个，全局变量用.data来标识，表示全局变量被放在data段，可读可写。后续访问全局变量用以下方式来获得全局变量的地址：  lui t0, %hi(globalVar)  addi t0, t0, %lo(globalVar)  或  la t0, globalVar  前者lui,addi两者的组合是riscv gcc提供的方式，lui操作表示将立即数加载到t0的高12位，%hi (globalVar)表示获取globalVar变量地址的高12位，%lo则是20位；后者la是伪指令。   1. **如何处理数组作为参数的情况，为什么可以这样做？**   首先明确：数组本身可以视为指针，下标的访问实质是指针的运算。  在前端中，数组作为参数需要注意这样一串文法：  FuncRParams -> Exp { ',' Exp }  Exp -> AddExp  AddExp -> MulExp { ('+' | '-') MulExp }  MulExp -> UnaryExp { ('\*' | '/' | '%') UnaryExp }  UnaryExp -> PrimaryExp  PrimaryExp -> LVal  LVal -> Ident {'[' Exp ']'}  其中FuncRParams表示函数被调用时传入的一系列参数，其中数组作为参数，会从LVal这个节点一路传递上去。因此在LVal节点使用getptr的ir获取指针，然后一路往上传就可以了。  然而在后端中，处理相对来说比较复杂：如果是数组作为参数，在传递参数时要先判断是全局还是局部，如果是局部，结合sp和offset把地址算出来传过去；全局就直接la然后传地址；如果该函数有数组作为入参，那么在这个函数，需要考虑这个数组（即指针）是什么情况：全局变量，局部变量（排除入参的情况），以及入参。如果是全局变量，la即可；局部变量的话，利用sp+offset拿到地址；如果是入参，由于在函数初始化时，入参作为参数也会在运行时栈分配一个地址，那么如果来自于a0-a7这种参数寄存器，直接mv即可，如果存放在栈空间中，那么还需要用sp+offset算出指针存放的位置，然后lw出来才能得到这个指针地址。   1. **如何支持短路运算？**   一共有两处文法有短路运算，分别介绍。  ①LOrExp -> LAndExp [ '||' LOrExp ]  这个文法的短路是为真则跳过剩余运算。如：0||2 – 3||func()，因为2-3为真，那么整个式子为假，func()是得不到执行的。因此，为了支持短路运算，我们需要在LandExp和一大堆的LOrExp项之间加入跳转逻辑。即每次进行逻辑或运算后，判断结果，如果为真，则直接跳到末尾；否则跳到下一项的运算，直到算出为假，才跳到末尾。  ②LAndExp -> EqExp [ '&&' LAndExp ]  这个文法的短路是为假则跳过剩余运算。如：0&&func()，因为0为假，那么整个式子为假，func()是得不到执行的。因此，为了支持短路运算，我们需要在LEqExp和一大堆的LAndExp项之间加入跳转逻辑。即每次进行逻辑与运算后，判断结果，如果为假，则直接跳到末尾；否则跳到下一项的运算，直到算出为假，才跳到末尾。 | | | | | |
| 四、实验测试   1. 测试程序是如何运行的？执行了什么命令？你的汇编是如何变成RISV程序并被执行的？   **①测试程序如何运行**  一共有四个python脚本：build.py, run.py, score.py, test.py.其中： build.py：一键编译整个工程 run.py：命令行接收四个选项：s0,s1,s2,S，分别用来跑词法分析（生成一系列token），语法分析（生成json格式的AST），IR（执行）以及生成汇编（并链接上IO库并执行），生成的结果被放在/test/output下面 score.py:将run.py生成的结果与/test/ref相比较（利用diff工具），并给出最后得分。同样接受四个参数，其中s0选项与/test/ref/s0下的结果进行比较（词法分析），s1与/test/ref/s1下面的结果进行比较（语法分析），s2及s的结果与/test/ref/s2下面的结果进行比较。  其中需要注意的是，如果选择s2选项，则调用测评机的位置在run.py，score仅负责比对结果；如果选择S选项，那么则会在run.py生成汇编到/test/output下，score.py则负责与sylib-riscv-linux.a，将io库链接进来，然后使用riscv gcc跑最终结果，同样在/test/output下面生成。  test.py: 在功能上可以认为是build.py + run.py + score.py，不再赘述。  **②执行了什么命令**  build.py:  os.system("cd ../build && cmake .. && make")  调用CMake工具和makefile工具来构建整个项目  run.py:  cmd = ' '.join([compiler\_path, testcase\_dir + src, step, "-o", output\_dir + fname + "." + oftype])  遍历/test/testcase下的所有用例，拿到所有sy文件（也就是SysY源程序），然后调用/bin/compiler来跑一遍结果（s0为词法分析，一系列token，s1为语法分析，为json格式的AST，s2生成ir表示，S生成riscv指令），结果放在/test/output下面。step为运行python脚本时传进来的选项，需要注意的是，s2会被修改为-e选项，用来执行ir测评机。  score.py:  cmd = ' '.join(["diff", ref\_dir + file, output\_dir + file, '-wB']  代码里面的s0,s1,s2三个分支其实都在做一个事情：使用diff工具遍历比较output和ref下面的结果，比对完全相同则计分，否则不计分。其中s0比对token是否正确，s1比对json语法树是否正确，s2比对ir测评机跑的结果是否正确  比较特殊的是S（汇编）。首先执行了这条命令：  cmd = ' '.join(["riscv32-unknown-linux-gnu-gcc", output\_dir + file, "sylib-riscv-linux.a", '-o', exec\_file])  这条命令的用处是将生成的汇编与装有IO函数的静态库链接在一起，并最终生成可执行文件。下一步则是运行二进制可执行文件。注意到：  *if* os.path.exists(input\_file):      cmd = ' '.join([cmd, "<", input\_file])  cmd = ' '.join([cmd, ">", output\_file])  这段代码的作用是重定向输入输出。注意testcase下面有一些后缀为in的文件，这些是对应样例的输入。output\_file则被重定向到output文件夹下，进行后续的比对工作。  cmd = ' '.join(["diff", ref\_file, output\_file, '-wB'])  最后是调用diff工具进行结果比对，不再赘述。  **③汇编如何变成rv**  首先跑/bin/compiler生成risc-v汇编，然后和编译出的有IO函数的sylib-riscv-linux.a静态库链接起来，最后生成二进制可执行文件并运行。具体流程在上面已经介绍，不再赘述。 | | | | | |
| 五、实验总结   1. 实验过程中所遇到的问题及解决办法   遇到的问题：部分样例不能通过，在实验三中有的汇编程序能够正常生成，但执行会报错，最常见的，如segment fault。  解决办法：通过打印中间过程，逐步排查问题。常见的问题包括但不限于：  ①实验二中，ir测评机打开宏后往往提示非法源操作数，这种情况大多数是由于某处程序的类型声明错误，比较典型的是本该是intliteral被声明成int了，还有些情况是对ir库的不熟，如store不允许字面量作为存储的源操作数。  ②实验二中的一大堆细节问题，比如：main函数第一行要先call globalfunc，部分测试点的数组的初始化不涵盖所有情况（如int a[10] = {1,2}）,临时变量的重复，部分测试点的void函数没有返回值要自己返回，等等  ③对于个别难改的程序（逻辑复杂，使用gdb工具也难以查找的逻辑上的bug），考虑在SysY文件里面加入打印语句，同时将程序放入cpp文件，也加入对应的打印程序，然后对比打印的结果，然后逐步缩小范围。最后定位出在并查集那个测试点，测出函数返回名一样的bug导致的错误。  ④最常见的段错误，一般都是非法访存导致的结果。如果是实验二就查数组越界和不合法的指针操作，实验三就重点检查load,store,getptr以及全局变量相关的操作。   1. 对实验的建议   下面主要是针对实验二进行建议，建议的理由主要是难度和工作量相对于其他两个实验比较失衡。从时间上来说，实验一和实验三花的时间加起来都赶不上实验二，从难度上说，由于涉及到很多细节问题和需要考虑的核心问题很多，也是比较难的。所以可以考虑降低实验二的难度。比如：   * 从语法制导翻译的角度，给出更具体的文法（规定属性相关的动作），引导大家注意到本应该需要注意到的问题； * 详细地给出SysY语言的规范，比如点出void函数允许不带返回值（某个测试点确实没有return，忘了有哪些），再例如明确提示数组大小可以在编译时确定，不会出现运行时确定数组大小的用例，再比如数组的Initializer的格式，etc. * ir库重载一下拷贝构造函数和赋值函数，以及Operand的< * 可以考虑把这个实验拆成两个实验，前一个实验做一部分文法，后一个实验做剩余部分的文法 | | | | | |