重庆大学编译原理课程实验报告

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | |  | | **姓名** |  |
| **实验题目** | **编译器设计与实现** | | | | |
| **实验时间** |  | | **实验地点** |  | |
| **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/](https://compiler.educg.net/" \t "http://114.117.246.238:4000/_blank)) 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. 实现内容：   所有测试点的完整词法，语法，语义和汇编翻译支持和常量优化，寄存器分配等汇编优化策略。项目整体采用宏定义和函数调用结合的方式实现，在汇编层提供了虚拟寄存器，指令打印等功能，同时在高维数组，浮点参数溢出，局部变量溢出等目前的测试样例不包含的情况提供了一定的实现以及相应的assert检查，具有良好的可扩展性。   1. 静态库链接   对于引用的外部函数，既可以在编译时直接使用依赖库函数所在的cpp文件，也可以只提供依赖库函数对应的.a文件。编译器会自动检查本文件的符号表信息，确认其中的外部函数，然后遍历各个静态链接文件的符号表，查询目标函数名称并完成相应的地址重定位工作，最后生成完整的，包括了依赖库函数实现的可执行文件，区别于动态链接文件的依赖库在程序运行时由操作系统加载，不包含在可执行文件中。  # --------------------- from lib ---------------------  link libxx.a  # u should rename libxx-x86-win.a or libxx-x86-linux.a to libxx.a according to ur own platform  link\_directories(./lib)  # --------------------- from lib ---------------------   1. 全局变量的前后端处理：   在前端的语义分析过程中，全局变量和其他类型的变量基本不作区分。当语义分析结束后，程序将全局变量单独从语义分析过程的符号表中提取出来，保存为生成的Program的一部分：      for(auto it = symbol\_table.scope\_stack[0].table.begin() ; it != symbol\_table.scope\_stack[0].table.end() ; it++)      {          auto &ste = it->second;// 添加全局变量          if(ste.dimension.size())              program.globalVal.push\_back({ste.operand, ste.size});          else              // TODO2.8;              program.globalVal.push\_back({ste.operand, 0});      }  全局变量一定位于定义域栈的最底层，其定义位于ir\_program中。其中变量的名称和数据类型使用Operand类的成员变量val表示。当且仅当全局变量是数组时，数组长度maxlen的值不为0。      struct GlobalVal      {          ir::Operand val;          int maxlen = 0;     //为数组长度设计          GlobalVal(ir::Operand va);          GlobalVal(ir::Operand va, int len);  };  当IR程序在IR评测机中模拟运行时，评测机将会单独处理全局变量，为其中的全局数组开辟相应的内存空间并进行零初始化。      // init global variables      for(const auto& gte: program->globalVal) {          std::pair<std::string, Value> entry = {gte.val.name, {gte.val.type, 0}};          if (gte.maxlen) {              if (gte.val.type == Type::IntPtr) {                  entry.second.\_val.iptr = new int[gte.maxlen];                  // global variable need to init as 0                  for (int i = 0; i < gte.maxlen; i++) {                      entry.second.\_val.iptr[i] = 0;                  }                }              else if (gte.val.type == Type::FloatPtr) {                  entry.second.\_val.fptr = new float[gte.maxlen];                  // global variable need to init as 0                  for (int i = 0; i < gte.maxlen; i++) {                      entry.second.\_val.fptr[i] = 0;                  }              }              else {                  assert(0 && "wrong global value type with maxlen > 0");              }          }          global\_vars.insert(entry);  }  对于非零全局数组的初始化则由伪函数“\_global”完成，其函数体由Analyzer类的成员变量g\_init\_inst定义。在前端的处理中，会固定在main函数的开头插入一条“\_global”函数的调用，以保证全局变量被正确的初始化。      Function glb("\_global", Type::null);// 定义全局函数      glb.InstVec = g\_init\_inst;      glb.addInst(INS({},{},{},\_return));  program.addFunction(glb);// 添加全局函数  g\_init\_inst的组成指令在全局变量的声明过程中确定。当且仅当编译单元的首个子节点类型为Decl时，这个声明一定是全局变量的声明。其后续的所有初始值解析过程都是g\_init\_inst的一部分。  void Analyzer::analyzeCompUnit(CompUnit\* root, ir::Program &program)  {      if(FST\_NODE\_IS(DECL))          ANALYSIS(Decl, 0, g\_init\_inst)      else      {          GET\_CHILD\_PTR(func, FuncDef, 0)          // TODO2.18          symbol\_table.add\_scope();          analyzeFuncDef(func, program);          symbol\_table.exit\_scope();      }      if(SIZE > 1)      {          GET\_CHILD\_PTR(comp, CompUnit, 1)          analyzeCompUnit(comp, program);      }  }  在后端的实现中，则会从program的所有函数中单独筛选出“\_global”函数并进行内存空间的申请和初始化。初始化不为零的全局变量将被布置在data段，否则将被布置在bss段：  void backend::Generator::gen\_glb()  {      auto &glb\_var = program.globalVal;      auto &glb\_func = program.functions.back();// 全局函数\_global在实验二中是最后添加的        std::set<std::string> \_data;      // TODO3.1;      // 所有的全局变量和数组,其初始化值都可以在编译时求值到整数。因此\_global函数中实际上只有三种指令:      // 1.mov/fmov,且赋的值一定是字面量      // 2.store,且存的值也一定是字面量      // 3.return null      // 所以,当\_global的函数体不止含有return指令时,表明存在.data段      if(glb\_func.InstVec.size() > 1)      {          // .data段          fout << "\t.data" << ENDL;          fout << "\t.balign\t4" << ENDL;          for(auto &ins : glb\_func.InstVec)          {               if(ins->op == Operator::mov || ins->op == Operator::fmov)              {                  assert(get\_scope(ins->op1) == Scope::Literal);                  if((ins->op1.type == Type::IntLiteral && stoi(ins->op1.name) != 0) || (ins->op1.type == Type::FloatLiteral && stof(ins->op1.name) != 0))                  {                      \_data.insert(ins->des.name);                      fout << "\t.global\t" << ins->des.name << ENDL;                      fout << "\t.type\t" << ins->des.name << ", @object" << ENDL;                      fout << ins->des.name << ":" << ENDL;                      fout << (ins->op == Operator::mov ? "\t.word\t" : "\t.single\t") << ins->op1.name << ENDL;                  }              }              else if(ins->op == Operator::store)              {                  assert(get\_scope(ins->des) == Scope::Literal);                  if(\_data.count(ins->op1.name) == 0)                  {                      \_data.insert(ins->op1.name);                      fout << "\t.global\t" << ins->op1.name << ENDL;                      fout << "\t.type\t" << ins->op1.name << ", @object" << ENDL;                      fout << ins->op1.name << ":" << ENDL;                  }                  fout << (ins->op1.type == Type::IntPtr ? "\t.word\t" : "\t.single\t") << ins->des.name << ENDL;              }              else                  assert(ins->op == Operator::\_return);          }          fout << ENDL;      }      // 当data段的符号数量和全局变量数不一致时,表明存在.bss段(暂不考虑.rodata等其他数据段)      if(\_data.size() < glb\_var.size())      {          // .bss段          fout << "\t.bss" << ENDL;          fout << "\t.balign\t4" << ENDL;          for(auto &var : glb\_var)          {              if(\_data.count(var.val.name) == 0)              {                  fout << "\t.global\t" << var.val.name << ENDL;                  fout << "\t.type\t" << var.val.name << ", @object" << ENDL;                  fout << var.val.name << ":" << ENDL;                  fout << "\t.space\t" << std::max(var.maxlen, 1) \* 4 << ENDL;              }          }          fout << ENDL;      }  }  本次实验设计中的汇编层ABI为RISCV32。典型的生成汇编如下所示，.balign 4表明之后的（数据）段以4字节对齐，单个变量或数组的最小单位占4个字节。.global表明该符号是全局符号,.type表明该符号属于对象类型，区别与函数类型@function。data段的初始值使用.word直接声明，bss段则使用.space申请相应的内存空间(变量为4，数组为maxlen\*4)并进行零初始化：      .data      .balign 4      .global INF\_Scope0      .type   INF\_Scope0, @object  INF\_Scope0:      .word   65535      .bss      .balign 4      .global book\_Scope0      .type   book\_Scope0, @object  book\_Scope0:      .space  64      .global dis\_Scope0      .type   dis\_Scope0, @object  dis\_Scope0:      .space  64  因此，代表段的解析过程不再重复解析“\_global”函数，其也不是.text段的一部分：      gen\_glb();// 数据段      if(program.functions.size() > 0)// 代码段      {          fout << "\t.text" << ENDL;          fout << "\t.align\t2" << ENDL;          for(auto &func : program.functions)          {              if(func.name == "\_global")                  continue;              gen\_func(func);          }  }  同样的，调用“\_global”函数的IR命令也不作解析：          case Operator::call:              {                  auto call = dynamic\_cast<ir::CallInst\*>(&ins);                  if(call->op1.name != "\_global")// 全局函数只用于存储数据段信息                  {                      ...                  }  这样的设计使得数据段和代码段的区分变得非常简单，方便了IR程序到RISCV汇编程序的转换过程。   1. IR评测机中的函数调用   当IR评测机检测到call指令时，函数调用过程开始。评测机首先在函数库中查询目标函数，当且仅当目标函数存在时，初始化目标函数的上下文对象cxt，记录返回值地址，然后同时解析目标函数调用的实参列表和目标函数的形参列表，检查数据类型的一致性，并后实参填充到cxt的变量空间中，使其可以访问。最后将当前函数的上下文压栈，函数体指针指向目标函数cxt，目标函数的执行随后开始；当IR评测机检测到return指令时，表明当前函数执行完成。调用者弹栈并覆写cur\_ctx，根据返回值地址将返回值覆写到目标位置：              case Operator::call: {                  auto callinst = dynamic\_cast<CallInst\*>(inst);                  auto fn = callinst->op1.name;                  // lib functions                  Value libfunc\_retval;                  if (exec\_lib\_function(callinst, &libfunc\_retval)) {                      if (callinst->des.type != Type::null) {                          \*get\_des\_operand(inst->des) = libfunc\_retval;                      }                      cur\_ctx->pc++;                      break;                  }                  // ir::Function                  Context\* cxt = nullptr;                  for(auto& f: program->functions) {                      if (f.name == fn) {                          cxt = new Context(&f);                      }                  }                  // return type checking                  assert(cxt->pfunc->returnType == Type::null || inst->des.type == cxt->pfunc->returnType);                  if (cxt->pfunc->returnType != Type::null) {                      cxt->retval\_addr = get\_des\_operand(inst->des);                  }                  if (cxt) {                      // type checking                      for (size\_t i = 0; i < cxt->pfunc->ParameterList.size(); i++) {                          auto para = cxt->pfunc->ParameterList[i];                          assert(i < callinst->argumentList.size() && "callinst's arguement list should match function's parameter list");                          auto arg = callinst->argumentList[i];                          switch (arg.type) {                          case Type::Int:                          case Type::IntLiteral:                              assert(para.type == Type::Int);                              break;                          case Type::Float:                          case Type::FloatLiteral:                              assert(para.type == Type::Float);                              break;                          // pointers                          default:                              assert(arg.type == para.type);                              break;                          }                          // pass arguement into new context                          cxt->mem.insert({para.name, find\_src\_operand(arg)});                      }                      cur\_ctx->pc++;                      cxt\_stack.push(cur\_ctx);                      cur\_ctx = cxt;                  }                  else {                      assert(0 && "could not find the function in ir::Program");                  }              } break;   1. 参数数组   参数数组的支持依赖于指针运算指令getptr的实现。就后端而言，Int，IntPtr和FloatPtr三者并没有本质不同，都表示为32位的整型数据，区别只在于使用目的不同。如果被调函数需要访问调用者所指定的数组空间，只需要调用者在函数调用前计算出目标数组的首地址（此处的首地址区别于数组地址。当调用者只需要函数访问目标数组的子数组时，传入的首地址实际上是子数组在内存空间中的起始位置）并传递到参数寄存器中即可。被调函数本身包含所有必要的偏移量信息，两相结合即可唯一确定目标地址。在汇编程序的任意位置访问相同的内存地址并不存在任何差别。          case Operator::getptr:              if(get\_scope(op1) == Scope::Global)// 全局数组              {                  cxt.add\_ins(rvOPCODE::LA, getRd(des), op1.name);                  if(op2.type == Type::IntLiteral)                      cxt.add\_ins(rvOPCODE::ADDI, getRd(des), getRs(des), INT(op2.name) \* 4);                  else                  {                      cxt.add\_ins(rvOPCODE::SLLI, getRd(op2), getRs(op2), 2);                      cxt.add\_ins(rvOPCODE::ADD, getRd(des), getRs(des), getRs(op2));                  }              }              else// 局部数组              {                  if(ARG(op1))                  {                      if(ARG\_OVERFLOW(op1))                          cxt.add\_ins(rvOPCODE::LW, getRd(des), rv\_reg(rvREG::fp), ARG\_OFFSET(op1));                      else                          cxt.add\_ins(rvOPCODE::MV, getRd(des), getRs(op1), {});                      if(op2.type == Type::IntLiteral)                          cxt.add\_ins(rvOPCODE::ADDI, getRd(des), getRs(des), INT(op2.name) \* 4);                      else                      {                          cxt.add\_ins(rvOPCODE::SLLI, getRd(op2), getRs(op2), 2);                          cxt.add\_ins(rvOPCODE::ADD, getRd(des), getRs(des), getRs(op2));                      }                      if(ARG\_OVERFLOW(op1))                          cxt.add\_ins(rvOPCODE::SW, {}, rv\_reg(rvREG::fp), getRs(des), ARG\_OFFSET(des));                  }                  else                  {                      assert(cxt.stack.count(op1.name));                      cxt.add\_ins(rvOPCODE::ADDI, getRd(des), rv\_reg(rvREG::sp), cxt.stack[op1.name]);                      if(op2.type == Type::IntLiteral)                          cxt.add\_ins(rvOPCODE::ADDI, getRd(des), getRs(des), INT(op2.name) \* 4);                      else                      {                          cxt.add\_ins(rvOPCODE::SLLI, getRd(op2), getRs(op2), 2);                          cxt.add\_ins(rvOPCODE::ADD, getRd(des), getRs(des), getRs(op2));                      }                  }              }              break;   1. 短路运算   短路运算在IR层实现，包括逻辑与运算和逻辑或运算两种。每次逻辑运算结束后，将对结果进行检查，并根据检查结果确定是否提前跳出条件判断语块。当逻辑运算的结果可以唯一确定时，此后的逻辑语块将不再继续执行：  void Analyzer::analyzeLOrExp(LOrExp\* root, vector<ir::Instruction\*> &pgm)  {      GET\_CHILD\_PTR(land, LAndExp, 0)      VINS land\_bbl;      analyzeLAndExp(land, land\_bbl);        if(SIZE > 1)      {          GET\_CHILD\_PTR(lor, LOrExp, 2)          VINS lor\_bbl;          analyzeLOrExp(lor, lor\_bbl);          if(LITERAL(land) && LITERAL(lor))// 都是常量则结果为常量          {              if(land->t == Type::IntLiteral && lor->t == Type::IntLiteral)                  root->v = STR(stoi(land->v) || stoi(lor->v));              else if(land->t == Type::IntLiteral && lor->t == Type::FloatLiteral)                  root->v = STR(stoi(land->v) || stof(lor->v));              else if(land->t == Type::FloatLiteral && lor->t == Type::IntLiteral)                  root->v = STR(stof(land->v) || stoi(lor->v));              else                  root->v = STR(stof(land->v) || stof(lor->v));              root->t = Type::IntLiteral;          }          else          {              if(LITERAL(land))// 第一个值为常量,第二个值为变量              {                  if((land->t == Type::IntLiteral && stoi(land->v) != 0) || (land->t == Type::Float && stof(land->v) != 0))// 第一个值为真则结果为常量, 不需要计算第二个值                  {                      root->v = "1";                      root->t = Type::IntLiteral;                  }                  else// 否则等效于第二个值                  {                      ADDBBL(lor\_bbl)// 运行时计算第二个值                      COPY\_NODE(lor)// 复制则不释放临时变量                  }              }              else// 第一个值是变量,第二个值要变成变量              {                  TO\_OPERAND(land, op1)                  TO\_OPERAND(lor, op2)                  NEW\_OPERAND(des, Type::Int);                  ADDBBL(land\_bbl)// land...                  // TODO2.40.1                  // mov                  // if, goto out                  // lor...                  // \_or                  // out                  // TODO2.33;                  ADDINS(op1, {}, des, mov)                  ADDINS(des, {}, Operand(STR(int(lor\_bbl.size()) + 2), Type::IntLiteral), \_goto)//第一个值为真则不计算第二个值                  ADDBBL(lor\_bbl)                  ADDINS(des, op2, des, \_or)                  BE\_ROOT(des)              }          }        }      else      {          if(!LITERAL(land))              ADDBBL(land\_bbl)// 只有变量可能需要添加运算指令          COPY\_NODE(land)// 复制不释放变量      }  }  void Analyzer::analyzeLAndExp(LAndExp\* root, vector<ir::Instruction\*> &pgm)  {      GET\_CHILD\_PTR(eq, EqExp, 0)      VINS eq\_bbl;      analyzeEqExp(eq, eq\_bbl);        if(SIZE > 1)      {          GET\_CHILD\_PTR(land, LAndExp, 2)          VINS land\_bbl;          analyzeLAndExp(land, land\_bbl);          if(LITERAL(eq) && LITERAL(land))// 都是常量则结果为常量          {              if(eq->t == Type::IntLiteral && land->t == Type::IntLiteral)                  root->v = STR(stoi(eq->v) && stoi(land->v));              else if(eq->t == Type::IntLiteral && land->t == Type::FloatLiteral)                  root->v = STR(stoi(eq->v) && stof(land->v));              else if(eq->t == Type::FloatLiteral && land->t == Type::IntLiteral)                  root->v = STR(stof(eq->v) && stoi(land->v));              else                  root->v = STR(stof(eq->v) && stof(land->v));              root->t = Type::IntLiteral;          }          else          {              if(LITERAL(eq))// 第一个值为常量,第二个值为变量              {                  if((eq->t == Type::IntLiteral && stoi(eq->v) == 0) || (eq->t == Type::Float && stof(eq->v) == 0))// 第一个值为真则结果为常量, 不需要计算第二个值                  {                      root->v = "0";                      root->t = Type::IntLiteral;                  }                  else// 否则等效于第二个值                  {                      ADDBBL(land\_bbl)// 运行时计算第二个值                      COPY\_NODE(land)// 复制则不释放临时变量                  }              }              else// 两个值需要是变量              {                  TO\_OPERAND(eq, op1)                  TO\_OPERAND(land, op2)                  NEW\_OPERAND(des, Type::Int)                  ADDBBL(eq\_bbl)// eq...                  // TODO2.40.2;                  assert(op1.type == Type::Int && op2.type == Type::Int);                  // mov                  // not(等效于eq zero)                  // if, goto out                  // land...                  // \_and                  // out                  NEW\_OPERAND(tmp, Type::Int)                  ADDINS(op1, {}, des, mov)                  ADDINS(des, ZERO, tmp, eq)                  ADDINS(tmp, {}, Operand(STR(int(land\_bbl.size()) + 2), Type::IntLiteral), \_goto)//第一个值为真则不计算第二个值                  ADDBBL(land\_bbl)                  ADDINS(des, op2, des, \_and)                  BE\_ROOT(des)              }          }      }      else      {          if(!LITERAL(eq))              ADDBBL(eq\_bbl)// 只有变量可能需要添加运算指令          COPY\_NODE(eq)      }  }  概括而言，IR程序的形式如下所示：  逻辑或：                  // mov 将第一个逻辑值移动到结果变量中                  // if, goto out // 当逻辑值为真时直接跳出语块                  // lor... // 否则计算第二个逻辑值                  // \_or // 进行或运算，保存到结果变量中                  // out  逻辑与：                  // mov                  // not(等效于eq zero)// 逻辑值为假时直接跳出语块                  // if, goto out                  // land...                  // \_and                  // out   1. 寄存器分配   本次实验设计的寄存器分配策略为线性寄存器分配。函数体的首次遍历过程所分配的寄存器不考虑真实架构中的寄存器数量，而只确定IR变量所对应的寄存器类型。局部变量分配到s和fs寄存器中，临时变量分配搭配t和ft寄存器中，函数参数分配到a和fa寄存器中。对于本次实验设计而言，保存寄存器数量是完全充足的，溢出的参数寄存器提供了溢出栈进制，因此只需要对临时寄存器进行分配；在第二次遍历过程中，数据结构last记录了每个虚拟寄存器最后一次出现的位置，代表了其生命周期的终点；最后一次遍历执行最终的寄存器分配任务，为指令中的虚拟寄存器分配真实的物理寄存器。当虚拟寄存器首次出现时，从空闲的物理寄存器中分配一个供其使用；当虚拟寄存器生命周期结束时，则对其所使用的物理寄存器进行回收。工程实践证明，临时寄存器的数量同样完全充足，其使用率在大多数情况下甚至不足50%。因此，本设计暂不提供保持寄存器和临时寄存器的溢出支持：  void backend::context::reallocate()// 寄存器重分配进行的是第二、三次遍历  {      std::map<rv\_reg, int> last;      int len = pseudocode.size();      for(int i = 0; i < len ; i++)      {          auto &ins = pseudocode[i];          if(VALID(ins.rs1))              last[ins.rs1] = i;          if(VALID(ins.rs2))              last[ins.rs2] = i;          if(VALID(ins.rd))              last[ins.rd] = i;      }      std::map<rv\_reg, int> realloc;      std::vector<int> free\_reg = {6, 5, 4, 3, 2, 1, 0};      std::set<int> active\_reg;      std::set<int> saved\_temp\_reg;      std::vector<int> free\_freg = {11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};      std::set<int> active\_freg;      std::set<int> saved\_temp\_freg;      for(int i = 0; i < len ; i++)      {          int active\_size = (active\_reg.size() + active\_freg.size()) \* 4;          int saved\_temp\_size = (saved\_temp\_reg.size() + saved\_temp\_freg.size()) \* 4;          auto &rs1 = pseudocode[i].rs1;          auto &rs2 = pseudocode[i].rs2;          auto &rd = pseudocode[i].rd;          if(VALID(rs1))          {              assert(realloc.count(rs1));              rs1.index = realloc[rs1];          }          if(VALID(rs2))          {              assert(realloc.count(rs2));              rs2.index = realloc[rs2];          }            for(auto &loc : realloc)          if(last.count(loc.first) && last[loc.first] <= i)          {              if(loc.first.type == rvREG::t)              {                  active\_reg.erase(loc.second);                  free\_reg.push\_back(loc.second);              }              else              {                  active\_freg.erase(loc.second);                  free\_freg.push\_back(loc.second);              }              last.erase(loc.first);          }          if(VALID(rd))          {              if(realloc.count(rd))                  rd.index = realloc[rd];              else              {                  if(rd.type == rvREG::t)                  {                      assert(free\_reg.size());                      rd.index = realloc[rd] = free\_reg.back();                      active\_reg.insert(free\_reg.back());                      free\_reg.pop\_back();                  }                  else                  {                      assert(free\_freg.size());                      rd.index = realloc[rd] = free\_freg.back();                      active\_freg.insert(free\_freg.back());                      free\_freg.pop\_back();                  }              }          }      }  }   1. 汇编层的函数调用   汇编层的调用过程可以概括如下：  对于调用者   1. 保存调用者的参数寄存器 2. 修改参数寄存器，将被调者所需的参数移动到参数寄存器中 3. 保存调用者的活跃的临时寄存器 4. 将溢出的参数进行压栈 5. 函数调用 6. 溢出参数弹栈 7. 恢复活跃的临时寄存器 8. 保存a0或fa0中的返回值到临时寄存器 9. 恢复参数寄存器 10. 恢复栈帧寄存器   对于被调者：   1. 将sp寄存器值移动到fp寄存器 2. 移动sp，申请栈空间 3. 保存fp和ra寄存器 4. 保存保存寄存器 5. 函数过程 6. 恢复保存寄存器 7. 恢复ra寄存器 8. 将fp寄存器值移动到sp寄存器   其中需要特别说明的是栈帧寄存器的恢复。在本次实验设计中，栈帧寄存器保存的是函数入口处的栈指针值，随后栈指针发生移动，开辟函数的过程帧空间。因此，栈帧寄存器之下的栈空间属于被调函数，之上则属于调用者。溢出的参数保存在栈帧寄存器之上，第一个溢出参数可以表示为0(fp)，然后是4(fp)，以此类推。栈帧寄存器的保存在函数的入口处，和返回地址寄存器ra的位置相邻。完整的实现如下所示：          case Operator::call:              {                  auto call = dynamic\_cast<ir::CallInst\*>(&ins);                  if(call->op1.name != "\_global")// 全局函数只用于存储数据段信息                  {                      int a\_cnt = 0, fa\_cnt = 0;                      context mva, ovf;                      for(auto &opd : call->argumentList)                      if(opd.type == Type::IntPtr || opd.type == Type::FloatPtr || opd.type == Type::Int || opd.type == Type::IntLiteral)                      {                          if(a\_cnt < ARG\_REG\_SIZE)                          {                              auto rd = rv\_reg(rvREG::a, a\_cnt);                              switch(get\_scope(opd))                              {                                  case Scope::Global:                                      mva.add\_ins(rvOPCODE::LA, rd, opd.name);                                      mva.add\_ins(rvOPCODE::LW, rd, rd, 0);                                      break;                                  case Scope::Local:                                  case Scope::Temporary:                                      mva.add\_ins(rvOPCODE::MV, rd, getRs(opd), {});                                      break;                                  case Scope::Literal:                                      mva.add\_ins(rvOPCODE::LI, rd, {}, INT(opd.name));                                  default:                                      break;                              }                          }                          else                          {                              rv\_reg rs;                              switch(get\_scope(opd))                              {                                  case Scope::Global:                                      rs = rv\_reg(rvREG::t, cxt.tmp++);                                      ovf.add\_ins(rvOPCODE::LA, rs, opd.name);                                      ovf.add\_ins(rvOPCODE::LW, rs, rs, 0);                                      break;                                  case Scope::Local:                                  case Scope::Temporary:                                      rs = getRs(opd);                                      break;                                  case Scope::Literal:                                      rs = rv\_reg(rvREG::t, cxt.tmp++);                                      ovf.add\_ins(rvOPCODE::LI, rs, {}, INT(opd.name));                                  default:                                      break;                              }                              ovf.add\_ins(rvOPCODE::SW, {}, rv\_reg(rvREG::sp), rs, (a\_cnt - ARG\_REG\_SIZE) \* 4);                          }                          a\_cnt++;                      }                      int arg\_ovf\_cnt = std::max(a\_cnt - ARG\_REG\_SIZE, 0);                      for(auto &opd : call->argumentList)                      if(opd.type == Type::Float || opd.type == Type::FloatLiteral)                      {                          if(fa\_cnt < FARG\_REG\_SIZE)                          {                              auto rd = rv\_reg(rvREG::fa, fa\_cnt);                              switch(get\_scope(opd))                              {                                  case Scope::Global:                                      mva.add\_ins(rvOPCODE::LA, rv\_reg(rvREG::t, cxt.tmp), opd.name);                                      mva.add\_ins(rvOPCODE::FLW, rd, rv\_reg(rvREG::t, cxt.tmp++), 0);                                      break;                                  case Scope::Local:                                  case Scope::Temporary:                                      mva.add\_ins(rvOPCODE::FMV, rd, fgetRs(opd), {});                                      break;                                  case Scope::Literal:                                      mva.add\_ins(rvOPCODE::LI, rv\_reg(rvREG::t, cxt.tmp), {}, f2li(std::stof(opd.name)));                                      mva.add\_ins(rvOPCODE::FMVWX, rd, rv\_reg(rvREG::t, cxt.tmp++), {});                                  default:                                      break;                              }                          }                          else                          {                              rv\_reg frs;                              switch(get\_scope(opd))                              {                                  case Scope::Global:                                      ovf.add\_ins(rvOPCODE::LA, rv\_reg(rvREG::t, cxt.tmp), opd.name);                                      ovf.add\_ins(rvOPCODE::FLW, frs, rv\_reg(rvREG::t, cxt.tmp++), 0);                                      break;                                  case Scope::Local:                                  case Scope::Temporary:                                      frs = fgetRs(opd);                                      break;                                  case Scope::Literal:                                      cxt.add\_ins(rvOPCODE::LI, rv\_reg(rvREG::t, cxt.tmp), {}, f2li(std::stof(op1.name)));                                      cxt.add\_ins(rvOPCODE::FMVWX, frs, rv\_reg(rvREG::t, cxt.tmp++), {});                                  default:                                      break;                              }                              ovf.add\_ins(rvOPCODE::FSW, {}, rv\_reg(rvREG::sp), frs, (fa\_cnt - FARG\_REG\_SIZE + arg\_ovf\_cnt) \* 4);                          }                          fa\_cnt++;                      }                      int ret\_save = (func.returnType == Type::IntPtr || func.returnType == Type::FloatPtr || func.returnType == Type::Int || func.returnType == Type::IntLiteral);                      int fret\_save = (func.returnType == Type::Float || func.returnType == Type::FloatLiteral);                      int arg\_save = std::min(std::min(cxt.argc, std::max(a\_cnt, ret\_save)), ARG\_REG\_SIZE);                      int farg\_save = std::min(std::min(cxt.fargc, std::max(fa\_cnt, fret\_save)), FARG\_REG\_SIZE);                        int arg\_size = (arg\_save + farg\_save) \* 4;                      // TODO3.11                      if(arg\_size)// 先保存参数寄存器                      {                          cxt.add\_ins(rvOPCODE::ADDI, rv\_reg(rvREG::sp), rv\_reg(rvREG::sp), -arg\_size);                          for(int i = 0 ; i < arg\_save ; i++)                              cxt.add\_ins(rvOPCODE::SW, {}, rv\_reg(rvREG::sp), rv\_reg(rvREG::a, i), i \* 4);                          for(int i = 0 ; i < farg\_save ; i++)                              cxt.add\_ins(rvOPCODE::FSW, {}, rv\_reg(rvREG::sp), rv\_reg(rvREG::fa, i), i \* 4);                      }                      // 然后修改参数寄存器s                      for(auto &ins : mva.pseudocode)                          cxt.pseudocode.push\_back(ins);                      // 然后保存临时寄存器                      cxt.add\_ins(rvOPCODE::STR, {});                      // 最后溢出参数压栈                      int ovf\_size = (arg\_ovf\_cnt + std::max(fa\_cnt - FARG\_REG\_SIZE, 0)) \* 4;                      if(ovf\_size)                      {                          cxt.add\_ins(rvOPCODE::ADDI, rv\_reg(rvREG::sp), rv\_reg(rvREG::sp), -ovf\_size);                          for(auto &ins : ovf.pseudocode)                              cxt.pseudocode.push\_back(ins);                      }                      // 函数调用                      cxt.add\_ins(rvOPCODE::CALL, op1.name);                      // 溢出参数弹栈                      if(ovf\_size)                          cxt.add\_ins(rvOPCODE::ADDI, rv\_reg(rvREG::sp), rv\_reg(rvREG::sp), ovf\_size);                        // 恢复临时寄存器                      cxt.add\_ins(rvOPCODE::LTR, {});                      // 保存返回值(可能会覆盖临时寄存器)                      switch(des.type)                      {                          case Type::Int:                              cxt.add\_ins(rvOPCODE::MV, getRd(des), rv\_reg(rvREG::a, 0), {});                              break;                          case Type::Float:                              cxt.add\_ins(rvOPCODE::FMV, fgetRd(des), rv\_reg(rvREG::fa, 0), {});                              break;                          case Type::null:                              break;                          default:                              assert(0 && "invalid call");                              break;                      }                      // 恢复参数寄存器                      if(arg\_size)                      {                          for(int i = farg\_save - 1 ; i >= 0 ; i--)                              cxt.add\_ins(rvOPCODE::FLW, rv\_reg(rvREG::fa, i), rv\_reg(rvREG::sp), i \* 4);                          for(int i = arg\_save - 1 ; i >= 0 ; i--)                              cxt.add\_ins(rvOPCODE::LW, rv\_reg(rvREG::a, i), rv\_reg(rvREG::sp), i \* 4);                          cxt.add\_ins(rvOPCODE::ADDI, rv\_reg(rvREG::sp), rv\_reg(rvREG::sp), arg\_size);                      }                      // 恢复栈帧寄存器                      cxt.add\_ins(rv::rvOPCODE::LW, rv\_reg(rvREG::fp), rv\_reg(rvREG::sp), cxt.arr\_size);                  }              }              break;  汇编层的返回指令和IR层没有太多区别，返回值固定由被调函数保存在a0寄存器或fa0寄存器中。由于每个函数可能有多个返回值地址，因此保存寄存器的恢复也需要复制多次。一个典型的函数调用如下所示：      .text      .align  2      .global func      .type   func, @function  func:      mv  fp, sp      addi    sp, sp, -8      sw  fp, 0(sp)      sw  ra, 4(sp)      mv  t0, a0      li  t1, 1      sub t0, t0, t1      mv  a0, t0      lw  ra, 4(sp)      mv  sp, fp      ret      .global main      .type   main, @function  main:      mv  fp, sp      addi    sp, sp, -12      sw  fp, 0(sp)      sw  ra, 4(sp)      sw  s1, 8(sp)      li  s1, 114514      la  t0, a\_Scope0      li  t1, 10      sw  t1, 0(t0)      la  a0, a\_Scope0      lw  a0, 0(a0)      call    func      mv  t1, a0      lw  fp, 0(sp)      mv  s1, t1      mv  a0, s1      lw  s1, 8(sp)      lw  ra, 4(sp)      mv  sp, fp      ret      li  a0, 0      lw  s1, 8(sp)      lw  ra, 4(sp)      mv  sp, fp      ret   1. 优化   本次实验设计的优化相对简单。常量传播的过程在IR层实现。对于所有的运算，当且仅当输入变量均不为立即数时，结果的类型为变量，否则按常数处理。常量变量在汇编层中保存在.data段，但不在.text段进行任何访问。汇编层直接加载编译时计算所得的立即数到寄存器；寄存器分配采用的是相对简单的线性寄存器分配，能够显著地降低内存访问次数。对于实验提供的测试点而言，当且仅当访问数组或参数溢出时需要访问内存，其他的所有运算都直接运行在寄存器上；对于死代码消除，本次实验设计基于经验主义删除了部分无效命令，例如恒真恒假条件相关的一系列逻辑运算和跳转指令。 | | | | | |
| 四、实验测试  1、修改CMakeLists.txt文件：  set(CMAKE\_C\_COMPILER "/usr/bin/x86\_64-linux-gnu-gcc-7")  set(CMAKE\_CXX\_COMPILER "/usr/bin/x86\_64-linux-gnu-g++-7")  添加上述语句，指定编译器的路径和版本。  2、在/coursegrader目录下创建build目录，cd并以下命令：  cmake ..  make  以完成项目的编译，生成可执行文件compiler。项目的每次修改都需要重新编译。   1. 在/test目录下运行run.py，使用自己实现的编译器逐个编译每个测试文件并生成相应的.tk,.json等目标文件。 2. 在/test目录下运行score.py,检查生成的目标文件的正确性并生成相应的评分。 3. 从汇编到RISC-V   riscv32-linux-gnu-gcc会将.s文件和.a文件链接并转换为RISC-V格式的二进制文件，由qemu模拟其硬件运行过程并返回结果。 | | | | | |
| 1. 实验总结   **1、问题和解决方法**  （1）可以运行python3 test.py [选项,例如-s1, S],它会重新编译项目,而不必在/test和/build间来回切换。  如果发现没有build，要使用mkdir build创建这个文件夹,而不是像指导书说的cd进去。框架本身没有提供/build。  如果实验二的时候在build下遇到了报错说版本不匹配,可以把CMakeLists.txt中的cmake\_minimum\_required(VERSION 3.20)更改为3.10,好像只有实验二有这个问题,可能是写的时候敲歪了。  如果报错说缺少某些文件,可以尝试检测CMakeLists.txt中的  # --------------------- 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})  # --------------------- from src ---------------------  保证没有注释掉这些资源文件路径,然后重新make。  （2）只有重新编译后再运行run.py或score.py才能应用更改。如果运行test的时候发现S选项会报错,参考test.py进行修改  它本身的结构其实比较简单,先后调用了编译,run.py和score.py,并没有做什么别的工作  （3）如果想只运行单个测试点,可以修改run.py的以下位置:  for src in src\_files:      fname, ftype = src.split('.')      if("XX" not in fname):          continue      cmd = ' '.join([compiler\_path, testcase\_dir + src, step, "-o", output\_dir + fname + "." + oftype])      if is\_windows:          cmd = cmd.replace('/','\\')      cp = subprocess.run(cmd, shell=True, stderr=subprocess.PIPE, stdout=subprocess.DEVNULL)      if cp.returncode != 0:          record[src] = {"retval": cp.returncode, "err\_detail": cp.stderr}      else:          record[src] = {"retval": 0}      print(src, record[src])  XX是测试点的序号,例如95。这句话的意思是如果测试点的名称里面没有95就什么都不做  （4）如果评测的时候卡死了，但控制台又没输出是哪个测试点,可以修改score.py的以下位置:      if step == "-s0":          for i in ["basic", "function"]:              output\_dir = output\_base + i + '/'              ref\_dir = ref\_base + i + '/'              if os.path.exists(output\_dir):                  files = os.listdir(output\_dir)                  for file in files:                      if not (file[-3:] == ".tk"):                          continue                      cmd = ' '.join(["diff", ref\_dir + file, output\_dir + file, '-w'])                      if is\_windows:                          cmd = cmd.replace('/','\\')                      # print(cmd)                      print(file, "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")                      cp = subprocess.run(cmd, shell=True, stderr=subprocess.DEVNULL, stdout=subprocess.PIPE)                      if cp.returncode != 0:                          record[file] = {"retval": cp.returncode, "err\_detail": "diff test failed"}                      else:                          score += 1                          record[file] = {"retval": 0}                      print(file, record[file])  这样它就会在测试每个样例前先打印名称。注意此处修改的是-s0的部分,如果在其他实验卡死了就修改对应的位置  （5）如果发现写的cout,哪怕是cerr也并没有在控制台输出任何内容,可以在感觉有问题的地方加上:  FILE \*trace = fopen("trace", "w");  fprintf(trace, "%s\n", toString(ins->op).c\_str());  fclose(trace);  // assert(0 && "xxx");  意思是新建一个叫trace的文件,然后把想打印的东西打印到这个文件里面去。"w"表示写模式,它会直接覆写原有的trace,没有就新建。  如果想在文件最后打印而不是完全覆写,可以使用"a"。如果程序正常结束了,那么不写fclose也会自动保存trace,无伤大雅。  但如果中途因为assert之类的问题挂掉了,那么不写fclose,想打印的内容就不会被实际写入文件,可能是留在缓冲区之类的地方,  然后就随着程序结束清空掉了。所以如果在trace里也没找到想要的输出,可以在报错的位置之前加上fclose。  assert报错的时候控制台会告诉是哪个文件的哪一行触发了这个断言,或者可以使用:  python3 xx.py [选项] | grep 'xxx'  其中xxx是写在assert里面的内容的一部分,例如对于assert(0 && "invalid"),可以填'0'或者'inv'之类的  |是管道符,用于将符号前的指令在控制台的输出用作符号后的指令的输入。grep则是linux本身提供的文本搜索指令。这条指令会筛选并高亮所搜索的输出内容。如果不幸错麻了,筛选完还是看不清,可以使用:  python3 xx.py [选项] | grep 'xxx' > GG.txt  输出方向符>会将输出重定位到GG.txt这个文件中,而不是打印在终端。>是覆盖,>>是追加。  **2、实验建议**  项目的编译和运行时间不太应该包含的运行计时中。实验提供的测试点规模很小，实现汇编优化所导致的项目的编译和运行时间的代价可能远大于优化在测试集上的实际效果。 | | | | | |