编译原理研讨课实验PR003实验报告

任务说明

本次实验主要分为如下三部分：

1. 中间代码的设计与生成
2. 汇编代码的生成
3. 代码优化

成员组成

代瀚堃：中间代码生成的大部分 (80% of intermediateRepresentation.cpp/h)、汇编代码生成 (all of codeGenerate.cpp/h)、流图的开头 (all of dataFlowAnalysis.cpp/h)

孙彬：翻译 unaryExp 和部分控制流语句成中间代码

郑天羽：书写报告中关于 cond 规则节点翻译成 IR 的部分

实验设计

设计思路

1. 整体流程

   对源文件进行词法、语法、语义分析后，得到保存有属性信息的 AST，根据 AST 节点上的信息生成中间代码；然后在数据流分析阶段划分基本块，进行机器无关优化；最后生成汇编代码，输出到文件

2. 中间代码的设计与生成

   我们将一条中间代码定义为如下结构体（这里省略了构造方法的定义）：

```
typedef struct intmed_code {
    intmed_op_t op;
    std::string result;
    val_type_t val_type;

    std::string arg1;
    std::string arg2;
} intmed_code_t;
```

其中 `op` 表示指令的操作码，`result` 表示指令的结果存放在哪一个临时变量，`arg1/2` 为操作数，`val_type` 为操作数/结果的类型

3. 汇编代码生成

   我们使用了龙书上 8.6 关于代码生成的算法，定义了寄存器描述符和地址描述符，按如下三种类型来将三地址代码翻译成汇编代码：

   `LD  reg, mem`

   `ST  mem, reg`

   `OP  reg, reg, reg`

   在寄存器分配时，由于用例中调用栈不是很深，我们倾向于优先分配 caller-saved 寄存器，分配时会顺序扫描它们，返回一个可用的寄存器

4. 机器无关优化

   主要是龙书上 8.5 的基本块优化和 8.7 的窥孔优化

实验实现

1. 中间代码的设计

同时，我们参考了 LLVM 和龙书上的中间表示，定义了如下几种中间代码：

操作码	result	val_type	arg1	arg2
LD	%temp	i/f/d/b	base	index/null
ST	%temp	i/f/d/b	base	index/null
CALL	%temp	i/f/d/b	func_name	argc
RETURN	null	i/f/d/b	%temp	null
FUNC_BEGIN	name	i/f/d/b	argc	null
FUNC_END	name	null	null	null
PARAM	null	i/f/d/b	%arg	null
D_ALLOC	%temp	i/f/d/b	num_items	init_values
G_ALLOC	#var	i/f/d/b	num_items	init_values
BEQ	target	i/f/d/b	%1	%2

操作码	result	val_type	arg1	arg2
BNE	target	i/f/d/b	%1	%2
BLT	target	i/f/d/b	%1	%2
BGT	target	i/f/d/b	%1	%2
J	target	void	null	null
ADD	%dest	i/f/d	%s1	%s2
ADD_V	%dest	i/f/d	%s1	%s2

说明：

`LD`：`LD, %1, int, base, index/null`，将从某地址上取 `int` 长度的数存放到 1 号临时变量中，不是数组的变量 `arg1=&var, arg2=null`，而数组元素则有 `arg1=base, arg2=index`；

`ST`：`ST, %1, int, base, index/null`，存数，与 `LD` 类似；

`CALL`：`CALL, %result, int, foo, 2`，调用名为 `foo` 的函数，参数个数为 `2`，返回类型为 `int`，返回的结果用临时变量 `%result` 表示；

`RETURN`：`RETURN, null, int, %result, null`，把临时变量 `%result` 中的值取一个 `int` 的长度，移到存放返回值的地方，将作为返回值返回；

`PARAM`：`PARAM, null, int, %arg, null`，把临时变量 `%arg` 中的值作为参数，传给之后将要调用的函数，参数类型为 `int`；

`FUNC_BEGIN`：开始一个函数的定义

`FUNC_END`：结束一个函数的定义

`D_ALLOC`：`D_ALLOC, %result, int, num_items, init_values` 函数执行时动态分配大小为 `num_items * sizeof(int)` 的空间，用字符串 `init_values` 中的值初始化（不足的项不初始化），这块空间的首地址赋给临时变量 `%result`，并在函数返回时释放；

`G_ALLOC`：`G_ALLOC, #var, int, num_items, init_values`，和上面类似，用于全局/静态分配，变量前带前缀 `#`；

条件/无条件跳转：语义和 RISC-V 指令类似

特别地，我们没有设计一个临时变量 `%1` 的值赋给另一个临时变量 `%2` 的 `MV` 指令，如果需要使用该值，直接用变量 `%1` 即可

另外，考虑到汇编代码生成的需要，如果在目标机器上使用栈来管理函数调用，那么中间代码 `PARAM` 需要按照目标机的 ABI 传参，`FUNC_BEGIN` 需要和 `FUNC_END` 协同完成栈的管理

2. 中间代码生成

(1) 变量/常量定义

根据变量/常量的位置，翻译成 `G_ALLOC` 或 `D_ALLOC` 即可，其翻译方案如下：

```
    if (current_scope == symbol_table.scope_root) {
        // global
        op = G_ALLOC;
    }else{
```

```
        // local
        op = D_ALLOC;
    }
    std::string id = ctx -> Ident() -> getText();
    var_symbol_item_t * var_item = &
(symbol_table.var_symbol_table[(name_in_scope){id, current_scope}]);
    std::string temp = newTemp(var_item -> var_type, var_item -> num_items,
var_item -> is_array, false, var_item);
    appendIntmed(
        op,
        temp,
        var_item -> var_type,
        intmed_arg_t(var_item -> var_name, var_item, false, var_item ->
num_items),
        value_list
    );
```

其中，如果有初始化，需要将初始值以逗号分隔的字符串形式存放在 value_list 中

(2) 函数定义

在函数头处生成 `FUNC_BEGIN`，并申请一个 label，作为函数统一的返回点，其实现如下：

```
    auto func_item = &(symbol_table.func_symbol_table[ctx -> Ident() ->
getText()]);
    // Notice: which code is next code?
    //func_item -> which_label = newLabel();
    current_func_end = newLabel();
    appendIntmed(FUNC_BEGIN, func_item -> func_name, func_item -> ret_type,
std::to_string((func_item -> fparam_list).size()));
```

对于函数中的 return 语句，将其翻译为 `RETURN` 的同时，还生成一条跳转到该 label 的指令；最终函数内所有的返回分支都会跳转到此处，在函数定义结束后生成一条 `FUNC_END` 指令，实现如下：

```
    auto & last_code = getACode(getIntmedNum() - 1);

    // Attach and End
    size_t num_code = getIntmedNum();
    attachLabel(current_func_end, num_code);
    appendIntmed(FUNC_END, ctx -> funcHeader() -> Ident() -> getText(),
void_type);
```

(3) 控制流语句的翻译

我们采用了龙书图 6.37 的翻译方案，在 eqExp 和 relExp 节点上生成比较的三地址代码，分别为：

```
        intmed_op_t op1;
        intmed_op_t op2 = J;
        std::string target1 = ctx -> truelist, target2 = ctx -> falselist;
        val_type_t rexp_type = (val_type_t)(ctx -> relExp() ->
rel_exp_val_type);
        std::string eexp_result = ctx -> eqExp() -> result_name;
        std::string rexp_result = ctx -> relExp() -> result_name;
```

```
        std::string relop = ctx -> eqOp() -> getText();

        switch(relop[0]){
            case '=': op1 = BEQ;    break;
            case '!': op1 = BNE;    break;
        }

        appendIntmed(
            op1,
            target1,
            rexp_type,
            intmed_arg_t(eexp_result, NULL, ctx -> eqExp() -> is_const),
            intmed_arg_t(rexp_result, NULL, ctx -> relExp() -> is_const)
        );

        appendIntmed(op2, target2, void_type);
```

和:

```
    intmed_op_t op1;
    intmed_op_t op2 = J;
    std::string target1 = ctx -> truelist, target2 = ctx -> falselist;
    val_type_t aexp_type = (val_type_t)(ctx -> addExp() ->
add_exp_val_type);
    std::string rexp_result = ctx -> relExp() -> result_name;
    std::string aexp_result = ctx -> addExp() -> result_name;

    std::string relop = ctx -> relOp() -> getText();

    switch(relop[0]){
        case '<': op1 = relop[1] == '=' ? BLE : BLT;    break;
        case '>': op1 = relop[1] == '=' ? BGE : BGT;    break;
    }

    appendIntmed(
        op1,
        target1,
        aexp_type,
        intmed_arg_t(rexp_result, NULL, ctx -> relExp() -> is_const),
        intmed_arg_t(aexp_result, NULL, ctx -> addExp() -> is_const)
    );

    appendIntmed(op2, target2, void_type);
```

我们将 stmt 规则节点的 if-else 分支改动如下:

`'if' '(' cond ')' lab stmt (go lab 'else' stmt)?`

其中 `lab` 和 `go` 可替换为 ε 串，分别带有继承属性 `use_label` 和 `goto_label`，在进入该 if 语句时计算出来，用于打上标签和生成 `goto` 代码

while 语句的翻译类似，在进入该节点时申请 while_begin 和 while_end 两个 label，while 结束时跳转到 while_begin，遇到 continue 时跳转到 while_begin，遇到 break 时跳转到 while_end

(4) 访存指令

取内存中变量的值只有一种情况，即 lVal 出现在等号右端，对应 primaryExp 的 lVal 分支，我们在此处生成 `LD` 指令即可：

```
    bool is_array = ctx -> lVal() -> is_array;
    bool is_const = ctx -> is_const;
    size_t num_items = ctx -> lVal() -> num_items;
    std::string var_name = ctx -> lVal() -> Ident() -> getText();
    auto symtab_item = &((symbol_table.findVarInTab(var_name,
current_scope)) -> second);
    val_type_t dtype = (val_type_t)(ctx -> lVal() -> lval_type);
    std::string temp = newTemp(dtype, num_items, is_array, false,
symtab_item);
    std::string offset_name = "";
    auto exp = ctx -> lVal() -> exp();
    if (exp) {
        offset_name = exp -> result_name;
    }
    appendIntmed(LD, intmed_arg_t(temp, NULL, false), dtype,
intmed_arg_t(symtab_item -> var_name, NULL, false), offset_name);

    ctx -> result_name = temp;
```

此后通过临时变量来访问这个值

为内存中变量赋值也只存在一种情况，即 lVal 出现在等号左端时，对应 stmt 规则中赋值的分支，在此处生成 `ST` 指令：

```
    std::string exp_name = ctx -> exp() -> result_name;

    std::string lval_name = ctx -> lVal() -> Ident() -> getText();
    var_symbol_item_t * symtab_item = &(symbol_table.findVarInTab(lval_name,
current_scope) -> second);
    val_type_t lval_type = symtab_item -> var_type;
    appendIntmed(
        ST,
        exp_name,
        lval_type,
        intmed_arg_t(symtab_item -> var_name, symtab_item, symtab_item ->
is_const, symtab_item -> num_items)
    );
```

于是，内存中的变量只能通过临时变量来赋值

(5) 运算类指令

根据 (4)，访存均通过 `LD` 和 `ST` 完成，于是运算的三/两个操作数只可能是临时变量或常数，直接翻译即可，而数组逐元素运算翻译为相应带 `V` 后缀的向量指令

3. 汇编代码生成

(1) 地址描述符的设计：

```
class AddrDscr {
private:
    std::string var_name;  // Name of this var
    std::vector <var_pos_t> pos_list;  // Array of possible position
    var_pos_t origin_pos;  // Original space
    val_type_t dtype;  // data type
public:
    some methods ...
};
```

其中，`var_name` 为该变量或常量的名称，`pos_list` 为该变量可能的位置，包括寄存器、全局数据区、栈，`origin_pos` 为该变量初始时存在的位置，`dtype` 为该变量的类型 (int, bool, float, double)，并封装了操作该描述符的若干方法

(2) 寄存器描述符的设计：

```
class RegDscr {
private:
    reg_name_t reg_name;  // Name of this register
    reg_type_t reg_type;  // Type of this register
    std::vector <std::string> var_list;  // Variables residing this register
};
```

其中，`reg_name` 为该寄存器的名字，`reg_type` 表示该寄存器为浮点或定点，`var_list` 表示驻留在该寄存器中的变量

(3) 翻译 `LD` 指令

`IR` 中的 `LD` 指令实际上并不一定发生访存，我们认为它只是将内存变量的值传到了一个临时变量中，而在这一临时变量实际被使用时 (e.g. 用于运算)，我们再通过它的地址描述符判断它是否在寄存器中，从而生成 `load` 的汇编指令。这样一种延迟加载的方案一定程度上可以减少不必要的访存 (否则，可能会加载了之后未使用，又被写回，实际使用时又再加载一次，造成性能损失)

于是，在我们的设计中，只需要将内存变量的地址描述符拷贝一份给临时变量即可

(4) 翻译 `ST` 指令

`ST` 指令将临时变量的值赋给内存变量，确实可能发生访存，在我们的方案中，它类似于龙书上描述的 `COPY` 指令，我们先检查临时变量在寄存器中是否有值，有则从该寄存器 `store`，否则先生成 `load`，再 `store`

需要注意的是，RISC-V 中，浮点立即数不能被编码进指令，如果需要使用浮点常数，需要把它放到全局数据区来 load-store，为简单起见，作为基址指针的寄存器固定分配为 `s11`

如果生成了 `riscv-load/store` 指令，需要按照龙书 8.6 的方法更新描述符

(5) 翻译运算指令

根据两个源操作数是否是立即数，以及操作码来划分，`ADD` 和 `SUB` 有对应的立即数指令，而乘除类没有，立即数需要先通过 `li` 指令加载到寄存器中，再运算

描述符的获取和更新方法与龙书 8.6.2 一致，对应代码中的 getReg, getRegForSrc, getRegForDest 等函数

(6) 控制流的翻译

由于我们关于控制流的 `IR` 设计比较底层，直接翻译即可，需要注意的是，用于比较的两个临时变量可能不存在于寄存器中，必要时需要加载，这通过 getVarOrConstInReg 方法完成

(7) 函数的实现

函数翻译的难点在于栈地址不容易确定。我们用队列 `bq_queue` 记录从栈上分配的地址信息 (e.g. 相对栈底的偏移，由于栈从高地址往低地址增长，变量也按照该顺序分配，因此总能确定出各自相对栈底的偏移)，在遇到 `FUNC_BEGIN` 时初始化为空，遇到 `FUNC_END` 时回填此前使用过的栈地址，这些地址之前会用 `?` 来标记

在调用函数时，根据 ABI 将实参传入寄存器 (寄存器中的原来变量可能需要保存)，在函数返回时将返回值传入 `a0` 或 `fa0` 寄存器，然后跳转到统一的一段函数结束代码，恢复 `ra` 和 `sp` 后跳转回原地址

另外，在调用和返回时还需保存 caller-saved 和恢复 callee-saved 寄存器，不过在我们的实现中只使用了 caller-saved 寄存器，所以只需完成前者，我们实现在 `clearDscrForCall` 方法中：

```
    for (size_t i = 0; i < caller_saved_fix.size(); i ++) {
        auto rd = getRegDscr(caller_saved_fix[i]);
        // No x to cover, move out directly
        moveVarOut(rd, "", false);
        removeRegFromAll(caller_saved_fix[i]);
    }
    for (size_t i = 0; i < caller_saved_float.size(); i ++) {
        auto rd = getRegDscr(caller_saved_float[i]);
        moveVarOut(rd, "", false);
        removeRegFromAll(caller_saved_float[i]);
    }
```

4. 代码优化

   由于组长一个人能力有限，很遗憾我们并没有完成严格意义上的代码优化，我们的思路遵循龙书 8.5 和 8.7 节，我们计划在 dataFlowAnalysis.cpp/h 中完成基本块的划分及流图的生成，并在此基础上做基本块优化和窥孔优化

   不过，由于采用了龙书上的代码生成算法，我们生成的代码相比最为朴素的方案仍有少量的优化。

   比如，此前在 `LD` 指令翻译所提到的延迟加载，可以减少冗余的 load-store

   在选取寄存器时，我们通过检查寄存器描述符和地址描述符，复用已有的值，减少了数据传输的指令

   另外，我们实现了 getReg 函数，如果 `x=y op z` 中源和目的操作数相同，可以复用寄存器，实现紧凑分配

总结

实验结果总结

在人手较为有限的情况下，我们已经尽力，通过了功能测试中的 0 ~ 13 号。当然，另一个原因是我们选择了龙书上比较复杂的翻译方案，导致实现起来遇到了很多困难，如果在此方面舍弃一些，选择较为简单保守的翻译策略 (e.g. 计算完后马上 store、固定分配寄存器)，可能也是一种不错的选择

剩下的功能测试未通过的原因及可能的解决方案如下：

14. 我们以函数为单位运行代码生成的算法，而实际应该以基本块为单位，这导致第一次进入 while 循环时 b 的值虽然能保存在 `t0` 寄存器中，但在后续进入时描述符信息不对 (当前描述符应该在块内有效，而非函数)，没有重新取 `b` 的值。

    解决方案：完善 dataFlowAnalysis 中的基本块划分，以基本块为单位运行算法

15/16. 同上

17/19. `ST` 和 `LD` 的翻译暂未支持带 offset 的形式

一种可能的翻译方案是先将数组基址通过 `la` 指令加载到一个固定用于存放基址的寄存器中，往上加一个偏移量 `i` 后，再通过 `lw/sw` 等指令来访存

18/20. 暂未支持数组逐元素运算

一种可能的翻译方案是，不再使用 `ADD_V` 这样的 `IR`，遇到数组逐元素运算时产生多条单元素的运算及加载指令来代替

分成员总结

代瀚堃：另外两位组员对待实验态度一直较为消极，从实验 1 ~ 3 仅贡献了 200 行左右的代码，且其调试也基本由组长完成，多次提醒后仍无改变。虽然我们在实验三刚开始时的构想比较好，想实现龙书上的大部分算法，但最终由于组长精力和能力有限，仅完成了部分，确实有些遗憾。

(两位组员后期也许会提交自己写的代码，但组长在截止日提交该报告时并没有看到)