stage-1 实验报告

计 93 王哲凡 2019011200

实验内容

step2：一元操作

在 TAC 码生成时，对于一元运算符，添加了操作符的对应，具体来说，将 `BitNot` 对应到 `NOT`，将 `LogicNot` 对应到 `SEQZ`，在 `frontend/tacgen/tacgen.py` 中的 `visitUnary()` 函数中：

```
op = {
    node.UnaryOp.Neg: tacop.UnaryOp.NEG,
    node.UnaryOp.BitNot: tacop.UnaryOp.NOT,
    node.UnaryOp.LogicNot: tacop.UnaryOp.SEQZ,
    # You can add unary operations here.
}[expr.op]
```

除此之外，为了区分一元运算符的 TAC 码输出，对于 `utils/tac/tacinstr.py` 中的 `Unary` 类的 `__str__()` 函数，修改为：

```
def __str__(self) -> str:
    return "%s = %s %s" % (
        self.dst,
        ("-" if (self.op == UnaryOp.NEG) else "!" if (self.op == UnaryOp.NOT) else "SNEZ" if (self.OP == UnaryOp.SNEZ) else "SEQZ"),
        self.operand,
    )

```

step3：加减乘除模

类似一元操作，在 `frontend/tacgen/tacgen.py` 中的 `visitBinary()` 函数中添加对应的二元运算符 TAC 码对应规则：

```
op = {
    node.BinaryOp.Add: tacop.BinaryOp.ADD,
    node.BinaryOp.Sub: tacop.BinaryOp.SUB,
    node.BinaryOp.Mul: tacop.BinaryOp.MUL,
    node.BinaryOp.Div: tacop.BinaryOp.DIV,
    node.BinaryOp.Mod: tacop.BinaryOp.REM,
    # You can add binary operations here.
}[expr.op]
```

step4：比较和逻辑表达式

在上述二元运算符的基础上继续添加对应规则：

```
op = {
    node.BinaryOp.Add: tacop.BinaryOp.ADD,
    node.BinaryOp.Sub: tacop.BinaryOp.SUB,
    node.BinaryOp.Mul: tacop.BinaryOp.MUL,
    node.BinaryOp.Div: tacop.BinaryOp.DIV,
    node.BinaryOp.Mod: tacop.BinaryOp.REM,
    node.BinaryOp.LT:  tacop.BinaryOp.SLT,
    node.BinaryOp.GT:  tacop.BinaryOp.SGT,
    node.BinaryOp.LogicAnd: tacop.BinaryOp.AND,
    node.BinaryOp.LogicOr:  tacop.BinaryOp.OR,
    node.BinaryOp.LE:  tacop.BinaryOp.LEQ,
    node.BinaryOp.GE:  tacop.BinaryOp.GEQ,
    node.BinaryOp.EQ:  tacop.BinaryOp.EQU,
    node.BinaryOp.NE:  tacop.BinaryOp.NEQ,
    # You can add binary operations here.
}[expr.op]
```

除此之外，由于一些运算符如 LEQ、AND 等无法通过一条 RISC-V 指令完成，因此还需要修改后端对于目标代码的生成。

具体来说，修改 backend/riscv/riscvasmemitter.py 中的 visitBinary() 函数：

```
def visitBinary(self, instr: Binary) -> None:
    if instr.op == BinaryOp.LEQ:
        self.seq.append(Riscv.Binary(BinaryOp.SGT, instr.dst, instr.lhs, instr.rhs))
        self.seq.append(Riscv.Unary(UnaryOp.SEQZ, instr.dst, instr.dst))
    elif instr.op == BinaryOp.GEQ:
        self.seq.append(Riscv.Binary(BinaryOp.SLT, instr.dst, instr.lhs, instr.rhs))
        self.seq.append(Riscv.Unary(UnaryOp.SEQZ, instr.dst, instr.dst))
    elif instr.op == BinaryOp.EQU:
        self.seq.append(Riscv.Binary(BinaryOp.SUB, instr.dst, instr.lhs, instr.rhs))
        self.seq.append(Riscv.Unary(UnaryOp.SEQZ, instr.dst, instr.dst))
    elif instr.op == BinaryOp.NEQ:
        self.seq.append(Riscv.Binary(BinaryOp.SUB, instr.dst, instr.lhs, instr.rhs))
```

```
        self.seq.append(Riscv.Unary(UnaryOp.SNEZ, instr.dst, instr.dst))
    elif instr.op == BinaryOp.AND:
        self.seq.append(Riscv.Unary(UnaryOp.SNEZ, instr.dst, instr.lhs))
        self.seq.append(Riscv.Binary(BinaryOp.SUB, instr.dst, 0, instr.dst))
        self.seq.append(Riscv.Binary(instr.op, instr.dst, instr.dst, instr.rhs))
        self.seq.append(Riscv.Unary(UnaryOp.SNEZ, instr.dst, instr.dst))
    elif instr.op == BinaryOp.OR:
        self.seq.append(Riscv.Binary(instr.op, instr.dst, instr.lhs, instr.rhs))
        self.seq.append(Riscv.Unary(UnaryOp.SNEZ, instr.dst, instr.dst))
    else:
        self.seq.append(Riscv.Binary(instr.op, instr.dst, instr.lhs, instr.rhs))

```

以 AND 为例，为实现 land 指令，我们使用了四条 RISC-V 指令来实现，即：

```
snez d, s1
neg d, d
and d, d, s2
snez d, d
```

实验思考题

step2：一元操作

1. 表达式为 -~2147483647，其中第一步经过 ~ 后，数值变为 -2147483648，再经过 - 取反得到 2147483648 即发生了越界。

step3：加减乘除模

1. 代码填写如下：

```
#include <stdio.h>

int main() {
  int a = -2147483648;
  int b = -1;
  printf("%d\n", a / b);
  return 0;
}
```

在本机（x86-64 架构）下运行后，产生 floating point exception：

```
[1]    67485 floating point exception  ./1
```

在 RISCV-32 的 qemu 模拟器中运行后，正常输出：

```
-2147483648
```

step4：比较和逻辑表达式

1. 我认为短路求值特性主要有以下优势：
    1. 加速运算：

        当有形如表达式 `A && B` 时，如果 A 已经为假，那么 B 的值不影响最后的结果，在对 B 求值的过程对后续代码无影响的前提下，我们可以省去对于 B 的计算而加速运算。

    2. 简化代码：

        考虑如下代码：

        ```
        if (i >= 0 && i < n && check(a[i])) {
            //...
        }
        ```

        如果没有短路特性，上述代码可能因为 `i` 不在合适的范围内而导致 `a[i]` 发生越界错误，因此代码必须修改为：

        ```
        if (i >= 0 && i < n)
            if (check(a[i]){
                //...
            }
        ```

        这样需要使用嵌套的 `if` 会导致代码编写难度加大。

        而带有短路特性可以增强程序员编写代码的灵活性，简化代码编写。

    3. 利于代码压缩：

        如：

        ```
        if (CONDITION) EXPRESSION;
        ```

        可以改写为：

        ```
        CONDITION && EXPRESSION;
        ```

        通过这样的变形，有利于必要的代码压缩。