Stage-1 Report

曹伦郗 *2020011020*

实验内容

step2

在*TACGen.visitUnary*中用于构建从*AST*的*op*映射到*TAC*的*op*的字典中，添加了按位取反、逻辑取反，共2项运算。

```
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]
```

step3

在*TACGen.visitBinary*中用于构建从*AST*的*op*映射到*TAC*的*op*的字典中，添加了减、乘、除、模，共4项运算。

```
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

在*TACGen.visitBinary*中用于构建从*AST*的*op*映射到*TAC*的*op*的字典中，添加了==、!=、<、>、<=、>=、逻辑与&&、逻辑或||，共8项运算。

```
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.EQ: tacop.BinaryOp.EQU,
    node.BinaryOp.NE: tacop.BinaryOp.NEQ,
    node.BinaryOp.LT: tacop.BinaryOp.SLT,
    node.BinaryOp.GT: tacop.BinaryOp.SGT,
    node.BinaryOp.LE: tacop.BinaryOp.LEQ,
    node.BinaryOp.GE: tacop.BinaryOp.GEQ,
    node.BinaryOp.LogicAnd: tacop.BinaryOp.AND,
    node.BinaryOp.LogicOr: tacop.BinaryOp.OR,
    # You can add binary operations here.
```

```
}[expr.op]
```

在*RiscvAsmEmitter.RiscvInstrSelector.visitBinary*中，==、!=、<=、>=、逻辑与&&、逻辑或||这6项运算没有直接对应的*RISC-V*指令，故为这些运算作手动的*RISC-V*指令翻译。而其余二元运算直接翻译为对应的*RISC-V*指令。

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

思考题

step2

*minidecaf*表达式为

```
-~2147483647
```

step3

```
#include <stdio.h>
int main()
{
    int a = -2147483648;
    int b = -1;
    printf("%d\n", a / b);
    return 0;
}
```

X86-64

```
Floating point exception
```

RISC-V/32

```
-2147483648
```

step4

短路求值受欢迎的原因：

这一特性使得在通过左操作数确定条件判断表达式的值时，可以避免计算右操作数，降低了程序计算量。

该特性为程序员带来的好处：

①利用它降低程序计算量，提升程序效率；

②可以利用该特性来确保某些可能出现运行时错误的表达式不会出现运行时错误。例如下面的代码即利用短路求值特性，避免了p为空指针时对p进行解引用所引起的运行时错误。

```
bool check_char(const char *p)
{
    return p != NULL && p[0] == 'a';
}
```