MiniDecaf-Stage1实验报告

计06 赵翊哲 2020010998

一、实验目的

本阶段的实验目的主要是构造常量表达式，完成基本的数学运算和逻辑比较运算，并在这个基础上熟悉整个项目的代码框架和运行流程。

构建常量表达式的过程主要分为：

- 添加一元操作符
- 添加二元操作符
    - 算术运算符
    - 添加比较和逻辑表达式

二、实验内容

2.1 Step 0 - 1

主要阅读了实验文档和项目代码，基本了解项目结构，明确编译的执行流程和核心实现文件。

2.2 Step 2 一元操作

- 修改内容：增加了一元操作符：取负 `-`、按位取反 `~` 以及逻辑非 `!`
- 实现：
    - 取负 `-` 在样例代码中已经实现
    - 按位取反 `~`：仿照 `tacgen.py` 中的 `visitUnary()` 函数中的实现，增加 `node.UnaryOp.BitNot: tacop.UnaryOp.NOT,` 实现按位取反
    - 逻辑非 `!`：在逻辑非中，核心是只有`!0=1`，因此可以通过 `tacop.UnaryOp.SEQZ` 实现，只对0置1，实现逻辑非
    - 核心修改代码如下：

```
op = {
  node.UnaryOp.Neg: tacop.UnaryOp.NEG,     # NEG: 负
  node.UnaryOp.BitNot: tacop.UnaryOp.NOT,  # NOT: 按位取反
  node.UnaryOp.LogicNot: tacop.UnaryOp.SEQZ,  # SEQZ: 对0置1
  # You can add unary operations here.
}[expr.op]
```

2.3 Step 3 算术二元操作

- 修改内容：增加了二元操作符：加 `+`、减 `-`、乘 `*`、整除 `/`、模 `%` 以及括号 `(` `)`。
- 实现：
    - 取负 `+` 在样例代码中已经实现

- 减 `-`：仿照 `tacgen.py` 中的 `visitBinary()` 函数中的实现，增加 `node.BinaryOp.Sub: tacop.BinaryOp.SUB,` 实现减法
- 其与操作符与加 `+`、减 `-` 类似，只需在 `op` 中添加对应关系即可
- 核心修改代码如下：

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

2.4 Step 4 逻辑二元操作、比较操作

- 修改内容：
  - 增加了比较大小和相等的二元操作：`<`、`<=`、`>=`, `>`, `==`, `!=`
  - 逻辑与 `&&`、逻辑或 `||`
- 实现：
  - 仿照 `tacgen.py` 中的 `visitBinary()` 函数中的实现，增加 `node.BinaryOp.LT: tacop.BinaryOp.SLt,` 等一系列关系即可，核心修改代码如下：

```
op = {
  # 比较运算
  node.BinaryOp.LT: tacop.BinaryOp.SLT,  # <
  node.BinaryOp.LE: tacop.BinaryOp.LEQ,  # <=
  node.BinaryOp.GE: tacop.BinaryOp.GEQ,  # >=
  node.BinaryOp.GT: tacop.BinaryOp.SGT,  # >
  node.BinaryOp.EQ: tacop.BinaryOp.EQU,  # ==
  node.BinaryOp.NE: tacop.BinaryOp.NEQ,  # !=
  # 逻辑运算
  node.BinaryOp.LogicAnd: tacop.BinaryOp.AND,  # &&
  node.BinaryOp.LogicOr: tacop.BinaryOp.OR,    # ||
  # You can add binary operations here.
}[expr.op]
```

  - 而 `NEQ` 、 `EQU` 、 `GEQ` 、 `LEQ` 四个指令在RISC-V内没有直接对应的实现指令，因此需要在 `riscvasmemitter.py` 中额外添加对其生成目标指令的处理，主要利用其他已实现的指令即可。对于 `NEQ` 、 `EQU` ，通过两个被比较数相减判断是否为零来等价处理判等；对于 `GEQ` 、 `LEQ` ，通过先判断 `SLT` 、 `SGT` ，获得一个暂时的结果，然后对其逻辑取反即可得到对应的大于等于、小于等于的判断结果；而逻辑与或则基于实验文档中的提示实现，具体代码如下：

```
def visitBinary(self, instr: Binary) -> None:
  if instr.op == tacop.BinaryOp.EQU:
```

```
    self.seq.append(Riscv.Binary(tacop.BinaryOp.SUB, instr.dst, instr.lhs,
instr.rhs))  # -
    self.seq.append(Riscv.Unary(tacop.UnaryOp.SEQZ, instr.dst, instr.dst))
  elif instr.op == tacop.BinaryOp.NEQ:
    self.seq.append(Riscv.Binary(tacop.BinaryOp.SUB, instr.dst, instr.lhs,
instr.rhs))  # -
    self.seq.append(Riscv.Unary(tacop.UnaryOp.SNEZ, instr.dst, instr.dst))
  elif instr.op == tacop.BinaryOp.GEQ:
    self.seq.append(Riscv.Binary(tacop.BinaryOp.SLT, instr.dst, instr.lhs,
instr.rhs))  # <
    self.seq.append(Riscv.Unary(tacop.UnaryOp.SEQZ, instr.dst, instr.dst))
  elif instr.op == tacop.BinaryOp.LEQ:
    self.seq.append(Riscv.Binary(tacop.BinaryOp.SGT, instr.dst, instr.lhs,
instr.rhs))  # >
    self.seq.append(Riscv.Unary(tacop.UnaryOp.SEQZ, instr.dst, instr.dst))
  elif instr.op == tacop.BinaryOp.AND:
    self.seq.append(Riscv.Unary(tacop.UnaryOp.SNEZ, instr.dst, instr.lhs))
    self.seq.append(Riscv.Binary(tacop.BinaryOp.SUB, instr.dst, Riscv.ZERO,
instr.dst))
    self.seq.append(Riscv.Binary(tacop.BinaryOp.AND, instr.dst, instr.dst,
instr.rhs))
    self.seq.append(Riscv.Unary(tacop.UnaryOp.SNEZ, instr.dst, instr.dst))
  elif instr.op == tacop.BinaryOp.OR:
    self.seq.append(Riscv.Binary(tacop.BinaryOp.OR, instr.dst, instr.lhs,
instr.rhs))
    self.seq.append(Riscv.Unary(tacop.UnaryOp.SNEZ, instr.dst, instr.dst))
  else: # 其与情况可直接使用RISC-V中的对应指令
    self.seq.append(Riscv.Binary(instr.op, instr.dst, instr.lhs, instr.rhs))
```

三、思考题

3.1 Step2 思考题

Q：我们在语义规范中规定整数运算越界是未定义行为，运算越界可以简单理解成理论上的运算结果没有办法保存在32位整数的空间中，必须截断高于32位的内容。请设计一个 minidecaf 表达式，只使用`-~!`这三个单目运算符和从 0 到 2147483647 范围内的非负整数，使得运算过程中发生越界。

> 32位可以表述的数据范围为 $[-2^{31}, 2^{31}-1]$，因此实现思路是先通过非负整数构造 -2^{31}，再对其取负即可得到越界的 2^{31}。

A：`-~2147483647` 会发生越界。

3.2 Step3 思考题

Q：我们知道“除数为零的除法是未定义行为”，但是即使除法的右操作数不是 0，仍然可能存在未定义行为。请问这时除法的左操作数和右操作数分别是什么？请将这时除法的左操作数和右操作数填入下面的代码中，分别在你的电脑（请标明你的电脑的架构，比如 x86-64 或 ARM）中和 RISCV-32 的 qemu 模拟器中编译运行下面的代码，并给出运行结果。（编译时请不要开启任何编译优化）

> 仿照Step2的思考题思路进行基于除法的越界设计即可，核心还是构造 2^{31}，利用int数据范围正负数表示区间差1的性质构造$(-2^{31}) \div (-1)$即可

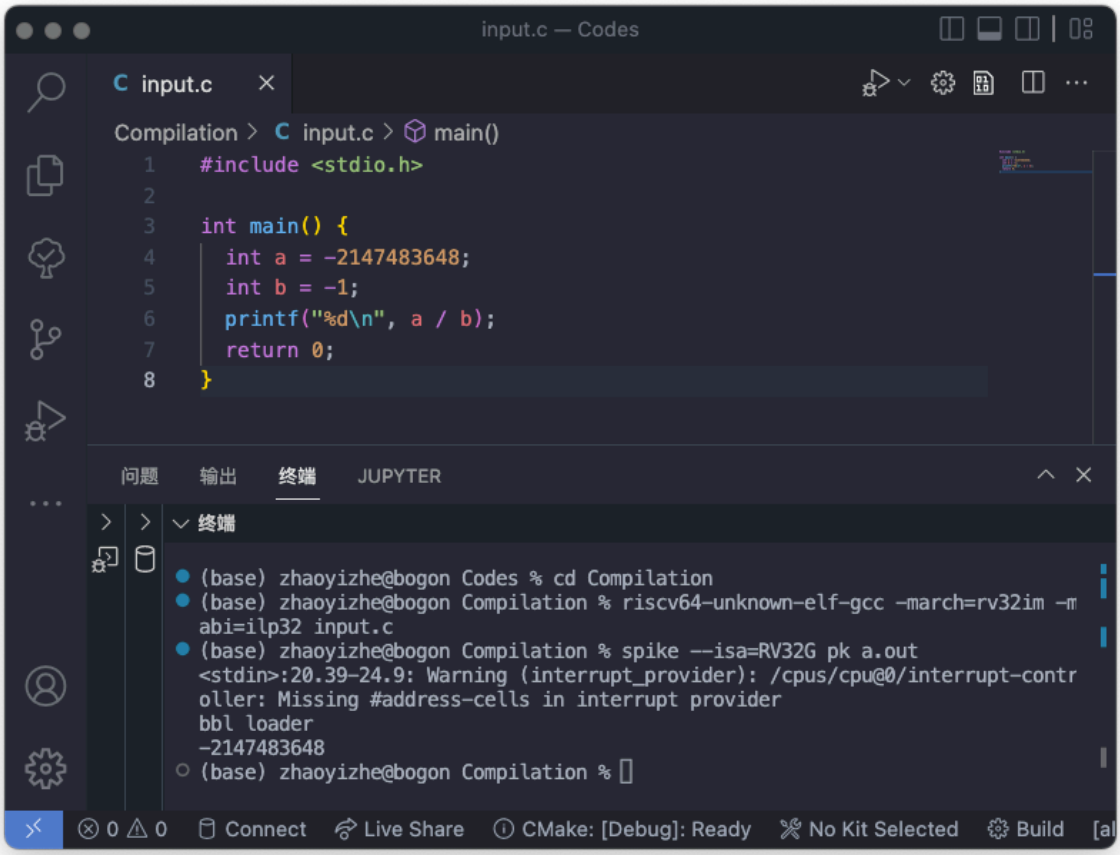
A：电脑架构：x86-64

```c
#include <stdio.h>

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

运行结果输出：`-2147483648`

运行结果截图：



3.3 Step4 思考题

Q：在 MiniDecaf 中，我们对于短路求值未做要求，但在包括 C 语言的大多数流行的语言中，短路求值都是被支持的。为何这一特性广受欢迎？你认为短路求值这一特性会给程序员带来怎样的好处？

A：短路求值的优点：（1）可以减少不必要的计算，比如在 `&&` 表达式中一项为假可以直接判断为假，在 `||` 表示中一项为真可以直接判断为真，提高了程序运行效率。（2）可以将具备依赖关系的条件表达式顺序放置，起到保护作用，例如 `if (t >= 0 && a[t] < 100)` ，可以保证在判断后一个条件时 t 一定是非负的，不会产生越界的情况。

对程序员的好处：可以提升程序运行效率，也可以避免在编写条件判断时设计更复杂的嵌套逻辑，提升代码简介性和可读性。

四、参考资料

实现过程中主要参考了实验文档和实验思路QA墙