八、优化部分

由于本学期数据库的截止时间与编译实验的截止时间高度重合，所以我优化部分做的不是很多，当然一方面也有自己的懒惰、保守等原因，总归确实还挺遗憾的。

8.1 乘除法优化

乘法优化：
先判断乘法的两个操作数有没有常数：

- 有常数，判断是否是2的幂次，若是，则转变为sll，若不是，用乘法指令
- 没有常数，用乘法指令

除法优化：
具体方法参见教程和往届各个大佬的博客就好，我这里就不多做赘述了，直接放上我的实现代码：

```
public void optimizeDiv(int targetReg, int regOp1, int d) {
    // 选择multiplier和shift部分
    int N = 32;
    int precision = N - 1;      //除数的大小描述，有 divisor < 2^precision
    int dAbs = Math.abs(d);         //除数的绝对值
    int log = N - Integer.numberOfLeadingZeros(dAbs - 1);
    int shift = log;
    double low = Math.floor(Math.pow(2, N + log) / dAbs);
    double high = Math.floor((Math.pow(2, N + log)
                    + Math.pow(2, N + log - precision)) / dAbs);
    while (Math.floor(low / 2.0) < Math.floor(high / 2.0) && shift > 0) {
        // 求shift
    }
    long m = (long) high;
    if (dAbs == 1) {
        // 结果等于被除数
    } else if (dAbs == Math.pow(2, log)) {      // 除2的幂
        // 类比乘法，转化为右移指令
    } else if (m < Math.pow(2, N - 1)) {        // multiplier比较小
        Li li = new Li(targetReg, (int) m);
        addMipsInstr(MipsInstrType.COMMON, li);
        Mult mult = new Mult(targetReg, regOp1);
        addMipsInstr(MipsInstrType.COMMON, mult);
        Mfhi mfhi = new Mfhi(targetReg);
        addMipsInstr(MipsInstrType.COMMON, mfhi);
        Sra sra1 = new Sra(targetReg, targetReg, shift);
        addMipsInstr(MipsInstrType.COMMON, sra1);
        Sra sra2 = new Sra(regOp1, regOp1, 31);
        addMipsInstr(MipsInstrType.COMMON, sra2);
        Subu subu = new Subu(targetReg, targetReg, regOp1);
        addMipsInstr(MipsInstrType.COMMON, subu);
    } else {
        long m1 = m - (long) Math.pow(2, N);
        Li li = new Li(targetReg, (int) m1);
        addMipsInstr(MipsInstrType.COMMON, li);
        Mult mult = new Mult(targetReg, regOp1);
        addMipsInstr(MipsInstrType.COMMON, mult);
        Mfhi mfhi = new Mfhi(targetReg);
        addMipsInstr(MipsInstrType.COMMON, mfhi);
        Addu addu = new Addu(targetReg, targetReg, regOp1);
        addMipsInstr(MipsInstrType.COMMON, addu);
        Sra sra1 = new Sra(targetReg, targetReg, shift);
        addMipsInstr(MipsInstrType.COMMON, sra1);
        Sra sra2 = new Sra(regOp1, regOp1, 31);
        addMipsInstr(MipsInstrType.COMMON, sra2);
        Subu subu = new Subu(targetReg, targetReg, regOp1);
```

```
            addMipsInstr(MipsInstrType.COMMON, subu);
        }
        if (d < 0) {
            // 负数的情况
            Subu subu = new Subu(targetReg, 0, targetReg);
            addMipsInstr(MipsInstrType.COMMON, subu);
        }
    }
```

8.2 基本块优化

mips中指令是顺序执行的，而LLVM中指令是一定要在一个基本块内执行，这就为我们带来了很多的优化空间，尤其是本人在实现LLVM中由于跳转语句的逻辑不是很清晰，带来了很多没有用的基本块和跳转语句，所以我选择这步优化是非常必要的且比较卓有成效。
其中基本块优化中我实现了合并基本块这部分，流程如下：

- 构建两个HashMap，分别是前驱集合和后继集合，两个集合的key都是 `BasicBlock` ，value都是 `ArrayList<BasicBlock>` 。代表着当前的基本块的前驱和后继集合。
- 构建基本块之间的流图。由于基本块的最后一条语句一定是一条出口语句（跳转语句或者返回语句），所以我们只需要看这个基本块最后一条语句是什么类型的语句即可，有以下这么几种情况：
  - `br Label` ，将当前基本块与Label进行双向链接，过程可类比双向链表
  - `br cond trueLabel, falseLabel` ，与两个目标基本块都进行连接
  - `return` ，没有后继，不用理会
- 我们合并基本块有两个基本条件，一个是当前基本块要和后继基本块相邻，另一个是该后继基本块不能有其他后继，如果满足上述两个条件，就可进行进一步的优化。
- 合并基本块的方法就是把该基本块的最后一条语句（一定是跳转语句）删掉，把后继基本块的指令全部加到该基本块中，删除后继基本块。这就要求我们的基本块内有且只有一条跳转语句，再之前生成LLVM的时候其实我是不能保证这一点的，所以完成这次优化我也顺便删除了一些不会被执行到的死代码。

8.3 指令选择

把所有的subiu都换成addiu，由于我本身用到的指令类型也不是很多，伪指令中有不合理展开的只有这一条，所以我就只优化了这一条代码。