# 代码优化设计

本次实现的优化为乘法优化， 除法优化和临时变量寄存器分配

本次优化遇到的最大困难还不少，首先应该还是疫情吧。较多的课程任务以及不太舒服的身体导致抽出来做优化的时间并不算很多。也导致了优化的不够完善，得到的结果也不算太好。

## 优化思路

看到相关的权重配比时，发现乘法和除法的权重较高，而且实现难度来看相比寄存器分配相对简单，需要改动的地方也相对较少。因此决定优化乘法和除法。
而后打算是大头的寄存器分配。在代码生成时，我是直接利用内存来存放变量的，寄存器只是一个中介。乘除优化都是针对生成的中间代码的，在中间代码生成的过程进行修改即可。

## 乘法优化

没找到很好的案例来优化，看到了一些针对特殊的情况优化, 也就是其可以转换为位运算计算的情况, 当乘数为

- $2^n$
- $2^n + 1$
- $2^n - 1$
- $2^m + 2^n$

其方法就是对于传入整数的乘数，判断是否与2的幂数相关

```
bool isPowerOfTwo(int n) {
    return n && (!(n & (n - 1)));
}

bool judge(int n, int &a, int &b) {
    for (int i = 1; i <= (int) sqrt(n); ++i) {
        if (n % i == 0) {
            if (isPowerOfTwo(i) and isPowerOfTwo(n / i - 1)) {
                a = (int) log2(i);
                b = (int) log2(n / i - 1);
                return true;
            } else if (isPowerOfTwo(n / i) and isPowerOfTwo(i - 1)) {
                a = (int) log2(n / i);
                b = (int) log2(i - 1);
                return true;
            }
        }
    }
    return false;
}
```

若相关就转换为位运算和加减。后面由二进制很自然的可以得出所有数都可以是2的幂次方之和，因此可以对乘数先进行分解，分解后进行位运算和加法运算。但这样其实无法肯定一定比乘法优，毕竟乘法的权值贡献不大，所以最后也就没有实现

# 除法运算

查看了一些编译器对于除法的优化，发现大多数都是利用"Magic Number"的理论进行除法优化。

说明如下：
设代码a/b = c　　b是常量
设b*k = 2^n　　2的n次直接用移位运算即可
a *2^n/ (b*2 ^n) = c
a * b *k/b = c* 2^n
a *k >> n =  c

因此对于每个除数需要获得其"Magic Number"

```
long long getMagicNumber(const BigInteger &divImm, int &leftBit, int &shPosition) {
      leftBit = 0;
      BigInteger twoBase(2);
      while (divImm > (twoBase ^ leftBit)) {
          leftBit++;
      }
      shPosition = leftBit;
//        cout << "leftBit: " << leftBit << endl;
      long long lowValue = ((twoBase ^ (leftBit + 32)) / divImm).toLongLong();
      long long highValue = (((twoBase ^ (leftBit + 32)) + (twoBase ^ (leftBit + 1))) / divImm).to
//        cout << lowValue << " " << highValue << endl;
      while ((highValue >> 1) > (lowValue >> 1) and shPosition > 0) {
          highValue >>= 1;
          lowValue >>= 1;
          shPosition--;
      }
      return highValue;
  }
```

待寻找到对应的MagicNumber之后，可以观察针对x86的架构对于除法进行的代码转换

```
mov eax,2E8BA2E9
imul ecx   //ecx是我们的a
sar edx,1
mov ecx,edx
shr ecx,1F
add edx,ecx
```

分析其过程可以知道

首先eax是由编译器生成的一个数，然后$a * eax$,得到的积低32位存储在eax中，高32位存储在edx中，如果积大于32位的话。

```
sar edx,1
```

则是直接取出高32位来看，相当于直接舍去了低32位，也就相当于先右移了32位。再右移1位，相当于我们得到的积总共向右移了33位

$a * (2E8BA2E9h >> 33)$等$a * (2E8BA2E9h/2^{33})$, $a * (2E8BA2E9h/2^{33}) \approx 0.090909$也就是所谓的倒数了。这边的2E8BA2E9也就是所谓的MagicNumber

第5行的右移31位是为了得到结果的符号位，这里主要是看符号位是否为1，也就是是否为负

数。如果是负数的话，符号位为1，最后的结果+1。
之所以要+1，这是因为负数算术右移跟负数除以2最后的结果其实是不同的，尽管常常认为右移1位就是除以2。
负数算术右移是向下取整的，换句话说负数, shr 的结果 + 1 = 负数除以2 的 结果 负数shr的结果+1 = 负数除以2的结果负数shr的结果+1=负数除以2的结果。所以最后要加1。

```
 if(objects[1]->getCategory() != 5 and objects[2]->getCategory() == 5) {
int dividend = objects[2]->getNum();
int result, shPosition, leftBit;
shared_ptr<IRCode> ircodeMagic, ircode1;
long long magicNumber = getMagicNumber(BigInteger(abs(objects[2]->getNum())), leftBit, shPosition)
if (abs(objects[2]->getNum()) == 1) {
    if (objects[2]->getNum() == 1) {
        ircode = make_shared<IRCode>(AssignOT, objects);
    } else {
        shared_ptr<Object> objects1[3];
        objects1[0] = objects[0];
        objects1[2] = objects[1];
        objects1[1] = make_shared<Object>(0);
        ircode = make_shared<IRCode>(MINUSOT, objects1);
    }
}
  else {
    // objects1 for positive or negative
    shared_ptr<Object> objectsMagic[3], objects1[3];
    objectsMagic[0] = objects[0];
    objectsMagic[1] = make_shared<Object>(magicNumber, "LONG");
    objectsMagic[2] = objects[1];
    ircodeMagic = make_shared<IRCode>(MULTHOT, std::move(objectsMagic));
    IRCodeList.push_back(std::move(ircodeMagic));
    if (magicNumber >= INF) {
        shared_ptr<Object> objects2[3];
        objects2[0] = generateTempVarT();
        objects2[1] = objects[1];
        objects2[2] = make_shared<Object>(1LL << 32, "LONG");
        shared_ptr<VarIRCode> var_temp = make_shared<VarIRCode>(objects2[0]->getName(), 0, false,
                                                                INTST,
                                                                currentDomain);
        varIndex += 1;
        if (currentDomain == 0) {
            global_offset += 1;
        }
        var_temp->setIsTemp(true);
        var_temp->setSpaceInfo(varIndex, 1);
        objects2[0]->setVar(var_temp);
        ircode1 = make_shared<IRCode>(MULTOT, objects2);
        IRCodeList.push_back(std::move(ircode1));
        objectsMagic[0] = objects[0];
        objectsMagic[1] = objects[0];
        objectsMagic[2] = objects2[0];
        ircodeMagic = make_shared<IRCode>(MINUSOT, std::move(objectsMagic));
        IRCodeList.push_back(std::move(ircodeMagic));
```

```
    }
    if (magicNumber >= INF) {
        objectsMagic[0] = objects[0];
        objectsMagic[1] = objects[0];
        objectsMagic[2] = before;
        ircodeMagic = make_shared<IRCode>(PLUSOT, std::move(objectsMagic));
        IRCodeList.push_back(std::move(ircodeMagic));
    }
    // >> shPosition
    objectsMagic[0] = objects[0];
    objectsMagic[2] = make_shared<Object>(shPosition);
    objectsMagic[1] = objects[0];
    ircodeMagic = make_shared<IRCode>(SRAOT, objectsMagic);
    IRCodeList.push_back(ircodeMagic);
    // get sign bit
    objects1[0] = generateTempVarT();
    objects1[1] = before;
    objects1[2] = make_shared<Object>(31);
    shared_ptr<VarIRCode> var_temp = make_shared<VarIRCode>(objects1[0]->getName(), 0, false,
                                                            INTST,
                                                            currentDomain);
    varIndex += 1;
    if (currentDomain == 0) {
        global_offset += 1;
    }
    var_temp->setIsTemp(true);
    var_temp->setSpaceInfo(varIndex, 1);
    objects1[0]->setVar(var_temp);
    ircode1 = make_shared<IRCode>(SRLOT, objects1);
    IRCodeList.push_back(ircode1);
    objects[1] = objects[0];
    objects[2] = objects1[0];
    ircode = make_shared<IRCode>(PLUSOT, objects);
    if (dividend < 0) {
        cout << "deal neg" << endl;
        objectsNeg[0] = objects[0];
        objectsNeg[1] = make_shared<Object>(0);
        objectsNeg[2] = objects[0];
        ircodeNeg = make_shared<IRCode>(MINUSOT, objectsNeg);
    }
}
```

在实际解决的过程会发现，计算MagicNumber的过程中有溢出的风险，因此对于C++选手就需要手写一个大整数类来应对了

```
class BigInteger {
    typedef bool Sign;

    friend ostream &operator<<(ostream &, const BigInteger &num);

    friend istream &operator<<(istream &, const BigInteger &num);

public:
    BigInteger() {
        numBits.clear();
    };

    BigInteger(const string &num);

    BigInteger(long long num);

    BigInteger(const BigInteger &num);

    BigInteger &operator=(const BigInteger &num);

    BigInteger operator-(const BigInteger &num) const;

    BigInteger operator+(const BigInteger &num) const;

    BigInteger operator*(const BigInteger &num) const;

    BigInteger operator/(const BigInteger &num) const;

    BigInteger operator^(const int num) const;

    long long toLongLong() const;

    bool operator==(const BigInteger &num) const;

    bool operator!=(const BigInteger &num) const;

    bool operator>(const BigInteger &num) const;

    bool operator<(const BigInteger &num) const;

    bool operator>=(const BigInteger &num) const;

    bool operator<=(const BigInteger &num) const;

private:
    static const int BASE = 10000;
```

```
    static const int BASE_LEN = 4;
    vector<int> numBits;

    static void fotmatting(BigInteger &num);
};
```

然后依照上述公式进行优化，就可以将常数除法优化掉，在竞速时也体现了出来。
mips相对有一个比x86方便地方，就是获得乘法结果的高32位，可以直接用mfhi指令。

# 寄存器分配

而后就是寄存器指派和分配，在截止日期，也就勉强完成了临时寄存器的分配，也就未能完成全局寄存器的分配，也就是图着色算法。

要对寄存器进行分配，第一步要先进行基本快的划分，这个相对简单，找到导致代码不顺序运行的操作符，分割代码块即可

```C++
void GenerateBlock::run() {
    vector<int> cutDots;
    bool flag = false;
    for (int i = 0; i < IRCodeList.size(); i++) {
        if (!flag) {
            if (IRCodeList[i]->getOp() == MainOT) {
                flag = true;
                setStart(i);
                continue;
            } else {
                continue;
            }
        }
       if (IRCodeList[i]->getOp() == JumpOT) {
           cutDots.push_back(i+1);
       } else if (IRCodeList[i]->getOp() == FuncOT) {
           cutDots.push_back(i);
       } else if (IRCodeList[i]->getOp() == CallOT) {
           cutDots.push_back(i+1);
       } else if (IRCodeList[i]->getOp() == ReturnOT) {
           cutDots.push_back(i+1);
       } else if (IRCodeList[i]->getOp() == LabelOT) {
           cutDots.push_back(i);
       }
    }
    sort(cutDots.begin(), cutDots.end());
    cutDots.erase(unique(cutDots.begin(), cutDots.end()), cutDots.end());
    int cnt = 0;
    while (cnt < cutDots.size()) {
        shared_ptr<BasicBlock> basicBlock;
        for (int i = cutDots[cnt]; i < (cnt == cutDots.size() - 1 ? IRCodeList.size() : cutDots[cnt+1]); i++) {
            if (i == cutDots[cnt]) {
                basicBlock = make_shared<BasicBlock>("B" + to_string(cnt));
            }
            basicBlock->addCode(IRCodeList[i]);
        }
        basicBlocks.push_back(basicBlock);
        ++cnt;
    }
}
```

临时寄存器分配的过程和书中的一致。建立一个临时寄存器池，检测到临时变量时，从池中取出一个寄存器，如果池中没有寄存器，采用的是FIFO方法，将其放入内存中。在基本块结束时，将所有临时寄存器中的变量放回内存中。与内存交互时，利用之前所写的load&save函数，将变

量放回内存中。

采用FIFO中，用一个变量来标记现在进行替换的寄存器是哪一个，如果该寄存器对应的变量在等式的右值，就顺延到下一个去。

```
void loadTempVar(MipsGenerator &mipsGenerator, shared_ptr<Object> &obj) {
    if (freeTempCnt > 0) {
        tempRegs[8 - freeTempCnt]->storeValue(mipsGenerator, obj);
        var2reg[obj->getVar()] = 8 - freeTempCnt;
        freeTempCnt -= 1;
    } else {
        var2reg.erase(tempRegs[pop_order]->getObject()->getVar());
        tempRegs[pop_order]->rollBack(mipsGenerator);
        tempRegs[pop_order]->storeValue(mipsGenerator, obj);
        var2reg[obj->getVar()] = pop_order;
        pop_order = (pop_order + 1) % 8;
    }
}

int moveTempVar(MipsGenerator &mipsGenerator, int idx, int idx2, fstream &output, shared_ptr<Objec
    if (idx2 == -1) {
        if (freeTempCnt > 0) {
            output << "move " << tempRegs[8 - freeTempCnt]->getId() << ", " << tempRegs[idx]->getI
                   << endl;
            tempRegs[idx]->moveTo(tempRegs[8 - freeTempCnt], obj);
            freeTempCnt -= 1;
            return 7 - freeTempCnt;
        } else {
            // accidentally same as the one to be moved, go to the next
            if (pop_order == idx)  pop_order = (pop_order + 1) % 8;
            tempRegs[pop_order]->rollBack(mipsGenerator);
            tempRegs[idx]->moveTo(tempRegs[pop_order], obj);
            output << "move " << tempRegs[pop_order]->getId() << ", " << tempRegs[idx]->getId() <<
            int last = pop_order;
            pop_order = (pop_order + 1) % 8;
            return last;
        }
    } else if (idx != idx2){
        output << "move " << tempRegs[idx2]->getId() << ", " << tempRegs[idx]->getId() << endl;
        tempRegs[idx]->moveTo(tempRegs[idx2], obj);
    }
    return idx2;
}
```

设计的寄存器类如下：

```
class Register {
private:
    bool isFree;
    shared_ptr<Object> object;
    string id;
public:
     Register(string id) : id(std::move(id)) {
        isFree = true;
    }

    void setFree(bool free) {
        isFree = free;
    }

    bool getFree() {
        return isFree;
    }

    void setObject(shared_ptr<Object> &obj) {
        object = obj;
    }

    shared_ptr<Object>& getObject() {
        return object;
    }

    string getId() {
        return id;
    }

    void rollBack(MipsGenerator &mipsGenerator);

    void storeValue(MipsGenerator &mipsGenerator, shared_ptr<Object> &obj);

    static void moveTo(shared_ptr<Register> &reg,  shared_ptr<Object> &obj);

};
```

rollback用于将寄存器储存的变量返回内存中，storeValue用于将变量放入寄存器中，moveTo用于将一个变量放入另一个寄存器中。

因此对于需要使用临时变量的指令，需要检查其是否在寄存器中（可以依靠map来帮助实现），如果在寄存器中，就直接使用，如果不在，就将其放入寄存器中，然后使用。

**存在的缺点**：

在于临时寄存器分配不够完善，临时寄存器检查的标准仅为是否是在运算时generate出来的，其真正的分配应该为是否跨越了基本块，导致效率提升有限。
并且利用FIFO也不是最优的替换方案，追求最佳，因为其已知了整个基本快是可以计算最远的使用距离太替换的，此外LRU也会比FIFO好不少。但时间问题也都没有实现。