优化文章

我实现的优化主要包括：死代码删除，Mem2Reg，寄存器分配，常量折叠，乘除法优化等

死代码删除

在死代码删除方面，我主要进行了三个维度的代码优化。

- 不可达的基本块：从每一个函数的**entryBb**开始，将基本块视作节点，`jump`或者`branch`指令视作两个基本块之间的有向边进行dfs遍历，删除未到达的基本块
- 未调用的函数：类似于基本块删除，将各个函数视作节点，函数调用视作节点之间的边，进行dfs遍历。删除不可到达的函数
- def-use分析：根据定义-使用链删除未使用的load指令等

```
public static void deleteDeadFunc(LlvmModule module) {
        ArrayList<Function> liveFuncs = new ArrayList<>();
        Function mainFunc = module.getMainFunc();
        findLiveFunc(mainFunc, liveFuncs);
        module.saveOnlyFunctions(liveFuncs);
    }

    public static void findLiveFunc(Function func, ArrayList<Function> liveFuncs)
{
        if (!liveFuncs.contains(func)) {
            liveFuncs.add(func);
            HashSet<Function> callFuncs = new HashSet<>();
            for (BasicBlock bb : func.getBasicBlocks()) {
                for (Instruction inst : bb.getInstructions()) {
                    if (inst instanceof Call) {
                        Function callFunc = ((Call) inst).getFunction();
                        callFuncs.add(callFunc);
                    }
                }
            }
            for (Function f : callFuncs) {
                // TODO 是否需要特判如果调用的是库函数的话，忽略访问， 但是应该不需要
                findLiveFunc(f, liveFuncs);
            }
        }
    }

    public static void deleteDeadBBlock(LlvmModule module) {
        for (Function func : module.getFunctions()) {
            ArrayList<BasicBlock> liveBBlocks = new ArrayList<>();
            BasicBlock entryBlock = func.getBasicBlock(0);
            findLiveBBlock(entryBlock, liveBBlocks);   // entryBBlock是函数入口的
BasicBlock
            func.saveOnlyBBlocks(liveBBlocks);    // 删除不是liveBBlocks中的BBlock
        }
    }

    public static void findLiveBBlock(BasicBlock bb, ArrayList<BasicBlock>
liveBBlocks) {
```

```
        if (!liveBBlocks.contains(bb)) {
            liveBBlocks.add(bb);
            Instruction endInstr = bb.getLastInstr();
//            for(Instruction endInstr : bb.getInstructions())  // TODO 这么写好像
有问题
            {
                if (endInstr instanceof Jump) {
                    BasicBlock targetBBlock = (BasicBlock) ((Jump)
endInstr).getTargetBBlock();
                    findLiveBBlock(targetBBlock, liveBBlocks);
                } else if (endInstr instanceof Branch) {
                    BasicBlock trueBBlock = (BasicBlock) ((Branch)
endInstr).getTrueBBlock();
                    findLiveBBlock(trueBBlock, liveBBlocks);
                    BasicBlock falseBBlock = (BasicBlock) ((Branch)
endInstr).getFalseBBlock();
                    findLiveBBlock(falseBBlock, liveBBlocks);
                }
            }

        }
    }
```

Mem2Reg优化

在进行Mem2Reg的优化之前，我们需要先进行一些准备工作，包括控制流图CFG计算，支配关系分析，活跃变量分析

准备工作

控制流图CFG计算

在`BasicBlock.Java`中增加以下属性

```
    private HashSet<BasicBlock> frontBBlocks = new HashSet<>();    // 前驱基本块
    private HashSet<BasicBlock> backBBlocks = new HashSet<>();    // 后继基本块
```

具体的计算方法主要是从每个函数的`entryBb`开始，将基本块视作节点，`jump`或者`branch`指令视作两个基本块之间的有向边进行dfs遍历，记录每个基本块的前驱与后继。这一点与基本块删除非常像

```
public void findFrontAndBackBBlock() {
        if (this.isVisited()) {
            return;
        }
        this.setIsVisited();
        Instruction ins = this.getLastInstr();
        if (ins instanceof Jump) {
            BasicBlock targetBBlock = (BasicBlock) ((Jump)
ins).getTargetBBlock();
            this.addBackBBlock(targetBBlock);
            targetBBlock.addFrontBBlock(this);
            targetBBlock.findFrontAndBackBBlock();
        } else if (ins instanceof Branch) {
            BasicBlock trueBBlock = (BasicBlock) ((Branch) ins).getTrueBBlock();
```

```
            BasicBlock falseBBlock = (BasicBlock) ((Branch)
ins).getFalseBBlock();

            this.addBackBBlock(trueBBlock);
            trueBBlock.addFrontBBlock(this);
            trueBBlock.findFrontAndBackBBlock();

            this.addBackBBlock(falseBBlock);
            falseBBlock.addFrontBBlock(this);
            falseBBlock.findFrontAndBackBBlock();
        }
    }
```

支配关系计算

基础支配关系根据教程所讲采用迭代计算方法计算

> 迭代计算。按照"某基本块的dom <- 某基本块所有前驱的dom的交集加上自己本身" 的策略进行更新，直到该基本块的dom集合不发生变化

```
public void calcDomBys() {
        // 初始化入口BBlock的支配关系
        BasicBlock entryBlock = this.getBasicBlock(0);
        entryBlock.domBys = new HashSet<>() {{
            add(entryBlock);
        }};   // 入口处的初始domys仅有自身

        for (int i = 1; i < basicBlocks.size(); i++) {
            basicBlocks.get(i).domBys = new HashSet<>(basicBlocks);
        }

        boolean needRecur = true;
        while (needRecur) {
            needRecur = false;
            for (int i = 1; i < this.basicBlocks.size(); i++) {
                BasicBlock bb = this.basicBlocks.get(i);

                // 如果没有前驱块，则只被自己支配（不可达块）
                if (bb.getFrontBBlocks().isEmpty()) {
                    HashSet<BasicBlock> newDomBys = new HashSet<>();
                    newDomBys.add(bb);
                    if (!bb.domBys.equals(newDomBys)) {
                        needRecur = true;
                    }
                    bb.domBys = newDomBys;
                    continue;
                }

                HashSet<BasicBlock> newDomBys = new HashSet<>(basicBlocks);
                for (BasicBlock frontBB : bb.getFrontBBlocks()) { // 取交集
                    HashSet<BasicBlock> tempDomBys = new HashSet<>();
                    for (BasicBlock frontBBDomBy : frontBB.domBys) {
                        if (newDomBys.contains(frontBBDomBy)) {
                            tempDomBys.add(frontBBDomBy);
                        }
```

```
                    }
                    newDomBys = tempDomBys;
                }
                newDomBys.add(bb);
                if (!bb.domBys.equals(newDomBys)) {
                    needRecur = true;
                }
                bb.domBys = newDomBys;
            }
        }
```

直接支配关系与严格支配关系计算可以从基础支配中直接计算

```
 public void calcImmeDom() {
        // 先清空所有块的直接支配关系
        for (BasicBlock bb : basicBlocks) {
            bb.immeDomTos.clear();
            bb.immeDomBy = null;
        }

        for (int i = 1; i < this.basicBlocks.size(); i++) {
            // 跳过第一个基本块，其不被直接支配
            // TODO 不跳过好像也没有关系
            BasicBlock curBb = this.basicBlocks.get(i);
            // 计算直接支配bb的基本块

            // 遍历基础支配bb的基本块，判断其是否是距离最近的
            // 直接支配者（immediate dominator, idom）：严格支配n，且不严格支配任何严格支配 n 的节点的节点(直观理解就是所有严格支配n的节点中离n最近的那一个)，我们称其为n的直接支配者
            for (BasicBlock domBy : curBb.domBys) {
                if (domBy.equals(curBb)) {
                    continue;  // 不属于严格支配，从而不属于直接支配
                }

                boolean isImmeDom = true;
                for (BasicBlock otherDomBy : curBb.domBys) {
                    // 判断other是否被domby支配
                    if (otherDomBy.equals(domBy)) {
                        continue;
                    }

                    if (otherDomBy.equals(curBb)) {
                        continue;
                    }

                    if (otherDomBy.domBys.contains(domBy)) {   // domBy严格支配otherDomBy，所以肯定不是直接支配
                        isImmeDom = false;
                        break;
                    }
                }
                if (isImmeDom) {
                    curBb.immeDomBy = domBy;
                    domBy.immeDomTos.add(curBb);
                    break;
```

```
                }
            }
        }
    }
```

支配边界：这里贴一张教程中给出的伪代码图片

Algorithm 3.2: Algorithm for computing the dominance frontier of each CFG node.

```
for (a, b) ∈ CFG edges do
    x ← a
    while x does not strictly dominate b do
        DF(x) ← DF(x) ∪ b
        x ← immediate dominator(x)
```

插入Phi指令

总的来说：通过定义使用链，我们可以分析出哪些基本块出现了变量汇聚，此处即是需要插入phi指令的地方。

Algorithm 3.1: Standard algorithm for inserting ϕ-functions

```
for v: variable names in original program do
    F ← {}                          ▷ set of basic blocks where φ is added
    W ← {}                          ▷ set of basic blocks that contain definitions of v
    for d ∈ Defs(v) do
        let B be the basic block containing d
        W ← W ∪ {B}
    while W ≠ {} do
        remove a basic block X from W
        for Y: basic block ∈ DF(X) do
            if Y ∉ F then
                add v ← φ(...) at entry of Y
                F ← F ∪ {Y}
                if Y ∉ Defs(v) then
                    W ← W ∪ {Y}
```

```
    public static void addPhi(Alloca alloca) {
        LLVMType varType = ((PointerType) alloca.getType()).getTargetType();
        for (Use use : alloca.getUses()) {
            User user = use.getUser();    // 使用该alloca的地方， 一般只有store与load指令
            BasicBlock host = (BasicBlock) user.getHost();    // user指令所属的基本块
            if (!host.isLived()) {     // 如果该基本块已经消除，则不进行多余操作。因为即使删除了基本块，但是user-use关系没有删除
                continue;
            }
            if (user instanceof Store store) {
                if (!writeBbList.contains(host)) {
                    writeBbList.add(host);
```

```
                }
                writeInstrList.add(store);
            } else if (user instanceof Load load) {
                if (!readBbList.contains(host)) {
                    readBbList.add(host);
                }
                readInstrList.add(load);
            }
        }

        HashSet<BasicBlock> visited = new HashSet<>();
        ArrayList<BasicBlock> workBBList = new ArrayList<>(writeBbList);    // 对于alloca进行写入的基本块，进行了重定义，其支配边界需要进行phi

        while (!workBBList.isEmpty()) {
            BasicBlock curBb = workBBList.remove(0);
            for (BasicBlock dfBb : curBb.domFro) {
                if (!visited.contains(dfBb)) {
                    visited.add(dfBb);
                    if (!writeBbList.contains(dfBb)) {
                        workBBList.add(dfBb);   // TODO
                    }
                    Phi phi = IRBuilder.makePhi(dfBb, varType);     // 写入块的支配边界意味着该变量可能有来自多重定义，插入phi指令

                    readInstrList.add(phi);
                    writeInstrList.add(phi);
                }
            }
        }
    }
```

重命名

插入phi指令后，def-use关系需要更新。同时进行dfs遍历，删除不必要的load指令

```
public static void rename(Alloca alloca, BasicBlock curBb) {
        int counter = 0;      // 记录当前基本块对alloca对应的变量进行了几次写入，用于维护writeStack
        Iterator<Instruction> iter = curBb.getInstructions().iterator();
        while (iter.hasNext()) {
            Instruction instr = iter.next();
            if (instr.equals(alloca)) {
                iter.remove();
            } else if (instr instanceof Phi phi && writeInstrList.contains(phi)) {  // 是针对当前alloca变量的phi指令
                // phi指令是针对于该变量的新一个定义点
                writeStack.push(phi);    // writeStack栈顶的是alloca的最新定义
                counter = counter + 1;
            } else if (instr instanceof Store store && writeInstrList.contains(store)) {
                counter = counter + 1;
                writeStack.push(store.getValue4Store());
                store.dropAllReferences();
                iter.remove();
```

```
            } else if (instr instanceof Load load &&
readInstrList.contains(load)) {    // 是当前该变量的读取指令
                // 此处不应该从内存中load变量的值，直接取stack栈顶的即是该变量的最新定义值
                Value curValue = writeStack.empty() ? new
ConstantData(LLVMType.INT32, 0, true) : writeStack.peek();
                // 如果写入栈为空，则读取内存默认值0，否则读取最新定义点(stack栈顶)

                // 删除前需要更新所有使用load的value与load使用的value的信息
                // 所有需要使用到load指令值的地方全部换成curValue
                load.replacAllUsesWith(curValue);
                // 删除load用到的value中存储的uses关系
                load.dropAllReferences();
                iter.remove();
            }
        }

        // 对于所有后继基本块中的该变量的phi指令，将来自于curBb的value填入phi中
        for (BasicBlock backBB : curBb.getBackBBlocks()) {
            // 遍历基本块中的所有phi指令（不再假设phi只在开头）
            for (Instruction instr : backBB.getInstructions()) {
                if (instr instanceof Phi phi && readInstrList.contains(phi)) {
                    Value valueFromCurBb = writeStack.empty() ? new
ConstantData(LLVMType.INT32, 0, true) : writeStack.peek();
                    phi.fill(curBb, valueFromCurBb);
                }
            }
        }

        // 对于curBb的直接支配块，在alloca的变量使用上是一体的，直接支配块可以继承当前块的写
入栈
        for (BasicBlock immeDomTo : curBb.immeDomTos) {
            rename(alloca, immeDomTo);
        }

        // 退出当前写入栈，避免对兄弟路径造成影响
        while (counter > 0) {
            counter--;
            writeStack.pop();
        }

    }
```

RemovePhi

phi指令与其他LLVM指令不同，不可以通过一条或者几条mips指令的组合来转换。需要总和考虑控制流等多个因素。

由于基本块多后继问题，我们在原有控制流路径中插入新的基本快，在该基本块中完成phi对应变量的move

ing line 13, replacing a'_i by a_0 in the following lines, and adding "remove the ϕ-function" after them.

Algorithm 3.5: Critical Edge Splitting Algorithm for making non-conventional SSA form conventional.

```
1  foreach B: basic block of the CFG do
2      let (E_1, ..., E_n) be the list of incoming edges of B
3      foreach E_i = (B_i, B) do
4          let PC_i be an empty parallel copy instruction
5          if B_i has several outgoing edges then
6              create fresh empty basic block B'_i
7              replace edge E_i by edges B_i → B'_i and B'_i → B
8              insert PC_i in B'_i
9          else
10             append PC_i at the end of B_i
11     foreach φ-function at the entry of B of the form a_0 = φ(B_1 : a_1, ..., B_n : a_n) do
12         foreach a_i (argument of the φ-function corresponding to B_i) do
13             let a'_i be a freshly created variable
14             add copy a'_i ← a_i to PC_i
15             replace a_i by a'_i in the φ-function
```

同时由于phi指令的并行赋值问题，我们需要插入临时中间变量来解决冲突

Algorithm 3.6: Replacement of parallel copies with sequences of sequential copy operations.

```
1  let pcopy denote the parallel copy to be sequentialized
2  let seq = () denote the sequence of copies
3  while ¬[∀(b ← a) ∈ pcopy, a = b] do
4      if ∃(b ← a) ∈ pcopy s.t. ∄(c ← b) ∈ pcopy then        ▷ b is not live-in of pcopy
5          append b ← a to seq
6          remove copy b ← a from pcopy
7      else                                                  ▷ pcopy is only made-up of cycles; Break one of them
8          let b ← a ∈ pcopy s.t. a ≠ b
9          let a' be a freshly created variable
10         append a' ← a to seq
11         replace in pcopy b ← a into b ← a'
```

```java
public static void insertCopy2Bb(BasicBlock curBb, HashMap<Value, Reg> value2Reg)
{
        // 将curBb的Phi指令转换为Copy指令，按照前驱块分组
        HashMap<BasicBlock, ArrayList<Copy>> frontBB2Copy = tranPhi2Copy(curBb);

        // 复制前驱块列表，避免在循环中修改原集合导致ConcurrentModificationException
        ArrayList<BasicBlock> frontBBList = new ArrayList<>
(curBb.getFrontBBlocks());
        for (BasicBlock frontBB : frontBBList) {
            ArrayList<Copy> copyList = frontBB2Copy.get(frontBB);
            if (copyList == null || copyList.isEmpty()) {
                continue;
```

```
            }
            // 解决Phi并行化赋值与copy序列行为不相符的问题
            ArrayList<Copy> paralleCopyList = solveParallelError(copyList,
curBb);
            // TODO 真的会出现寄存器共用带来的问题吗？？？
            ArrayList<Copy> endCopyList = solveRegError(paralleCopyList, curBb,
value2Reg);
            // 将copy指令列表插入到frontBB与curBb的路径中间

            // frontBB的后继只有curBb，则copy直接插入到frontBB后面去
            if (frontBB.getBackBBlocks().size() == 1) {
                for (Copy copy : endCopyList) {
                    copy.setHost(frontBB);  // 设置 copy 的 host
                    ArrayList<Instruction> instrs = frontBB.getInstructions();
                    int index = instrs.size() - 1;
                    instrs.add(index, copy);
                }
            } else {
                // 生成一个中间块：frontBB→中间块→curBb
                Function hostFunc = (Function) curBb.getHost();
                BasicBlock midBb = new BasicBlock(IRBuilder.namecnt++);
                midBb.setHost(hostFunc);
                hostFunc.addBasicBlockBeforBb(curBb, midBb);   // 插入新的基本块
                for (Copy copy : endCopyList) {
                    midBb.addInstruction(copy);
                    copy.setHost(midBb);
                }
                Jump jump = new Jump(curBb);   // 添加跳转指令
                jump.setHost(midBb);
                midBb.addInstruction(jump);    // 将跳转指令添加到中间块

                // 修改frontBB最后一条跳转指令(curBB为midBb)
                Branch branch = (Branch) frontBB.getLastInstr();
                branch.replaceOperand(curBb, midBb);

                // 更新CFG前驱后继关系
                // frontBB -> midBb -> curBb
                frontBB.getBackBBlocks().remove(curBb);
                frontBB.getBackBBlocks().add(midBb);
                midBb.addFrontBBlock(frontBB);
                midBb.addBackBBlock(curBb);
                curBb.getFrontBBlocks().remove(frontBB);
                curBb.getFrontBBlocks().add(midBb);
            }

        }
    }
```

> 易错：在插入基本块后，没有及时维护CFG控制流图，导致后分析出错

寄存器分配

我采用的是基于引用计数的寄存器分配策略

- 遍历各个指令统计各个值的使用次数与生存周期，为使用次数多的值赋予较高的权重，减少不必要的内存访问。同时在变量死亡(最后一次使用)之后，及时释放寄存器。
- `$v1,$t0-$t9, $s0-$s7, $fp`等寄存器参与寄存器分配

在开始寄存器分配之前，我们需要进行活跃变量分析，统计各个基本块的`in-out`集合

```
/// 活跃变量分析
    /// out[-BB] = ∪ in[-后继块]
    /// in[-BB] = useSet[-BB] ∪ (out[-BB] - defSet[-BB])
    public void analyzeActivateVar() {
        for (Function func : module.getFunctions()) {
            // 基本块的in与out集合
            HashMap<BasicBlock, HashSet<Value>> bbLiveInSets = new HashMap<>();
            HashMap<BasicBlock, HashSet<Value>> bbLiveOutSets = new HashMap<>();
            // 初始化，计算各个基本块的def-use集合，初始化各个基本块的in-out集合为空
            for (BasicBlock bb : func.getBasicBlocks()) {
                bb.computeDefUseSet();
                bbLiveInSets.put(bb, new HashSet<>());
                bbLiveOutSets.put(bb, new HashSet<>());
            }

            boolean flag = false;    // 活跃变量分析是否结束(in不再变化)
            while (!flag) {
                flag = true;
                // 控制流图逆序分析---收敛更快
                ArrayList<BasicBlock> bbs = func.getBasicBlocks();
                for (int i = bbs.size() - 1; i >= 0; i--) {
                    BasicBlock curBB = bbs.get(i);
                    HashSet<Value> outSet = new HashSet<>();
                    HashSet<Value> inSet = new HashSet<>();

                    // out[-BB] = ∪ in[-后继块]
                    for (BasicBlock backBB : curBB.getBackBBlocks()) {
                        outSet.addAll(bbLiveInSets.get(backBB));
                    }

                    // in[-BB] = useSet[-BB] ∪ (out[-BB] - defSet[-BB])
                    inSet.addAll(outSet);
                    for (Value v : curBB.defSet) {
                        inSet.remove(v);
                    }
                    inSet.addAll(curBB.useSet);

                    if (!outSet.equals(bbLiveOutSets.get(curBB)) ||
!inSet.equals(bbLiveInSets.get(curBB))) {
                        flag = false;
                    }

                    bbLiveInSets.put(curBB, inSet);
                    bbLiveOutSets.put(curBB, outSet);
                    curBB.liveInSet = inSet;
                    curBB.liveOutSet = outSet;
                }
            }
        }
```

```
    }
```

之后，根据各个变量的使用频率来进行寄存器分配

```
public void allocReg4Func(Function func) {
        value2citeNum = new HashMap<>();
        reg2value = new HashMap<>();
        value2reg = new HashMap<>();
        freeRegs = getCanAllocaRegs();
        ArrayList<Instruction> instrs = func.getAllInstrs();
        int size = instrs.size();

        // 计算引用权重，考虑循环深度（循环内的值权重更高）
        for (int i = 0; i < size; i++) {
            Instruction instr = instrs.get(i);
            BasicBlock bb = (BasicBlock) instr.getHost();


            if (!instr.getType().isVoid()) {
                Double doubleVal = value2citeNum.get(instr);
                if (doubleVal != null) {
                    double tmp = doubleVal + 1;
                    value2citeNum.put(instr, tmp);
                } else {
                    double tmp = 1;
                    value2citeNum.put(instr, tmp);
                }
            }
            for (Value v : instr.getOperands()) {
                Double doubleVal = value2citeNum.get(v);
                if (doubleVal != null) {
                    double tmp = doubleVal + 1;
                    value2citeNum.put(v, tmp);
                } else {
                    double tmp = 1;
                    value2citeNum.put(v, tmp);
                }
            }
        }

        BasicBlock entryBb = func.getFirstBasicBlock();
        allocReg4BBlock(entryBb);

        // 计算call调用前应该存活的寄存器
        for (BasicBlock bb : func.getBasicBlocks()) {
            ArrayList<Instruction> bbInstrs = bb.getInstructions();
            for (int i = 0; i < bbInstrs.size(); i++) {
                Instruction instr = bbInstrs.get(i);
                if (!(instr instanceof Call)) {
                    continue;
                }
                // out块中活跃的变量
                HashSet<Reg> activeRegs = new HashSet<>();
                for (Value v : bb.liveOutSet) {
                    if (value2reg.containsKey(v)) {
```

```
                    Reg r = value2reg.get(v);
                    activeRegs.add(r);
                }
            }
            // 当前基本块后续指令依旧需要的寄存器中的值
            for (int j = i + 1; j < bbInstrs.size(); j++) {
                Instruction instr2 = bbInstrs.get(j);
                for (Value v : instr2.getOperands()) {
                    if (value2reg.containsKey(v)) {
                        Reg r = value2reg.get(v);
                        activeRegs.add(r);
                    }
                }
            }
            ((Call) instr).liveRegSet = activeRegs;
        }
    }

    func.setValue2Reg(value2reg);
}
```

其余优化

常量折叠

在LLVM阶段进行如下优化

```
a + 0 = a
a - 0 = a
a * 0 = 0
2 * a = a + a
a * 1 = a
a / 1 = a
a / a = 1
```

乘除优化

乘法优化我只判断了常数是否为2的整数次幂，并将其转换为移位运算

除法优化思路可根据教程得到：

> **2. 除法优化**
>
> 除法优化的思路是将除法指令转化为乘法指令和移位指令，即如下公式：
>
> $$quotient = \frac{dividend}{divisor} = (dividend * multiplier) >> shift$$
>
> 该公式先乘一个较大的常数，然后用右移*shift*位得到最终的答案，使用该式子计算除法的核心是如何得到*multiplier*，另外由于mips架构是32位的，还必须要考虑溢出的问题。mips在执行乘法的时候会将结果保存到hi和lo寄存器当中，在被除数乘*multiplier*之后，很容易就会超过32位，这个就会出现一部分答案在hi寄存器中，一部分答案在lo寄存器当中，所以我们需要使*multiplier*尽量大（超过2^{32}），能够使答案的部分在hi寄存器当中。所以最终的公式可以写为如下形式
>
> $$quotient = \lfloor \frac{n}{d} \rfloor = \lfloor \frac{m * n}{2^{N+l}} \rfloor$$
>
> 其中N是机器码长度32，最终获得答案的公式为$SRL(MULUH(m,n), shift)$，$MULUH$表示乘法之后取HI寄存器。