代码优化

优化成果

除了test6为70+之外，截至12.11日其余所有点均在前50

前端拦截

本部分主要是在前端做的努力

由于前端相比于中端和后端，对代码有着更宏观的把控，所以有一部分东西在这里完成

乘除优化

变量和常量计算

如果对于一个乘法mulExp，，其factor序列为a1,a2...an若前ai个均为常数，则这前ai个可以化简为一个常数

如果其中有一个0，则整个就是0。在前端部分进行优化

具体策略是：首先获取所有factor的irCode，如果其中有constant且为0，那么本mulExp返回的结果是Constant(0)

乘法优化

constant - constant

不生成指令，直接计算出来结果，并代替该指令得到的值

包括其他所有计算类指令也一样

constant - x

通过shift比较计算.

具体策略是:

根据constant的值进行2的幂拆分，看看能表示成几个sll运算

如果计算出来的长度显示其性能超过直接相乘的，那么就将乘法替换为多个sll相加

```
1 public static ArrayList<Integer> shiftList(int num) {
```

```
    // 一次shift的代价是1，一次add的代价也是1，计算其代价为：
    // ans[0]==0? 2*len-2/2*len-1
    // 和5比较: 2*len-1<=5
    // len<=3
    // 注意如果是0的情况,那么会优先判断得到0
    ArrayList<Integer> ans = new ArrayList<>();
    int bit = 0;
    while (num != 0) {
        if (num % 2 == 1) {
            ans.add(bit);
        }
        num /= 2;
        bit++;
    }
    if (ans.size() <= 3) {
        return ans;
    }
    return null;
}
```

x - x

对此没有进行任何优化

除法优化

碍于时间，并不能来得及完成该部分的内容，暂时搁置

中端优化

中端是优化的核心，在这里我进行了以下的优化

```
new Mem2Reg().solve(irUnit);
new PhiRemove().phi2Pc(irUnit);
new GVN().solve(irUnit);
new DeadCodeRemove().solve(irUnit);
new PhiRemove().pc2Move(irUnit);
new GVN().solve(irUnit);
new DeadCodeRemove().solve(irUnit);
new ActiveAnalysis().solve(irUnit);
new RegAlloc().solve(irUnit);
```

CFG

首先解决的是多余代码问题，由此生成真正的基本块

比如在

```
if(a>1)
{
 b=b+1;
 return 0;
}
```

这个代码中，我会生成:

```
b1:
  ret i32 1
  br label %b3
b2:
  ret i32 2
  br label %b3
```

这种形式的，这显然是会错误估计支配边界的。

因此，我们做一个检查，对于一个block，如果遇到了一个离开指令，那么删除后面所有的代码。

```
public void cleanInstrAfterOut() {
        int mark = instrList.size();
        for (int i = 0; i < instrList.size(); i++) {
            Instr instr = instrList.get(i);
            if (instr instanceof BranchInstr || instr
instanceof JumpInstr || instr instanceof ReturnInstr) {
                mark = i;
                break;
            }
        }
        // 0-mark共mark+1个指令
        while (instrList.size() > mark + 1) {
            instrList.remove(mark + 1);
        }
    }
```

然后给出我们进行流分析的具体流程

(本部分按照面向对象的方法书写，因此下面给出的顶层过程是会下降到block处实现的)

```
public void doCFG() {
        refillFlowChart();
        cleanUnReachableBlock();
        queryDominates();
        queryImmDomTree();
        queryDf();
    }
```

首先对流图的前后关系进行维护:

```
public void refillFlowChart() {
        Instr instr = instrList.get(instrList.size() - 1);
        if (instr instanceof JumpInstr) {
            BasicBlock after = ((JumpInstr)
instr).getTargetBlock();
            next.add(after);
            after.prev.add(this);
        } else if (instr instanceof BranchInstr) {
            BasicBlock thenBlock = ((BranchInstr)
instr).getThenBlock();
            BasicBlock elseBlock = ((BranchInstr)
instr).getElseBlock();
            next.add(thenBlock);
            next.add(elseBlock);
            thenBlock.prev.add(this);
            elseBlock.prev.add(this);
        }
    }
```

同时，我们需要保证删除所有不能到达的块，防止对我们的分析造成影响,具体实现是从入口代码进行深搜，删除没有被访问到的块

```
public void cleanUnReachableBlock() {
        HashSet<BasicBlock> unReach = new HashSet<>
(basicBlocks);
        dfsDomTreeCheckReach(basicBlocks.get(0), unReach);
        basicBlocks.removeIf(unReach::contains);
        // 别忘了删除他们的使用前后节点关系，不然始终藕断丝连!!!
        for (BasicBlock block : basicBlocks) {
            block.next.removeAll(unReach);
            block.prev.removeAll(unReach);
        }
    }
```

```
11  private void dfsDomTreeCheckReach(BasicBlock entry,
    HashSet<BasicBlock> unReachable) {
12          unReachable.remove(entry);
13          for (BasicBlock block : entry.next) {
14              if (unReachable.contains(block)) {
15                  dfsDomTreeCheckReach(block, unReachable);
16              }
17          }
18      }
```

之后就按照按指导书给定的操作进行dom的计算

> 推荐的一个方法是迭代计算。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

注意，按照此算法得到得是谁支配我，而不是我支配谁，所以需要再反向填一下我支配谁的表，该错误给我造成过很大的困扰

```
1   public void queryDom() {
2           showPrev();
3           showNext();
4           boolean ok = false;
5           int count = 0;
6           // out[entry]=v_entry,  out == dominates
7           basicBlocks.get(0).whoDomMe.add(basicBlocks.get(0));
8           // for (Entry外的每个基本块) out[B] = T ,这里 T = N
9           for (int i = 1; i < basicBlocks.size(); i++) {
10              basicBlocks.get(i).whoDomMe.addAll(basicBlocks);
11          }
12          // while 某个out值发生变化
13          while (!ok) {
14              ok = true;
15              System.out.println("pass" + ++count);
16              for (int i = 0; i < basicBlocks.size(); i++) {
17                  if (i == 0) {
18
     basicBlocks.get(i).printContainBlocks(basicBlocks.get(i).whoD
    omMe);
19                  } // 除了entry外的每个基本块B
20                  else if
    (!basicBlocks.get(i).queryWhoDomMe(basicBlocks)) {
21                      ok = false;// 这里如果用&&莫名其妙会优化?
22                  }
23              }
```

```
        }
        for (BasicBlock block : basicBlocks) {
            block.refillMeDomWho();
        }
        for (BasicBlock block : basicBlocks) {
            block.printMeDomWho();
        }
    }
```

接下来就是支配建树，按照其定义翻译即可

> - **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

```
public void queryImmDomer() {
        // 单独开一个，方便遍历节点
        HashSet<BasicBlock> set = new HashSet<>(meDomWho);
        set.remove(this);
        for (BasicBlock blockA : meDomWho) {
            // 如果是严格支配A
            if (blockA != this) {
                // 当前没有找到B
                boolean notContainBDomA = true;
                for (BasicBlock blockB : set) {
                    if (blockB.strictDom(blockA)) {
                        notContainBDomA = false;
                        break;
                    }
                }
                if (notContainBDomA) {
                    blockA.immDomer = this;
                    this.beImmDom.add(blockA);
                }
            }
        }
    }
```

之后我们需要按照算法求出支配边界

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)
```

照着写即可，没有坑点

```java
public void queryDf() {
    // a - > b
    // a is this
    for (BasicBlock b : next) {
        BasicBlock x = this;
        while (x != null && !x.strictDom(b)) {
            x.df.add(b);
            x = x.immDomer;
        }
    }
}
```

此时我就完成了所有支配相关的流图分析，可以进行mem2reg了

mem2reg

在初步的llvm ir中，所有局部变量都变成了`alloca/load/store`形式

但是这样load store的开销回非常大，我们要把这个时刻从mem中访存的操作，换成用虚拟寄存器的操作

对于全局变量和数组，我们不做这个优化。我们的 mem2reg 只需要对局部的 `int` 类型的变量进行处理

教程给出了以下的代码，我将按照它进行实现

Algorithm 3.1: Standard algorithm for inserting ϕ-functions

```
1  for v: variable names in original program do
2      F ← {}                          ▷ set of basic blocks where φ is added
3      W ← {}                          ▷ set of basic blocks that contain definitions of v
4      for d ∈ Defs(v) do
5          let B be the basic block containing d
6          W ← W ∪ {B}
7      while W ≠ {} do
8          remove a basic block X from W
9          for Y: basic block ∈ DF(X) do
10             if Y ∉ F then
11                 add v ← φ(...) at entry of Y
12                 F ← F ∪ {Y}
13                 if Y ∉ Defs(v) then
14                     W ← W ∪ {Y}
```

现在来理解下这个算法：

F是做了一个记录，记录哪些已经插过phi了，一个块对一个变量插一个就足够了

假定defs(v)是所有定义了变量v的块。这个和初始的W是一个意思

W是所有定义(修改该值）的块，加上后代表目前还没有考虑到这个

接下来对于所有修改该值的块，如果还有没有考虑到的，那么就开始考虑这个，同时从W中移除

对于这个块的所有支配边界，这个变量的值可能会被其他块的这个量影响，所以要插phi(这个phi的来源值是目前还没有确定，需要等到变量重命名的时候确定)，同时标记这个块已经被这个变量插过了。当然，phi指令本身就是一个对变量赋值的，所以插入后这个指令也要算作defs(v)的，就是w

注意，虽然可能一个块没有涉及到一个变量，但是经过它的后续都有可能涉及到这个变量，所以在这个块上插入phi是理所当然的。

刚才我们已经完成了df的求解，得到了每一个块的支配边界。但是注意，我们仅仅完成了块的。

和变量有关的是一点没碰啊。。。

同时，发现这里很多东西都是跨块的。如果再把一堆东西都放到块里面执行，就太复杂了，这个功能单独拿出来执行即可。

注意，重命名，什么的都是针对的变量，这个变量不是llvmValue，就是不是%那个东西，而是在源程序里的那个 int a什么的，它被且仅被alloca定义(数组在每个位置上的使用，难以分析，我们选择只分析单一变量，也就是说正常alloca数组)

而int a这样的localVar会在什么地方被声明呢？显然就是alloca那里。在什么地方定义呢？有=就算，就是store。

算法实现

首先，我们需要获取所有由alloca生成的变量

```
private void searchVariableNames(Function func) {
    for (BasicBlock block : func.basicBlocks) {
        for (Instr instr : block.instrList) {
            if (instr instanceof AllocInstr && !
(((PointerType) instr.getAns().type).objType instanceof
ArrayType)) {
                variableNames.add(instr.getAns());
                defs.put(instr.getAns(), new HashSet<>());
                // 其中，alloc必然是该变量第一次亮相，所以我只需要判断
已经加上的variableNames即可.
            } else if (instr instanceof StoreInstr) {
                for (Value var : variableNames) {
                    if (((StoreInstr) instr).hasDef(var)) {
                        defs.get(var).add(block);
                    }
                }
            }
        }
    }
}
```

然后我们需要对每个变量进行插phi

```
private void solveVarPhi(Value var) {
        HashSet<BasicBlock> F = new HashSet<>();
        // F <- {}
        LinkedList<BasicBlock> W = new LinkedList<BasicBlock>
(defs.get(var)) {
        };
        //W <- {}
        // W <- W + B (contains def of var)
        while (W.size() != 0) {
            BasicBlock X = W.removeFirst();
            // remove a block X from W
            for (BasicBlock Y : X.df) {
                if (!F.contains(Y)) {
                    F.add(Y);
                    // insert phi
                    Y.instrList.add(0, new PhiInstr(var));
                    if (!defs.get(var).contains(Y)) {
```

```
17                          W.add(Y);
18                      }
19                  }
20              }
21          }
22      }
```

功能构成

这里还要再在func里面把功能教给func吗？我觉得直接把东西专门拿出来到一个类里面实现是比较好的

alloca： 代表一个变量(不define)

load : use值，add什么的算use吗，算，但是都需要load专门拿出来，因此load是必须的。

store : define 值

变量重命名

什么叫reachingDef? 就是一个变量在这里，它当前的值。

比如store 一个值x进到一个指针，这个指针代表变量a，那么这里它的到达定义就是x了，把所有从a里store都算作load即可

激动人心的时刻来了！！

Procedure updateReachingDef(v,i) Utility function for SSA renaming

Data: v : variable from program
Data: i : instruction from program
▷ *search through chain of definitions for v until we find the closest definition that dominates i, then update v.*reachingDef *in-place with this definition*

1 $r \leftarrow v$.reachingDef
2 **while** not ($r == \perp$ or definition(r) dominates i) **do**
3 $\quad r \leftarrow r$.reachingDef
4 v.reachingDef $\leftarrow r$

在支配树上进行 DFS，DFS 的过程中，计算并更新每个变量 v 当前的定义 v.reachingDef，并创建新的变量。算法如下：

Algorithm 3.3: Renaming algorithm for second phase of SSA construction

```
▷ rename variable definitions and uses to have one definition per variable name
1  foreach v : Variable do
2      v.reachingDef ← ⊥
3  foreach BB: basic Block in depth-first search preorder traversal of the dom. tree do
4      foreach i : instruction in linear code sequence of BB do
5          foreach v : variable used by non-φ-function i do
6              updateReachingDef(v, i)
7              replace this use of v by v.reachingDef in i
8          foreach v : variable defined by i (may be a φ-function) do
9              updateReachingDef(v, i)
10             create fresh variable v′
11             replace this definition of v by v′ in i
12             v′.reachingDef ← v.reachingDef
13             v.reachingDef ← v′
14     foreach φ: φ-function in a successor of BB do
15         foreach v : variable used by φ do
16             updateReachingDef(v, φ)
17             replace this use of v by v.reachingDef in φ
```

注意，更换value的时候，要记得把其中的user关系更换。

这里我查看了hyggge的博客，并没有采取翻译伪代码的方法，具体执行逻辑翻译如下：

对所有alloca变量开一个栈，用hashMap维护。

然后按照支配树遍历所有块

如果遇到一个alloca，则已经开好了栈了，删除该指令

```
if (instr instanceof AllocInstr) {
                // 数组不搞rename
                if (instr.type instanceof ArrayType) {
                    continue;
                }
                reachDefs.put(instr.getAns(), new Stack<>());
                instr.isDeleted=true;
                iterator.remove();
            }
```

如果遇到了store，则将store进去的值压入栈顶，代表该值目前覆盖了其支配节点的该变量的到达定义,之后删除该变量。

```
else if (instr instanceof StoreInstr) {
                if (!reachDefs.containsKey(((StoreInstr)
instr).getDstPointer())) {
                    continue;
                }
                reachDefs.get(((StoreInstr)
instr).getDstPointer()).push(((StoreInstr)
instr).getStoreInValue());
                iterator.remove();
                instr.isDeleted=true;
            }
```

如果遇到load，代表取用该指针的值，当然现在有其到达定义所决定，所以读取栈内容，获取其值

```
else if (instr instanceof LoadInstr) {
    if (!reachDefs.containsKey(((LoadInstr)
instr).getFromPointer())) {
        continue;
    }
    // 定义了倒是，但是一定有值吗？
    // 不一定,那怎么办?这个时候是未定义的?
    // 未定义的统统赋值为0，如果因为这个错了的话那就见鬼了.
    if (reachDefs.get(((LoadInstr)
instr).getFromPointer()).empty()) {
        instr.getAns().userReplaceMeWith(new Constant());
    } else {
        Value newValue = reachDefs.get(((LoadInstr)
instr).getFromPointer()).peek();
        instr.getAns().userReplaceMeWith(newValue);
    }
    instr.isDeleted=true;
    iterator.remove();
}
```

如果遇到phi指令，直接将其绑定的store压栈即可。

bug

phi缺元素问题

```
PHINode should have one entry for each predecessor of its
parent basic block!
  %V45 = phi i32 [ %V44, %b3 ]
```

比如上面这个

问题在于phi插是插了，但是填写的不够全.

```
for (BasicBlock next : block.next) {
            for (Instr instr : next.instrList) {
                if (instr instanceof PhiInstr) {
                    if (!reachDefs.containsKey(((PhiInstr)
instr).tieValue)) {
                        continue;
                    }
                    if (reachDefs.get(((PhiInstr)
instr).tieValue).empty()) {
                        continue;
                    }
                    Value newValue = reachDefs.get(((PhiInstr)
instr).tieValue).peek();
                    ((PhiInstr) instr).refill(newValue,
block);
                }
            }
        }
```

我们回填的逻辑是，由于回填phi是对每一个的后继节点的phi进行回填，所以每一个phi，会被其cfg上的父块唯一的访问一次，所以无论当前父块有无这个变量的定义，都需要对其填写(如果没有遇到那么就填写为0就好了，反正也不可能到达)

"提前引用"的变量

考虑到我们设置phi是按照程序流图的方向，而不是按照块的顺序，我们可能会在list前面的块用到了后面才会定义的东西。

所以我们在进行mips指令生成前，需要先对所有指令的getAns进行内存的分配。

```
public void allocSelf() {
        if (getAns() == null) {
            return;
        }
        int offset = MipsBuilder.MB.allocOnStack(4);
        MipsBuilder.MB.addVarSymbol(new MipsSymbol(getAns(),
offset));
    }
```

新增变量命名问题：

在优化时，默认在main里面。但是别忘了优化可是对所有函数负责的。

因此新插入phi的时候可能会重复命名变量

解决:记录所有函数的变量到了第几个数了。进入这个函数优化的时候把localcnt拿回来继续之前的计数即可

回填phi

注意，phi是针对每一个前驱基本块而言的，因此我们需要遍历所有该块的next，然后给他们的phi填上在本块block的当前到达定义.

```
for (BasicBlock next : block.next) {
            for (Instr instr : next.instrList) {
                if (instr instanceof PhiInstr) {
                    // 为什么会遇到前两种这样的情况呢?显然因为一开始的块携带的东西太少了，但是也被算做前驱了
                    if (!reachDefs.containsKey(((PhiInstr) instr).tieValue)) {
                        // 如果是不存在这个,代表未定义，这里暂时赋值成0,constant 默认是undefine
                        ((PhiInstr) instr).refill(new Constant(), block);
                        continue;
                    }
                    if (reachDefs.get(((PhiInstr) instr).tieValue).empty()) {
                        // 如果是empty,代表未定义，这里暂时赋值成0
                        ((PhiInstr) instr).refill(new Constant(), block);
                        continue;
                    }
                    Value newValue = reachDefs.get(((PhiInstr) instr).tieValue).peek();
                    ((PhiInstr) instr).refill(newValue, block);
                }
            }
        }
```

关于栈

考虑到是按照树进行dfs，所以需要时刻维护一个当前的到达定义栈的状态，开始的时候把他们存起来，结束一个节点的访问后再恢复回去

```
//------------------------------------------------------------
        // 保存栈状态
        HashMap<Value, Integer> stackBefore = new HashMap<>();
        for (Value value : reachDefs.keySet()) {
            stackBefore.put(value,
reachDefs.get(value).size());
        }
        ........
       //恢复栈状态
       reachDefs.keySet().retainAll(stackBefore.keySet());
        for (Value value : reachDefs.keySet()) {
            int size = stackBefore.get(value);
            while (reachDefs.get(value).size() > size) {
                reachDefs.get(value).pop();
            }
        }
```

消phi

phi，代表了，从哪里来，值该是多少。同时有性质：如果一个块有phi，那么phi指令一定是在所有指令的最开始

一开始设想的是记录一下来自哪个块，然后switch。

但是发现有更容易的：那个来自的块最后(当然，指跳转前)给这个变量附上相应的值就可以。

但是如果这个from块，可能跳到不一样的后继(事实上，最多俩，branch 或者 jump)，这些不一样的后继可能出现不一样的情况。如果两个分支统一在branch块里解决可能会出问题。因为我实际上并不需要在去往这个块里给这个变量赋值。虽然事实上我的设计可以保证如果是在branch块，它的两个分支里面是不会有phi的，但是以防万一，还是把情况都预备上为好.

因此，由于块A最后的指令只能为：jump，branch，ret三个指令，不考虑ret的情况，块A的转移无非两种情况：

第一种，如果jump到后继块B，那么直接在jump之前赋值即可，具体的赋值是搜索它后继的块(事实上可以直接通过jump找到)，找到来自它的phi指令，然后在A的后面对phi的getAns进行phcopy。它不需要增加辅助块。因此不需要设置修改目标块的操作。

第二，如果brachC或者d，那么检查自己的两个块的phi，看看是否有phi(如果有的话，根据phi的性质，必然存在一个来自A块的value，注意，如果是constant的话，需要检查一下是否是未定义，是的话直接赋值没有任何意义，于是就会选择不生成对应的move)。如果有，那么就新开一个块，更改指向关系，然后在这个新块里面插入pcopy指令。如果没有的话，当然就没有必要新开了

删除phi指令应该在什么时候执行呢？事实上，遍历函数所有块之后，我们就可以再过一遍块，把phi指令都删掉，只留下copy。

```
private void turnPhiToPC(Function function) {
        // 因为后续要添加中间block，因此不希望在原本上遍历
        ArrayList<BasicBlock> blocks = new ArrayList<>
(function.basicBlocks);
        // 插入pc
        for (BasicBlock block : blocks) {
            // 如果只有一个jump,说明可以直接插入
            if (block.lastInstr() instanceof JumpInstr) {
                JumpInstr jumpInstr = (JumpInstr)
block.lastInstr();
                BasicBlock target =
jumpInstr.getTargetBlock();
                for (Instr instr : target.instrList) {
                    if (instr instanceof PhiInstr) {
                        Value value = ((PhiInstr)
instr).getValueByBlock(block);
                        if (value instanceof Constant &&
((Constant)value).isUndefine) {
                            continue;
                        }
                        block.insertAtLast(new
PcopyInstr(instr.getAns(), value));
                    } else {
                        break;
                    }
                }
            }
            // 考虑到每个phi指令必然对于每一个该块的前驱块都有对应取值。
            // 因此只要B是A的后继节点，必然可以在B的所有phi里面找到对应
的from，获取其中的值
            // 所以我们可以直接找到最后的指令判断它的后继
```

```
25              else if (block.lastInstr() instanceof BranchInstr)
{
26                  BranchInstr branchInstr = (BranchInstr)
block.lastInstr();
27                  BasicBlock thenBlock =
branchInstr.getThenBlock();
28                  if (thenBlock.hasPhi()) {
29                      BasicBlock midForThen = new BasicBlock();
30                      function.basicBlocks.add(midForThen);
31                      block.insertBlock(thenBlock, midForThen);
32                      branchInstr.changeThen(midForThen);
33                      midForThen.addInstr(new
JumpInstr(thenBlock));
34                      for (Instr instr : thenBlock.instrList) {
35                          if (instr instanceof PhiInstr) {
36                              Value value = ((PhiInstr)
instr).getValueByBlock(block);
37                              if (value instanceof Constant &&
((Constant)value).isUndefine) {
38                                  continue;
39                              }
40
midForThen.insertAtLast(newPcopyInstr(instr.getAns(),value));
41                          } else {
42                              break;
43                          }
44                      }
45                  }
46                  BasicBlock elseBlock =
branchInstr.getElseBlock();
47                  if (elseBlock.hasPhi()) {
48                      BasicBlock midForElse = new BasicBlock();
49                      function.basicBlocks.add(midForElse);
50                      block.insertBlock(elseBlock, midForElse);
51                      // 该insert处实现了修改前后块的功能，因此不用担心
光改指令
52                      branchInstr.changeElse(midForElse);
53                      midForElse.addInstr(new
JumpInstr(elseBlock));
54                      for (Instr instr : elseBlock.instrList) {
55                          if (instr instanceof PhiInstr) {
56                              Value value = ((PhiInstr)
instr).getValueByBlock(block);
57                              if (value instanceof Constant &&
((Constant) value).isUndefine) {
```

```
                                continue;
                            }
                            midForElse.insertAtLast(new
    PcopyInstr(instr.getAns(), value));
                        } else {
                            break;
                        }
                    }
                }
            }
            // 如果最后是一个ret,那么不用管
        }
        for (BasicBlock block : function.basicBlocks) {
            block.removePhi();
        }
    }
```

pcopy - > move

pcopy是一个并行的move，就像phi是一个并行的phi

但是我们的指令是顺序执行的，因此，我们需要在顺序执行无法模拟并行执行的时候适当插入中间变量

```
                    LocalVar temp = new LocalVar(BaseType.I32,
false);
                    moves.add(new MoveInstr(temp,
now.getMoveIn()));
                    moves.add(new MoveInstr(now.getTarget(),
now.getMoveIn()));
                    for (PcopyInstr pc : edges) {
                        if (pc.getMoveIn() == now.getTarget())
{
                            pc.setMoveIn(temp);
                        }
                    }
```

初步GVN

目前只实现了GVN的简单步骤，首先是常量折叠，如果两个操作数都是常量，那么就删除这条指令，而把所有用这条指令ans的换成计算后的操作数

该方法的实现前提是llvm满足一次赋值的特性，如果后续进行了任何修改，则不应当使用该方法。

```
private void constFolding(Function function) {
        // 策略,在pc2move之前使用考虑到llvm value是单次赋值的,所以一个表达式得到的结果必然没有在之前被用过
        // 之后用到的也就是当前这个值,而不会给里面赋值什么的,所以可以直接替换
        for (BasicBlock block : function.basicBlocks) {
            Iterator<Instr> iterator = block.instrList.iterator();
            while (iterator.hasNext()) {
                Instr instr = iterator.next();
                if (instr instanceof ALUInstr && ((ALUInstr) instr).foldConst()) {
                    instr.isDeleted=true;
                    iterator.remove();
                } else if (instr instanceof IcmpInstr && ((IcmpInstr) instr).foldConst()) {
                    instr.isDeleted=true;
                    iterator.remove();
                } else if (instr instanceof ZextInstr && ((ZextInstr) instr).foldSelf()) {
                    instr.isDeleted=true;
                    iterator.remove();
                }
            }
        }
    }
```

然后是跳转压缩，如果br的判断量是常量，那么将必然只有一方为可达到的，把br换成jump即可.这么做可以很好的减少到不了的块，为后面寄存器分配节约空间

```
public void constBranchTpJump() {
        Instr i = lastInstr();
        if (i instanceof BranchInstr && ((BranchInstr) i).condConst()) {
            BranchInstr bi = (BranchInstr) i;
            instrList.set(instrList.size() - 1, bi.makeEqualJump());
            // 修改 变量关系.支配关系还要改吗?
            //TODO 这里支配关系没有被修改!
            BasicBlock abandon = bi.abandonTarget();
            // 删除下一个块的前驱
            abandon.prev.remove(this);
            // 删除本块的后继
            this.next.remove(abandon);
        }
```

```
14      }
```

初步死代码删除

考虑到经过上面的GVN，很多块其实是跳不到的，或者开了一个块仅仅是为了jump一下，没有什么意思。

因此，我的代码实现了：

将只有一个jump指令的块删除，并修改对应的前后关系

将无法到达的基本块删除

寄存器分配

之前在mips生成的时候，为了方便，我一直用的是全部把数据存到内存中。

这样的效率是十分低下的。

所以，我们需要在适当的时候，使用寄存器分配。

任务：

首先我们明确一个方针，一个%local变量必然对应一段内存,这个是在llvm分配阶段就已经分配好的了。不过，我们可以为了优化，在平时的时候选择把它的值暂存到寄存器里。当需要用到这个值的时候，我们首先查询它是否被寄存器存储，如果是，则可以直接调用。否则，需要把它从地址拿进来。

同时，应当在mips生成前把寄存器分配好了

首先可以肯定的是，我们需要进行活跃变量分析。

对每一个块，不断地做

$$IN[B] = useBU(OUT[B] - defB)$$

$$OUT[B] = Us$$

直到in部分不产生任何变化。

```
1  while(某个IN的值发生了改变)
2          while (inChanged) {
3              inChanged = false;
4              for (int i = blocks.size() - 1; i >= 0; i--) {
5                  BasicBlock b = blocks.get(i);
6                  //OUT[B]=Us是B的一个后继IN[s]
7                  //这个是等于,而不是add,但是事实上只会加.不过不差这点性
   能
```

```
                b.out.clear();
                for (BasicBlock s : b.next) {
                    b.out.addAll(s.in);
                }
                // IN[B] = useB U (OUT[B] - defB)
                // 可以变换操作先取out,再减def,最后加上user
                HashSet<Value> newIn = new HashSet<>(b.out);
                newIn.removeAll(b.def);
                newIn.addAll(b.use);
                // 该IN值发生了改变
                if (newIn.size() != b.in.size() ||
!newIn.containsAll(b.in)) {
                    b.in.clear();
                    b.in.addAll(newIn);
                    inChanged = true;
                }

            }
        }
```

需要确定的是，活跃变量分析是一个一脉相承的过程，in中有，代表这个变量在A之前生效，out没有，代表出了A，至少在A的后继节点里，必然不存在了。因此可以暂时的在遍历它后继的东西的时候把这些变量删掉。

同时可以拓宽的思考，A的out是针对几个in的，但是B的in这个东西是针对一个prev的，所以B的in里的东西比

A的out里的还少不少

思考，在转化代码的时候，我们的设计是在mips生成阶段,一开始就扫描一遍，为每个人alloc一个地址。这个地址应当是恒久不变的，而且平时也用不到。只有在寄存器溢出的时候才需要。那么实际上的寄存器呢?可以在varmanager里预设定一个值，那个就是该value将要被装进的reg,目前不和varsymbol耦合，只有当前值在寄存器的时候才存到varsymbol里面。

全局变量是不分配寄存器的,我们统一选择使用临时寄存器，从地址中取值.

处理spill

效率问题：如果我是选择当前分配完寄存器的话，后续的都不使用寄存器，都直接从内存拿，拿到临时寄存器里，和每次发现没寄存器了，就把一个换回去，把现在的拿出来用，哪个效率更高?

前者的话，其操作是：

```
load
```

坏处是如果大量使用这个寄存器的话那么反而得不偿失，每次都拿一下，需要赋值的时候再放回去

后者的话，其操作其实是:

```
store // 把原本寄存器的存回去
load // 把这个值拿出来
```

好处是load出来的寄存器这段时间可以一直用

最后笔者选择了前者

处理zext

注意到，在mips里面其实是没有i1的。

所以在执行的时候我们需要按照i32存这个。

我们不需要特殊对待zext,这个指令就相当于move一样.

call遗留问题

之前不论是不是void，都开了一个新的量，是愚笨的，撤销即可

进入call之前，我们需要把寄存器的值都存到内存里。

好消息是，目前由于对每一个变量分配了其地址，我们可以不移动栈指针，而是直接将寄存器的值存回对应的地方

```
public void regBackToMem() {
        for (Value value : memMap.keySet()) {
            if (memMap.get(value).register != null) {
                int offset = memMap.get(value).offset;
                new MemAsm(MemAsm.SW,
memMap.get(value).register, Register.SP, offset);
            }
        }
    }

    public void memBackToReg() {
        for (Value value : memMap.keySet()) {
            if (memMap.get(value).register != null) {
                int offset = memMap.get(value).offset;
```

```
                new MemAsm(MemAsm.LW,
memMap.get(value).register, Register.SP, offset);
            }
        }
    }
```

函数问题

之前把a1-a3一视同仁,但是实际上考虑到传参的时候

这个函数的a1,a2,a3

传到后面的a2,a3,a1

就乱套了

所以不能在平时用a1a2a3存。

而函数参数存储对应关系

p1,p2...pn

mem1...memn

前三个额外有a1,a2,a3这三个寄存器用来传递

实际上,这mem1....memn都是在新函数栈底依次排列的

在call的时候

||旧sp　|新sp

　　ra|m1,m2,m3

把m4以后的参数直接存到他们的内存里面

m1,2,3直接放到a1,a2,a3里。

然后在进入到新的函数之后

首先i<=3必然会被分配寄存器

如果给参数pi分配寄存器了，如果i<=3，那么move * ai

否则mem * mi

如果没有分配寄存器,那么已经存到对应地址里面了,就不用管了

寄存器分配策略

一开始，我是采取dfs遍历前驱后继图的方式来进行寄存器的分配

理论：考虑到活跃变量分析可以得到的in out，就是当前正在活跃的变量，不活跃的直接释放掉它的寄存器也没关系，因为他们的值不会用到。而且如果在一个块里，in中没有value A，但是out中有，那么A一定会在该块里进行赋值。

因此，我一开始想的是,可以根据前后驱遍历基本块，然后对基本块内的变量进行分配寄存器

同时，为了防止冲突，需要在一开始将已经分配好的寄存器(in和out中的)提前放到已经占用的寄存器中，防止冲突。

但是，在最后，我发现，这个设计无法避免一种情况,就是in和out如果已经分配好寄存器，但是在这个块彼此冲突。虽然在测评上无法体现，但是确实有这个问题存在.

所以，最后，我更换了自己的寄存器分配策略，参考了hyggge的博客中提到的方法。

就是因为支配，是前后驱的上位替代，(支配的节点一定是其后继或者后继子孙节点,繁殖不成立)前后驱可能有循环，但是支配不会有循环一说。

因此，采用这样的手段是合适的。

但是，和他的操作不同，我将这个阶段置于消phi之后实现，因此不用考虑phi这个特殊指令的影响

具体的做法是

- 对块的支配树进行dfs
- 在dfs一个块的时候，首先求出所有value(para,local,而不能是constant)的最后一次使用
- 然后再遍历所有指令,如果当时是这个value的最后一次使用,并且它不会out(也就是将影响传播出去)那么就将这个寄存器弃用。如果这条指令所定义的量还没有准备好寄存器,那么就立即给他分配一个(当然,如果冲突的话，我会选择不分配)
- 准备进入dfs阶段，将当前的寄存器分配状态存好，然后进入子程序的遍历
- 恢复寄存器分配状态
- 删除该基本块定义的变量的占用寄存器,因为要离开这个基本块了

同时，考虑到指令pcopy的特殊性，我也没有在该阶段为他进行分配寄存器，

在最后，考虑到pcopy其实也需要分配寄存器,因此我们将所有不曾用过的寄存器，保守的一个一个唯一分给这些pcopy需要用的寄存器即可。