代码优化

一、中端优化

一般而言，还没开始优化前，LLVM IR 是通过内存读取实现的 SSA。为了更好地进行优化，需要将其从内存形式转换为含有 phi 函数的 SSA，这一过程称为 Mem2reg，之后，基于这种形式的 SSA 展开各种优化。

如果临时变量命名为 `%数字`，优化完成后应对重新编号，保证编号符合 LLVM IR 规范。

1. 死代码删除

在进行 Mem2reg 之前，先实现死代码删除。这一步会经常性用到。死代码删除主要删除以下内容：

- 未使用的函数
- 未使用的全局变量
- 不可达的指令、基本块以及空的基本块

如果中间代码的组织形式中，对每条指令维护了一个 useList，即维护了其使用情况，则可以**迭代进行**：

- 删除 useList 为空的全局变量、字符串字面量
- 删除 useList 为空的非 main 函数
- 遍历每个函数中的基本块，删除 useList 为空的基本块
- 遍历基本块中的指令，如果指令 useList 为空，且不影响程序语义，则将其删去
    - 删去不影响程序语义指令为：基本的运算指令（加减乘除模）、Phi 指令、Load 指令、Alloca 指令等。

进行上述操作时，应当维护好 useList，删除函数、基本块时，里面每条指令的操作对象的 useList 应当移除 user 为被删去的指令的 use。

在进行 Mem2reg 之前，必须先进行一次死代码删除，**主要是为了删除不可达的基本块，这是为了确保 CFG 构建不出错。**

2. Mem2reg

Mem2reg 即将 LLVM IR 从内存形式 SSA 转换为含有 phi 函数的 SSA。LLVM 工具链提供了这种转换的接口，可以用如下指令进行转换：

```
opt -mem2reg -S source.ll -o target.ll
```

可以参照其结果实现 Mem2reg。

> 命令 clang -S -emit-llvm test.c -o test.ll -O0 生成的文件不能直接使用以上命令转换，需删去一些内容。

Mem2reg 主要有两个步骤：插入 phi 指令以及变量重命名。

(1) 插入phi指令

插入 phi 指令需要使用控制流程图（CFG）。

Ⅰ 构建CFG

CFG 的结点为基本块，如果一个基本块 A 的可能跳转到基本块 B，则有一条由 A 指向 B 的有向边。

构建CFG较为简单，只需遍历基本块，将其加入图中，如果有跳转到另一基本块的 br 指令，则两个基本块间添加一条有向边。

之后需要计算：

- **支配**：如果CFG中从起始节点到基本块y的所有路径都经过了基本块x，我们说**x支配y。**
- **严格支配**：显然每个基本块都支配它自己。如果x支配y，且x不等于y，那么**x严格支配y。**
- **直接支配者**：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。
- **支配边界**：节点 n 的支配边界是 CFG 中刚好不被 n 支配到的节点集合，即 $DF(n) = \{x | n支配x的前驱结点，但n不严格支配x\}$

计算支配关系通过迭代的方式进行：

- **从函数的入口块开始**，不断迭代：令基本块的支配结点 = 某基本块**所有前驱的支配结点的交集**加上自己本身，直到所有基本块的支配结点都不再变化。

计算严格支配关系：

- 每个基本块的支配结点去除它自身就获得了对应的严格支配结点。

计算直接支配者：

- 根据定义计算，遍历基本块的严格支配结点计算即可。

计算支配边界：

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)
```

Ⅱ 插入phi指令

算法思路如下：

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}
```

具体结合 LLVM IR，实现思路为（用到的集合名称同上述伪代码）：

- 遍历函数中的 Alloca，这就是所有用到的变量（忽略数组的 Alloca）
  - 寻找变量定义所在的基本块，就是所有目标地址为对应 Alloca 的 Store 指令所在的基本块，将这些基本块插入集合 W。
  - 若 W 非空，循环执行：
    - 从 W 中取出一个基本块 X
    - 对于 X 的支配边界中的每一个基本块 Y：
      - 如果 F 中不包含 Y：
        - 在 Y 的首部添加一个参数为空 phi 指令，参数后面再填。需要记录 phi 指令对应的 Alloca
        - 将 Y 加入 F
        - 如果 Y 不在 Defs(v)（即未取出任何元素的 W）
          - 将 Y 加入 W
- 回填所有 phi 指令的参数，对于每条 phi 指令P，设其对应的变量为 V，其参数为：
  - 所在基本块的前驱基本块，以及到达该前驱基本块中的变量 V 的定义（Store 或 Phi）。可能存在没有定义的情况，我定义了一个类 Undef 来表示。

必须预先插入phi指令，再填充其参数，这是因为插入的 phi 指令也可能作为其他 phi 指令的参数。

(2) 变量重命名

之后还需要对变量进行重命名。在程序中插入 phi 函数使得每一个变量的存活区间被切割成了几片，变量重命名要做的就是给每个单独的存活区间一个新的变量名。

结合 LLVM IR，实现思路为：

- 遍历每条指令 ins：
  - 如果 ins 为非数组的 Alloca，直接删去
  - 如果 ins 为 Load，且不是读取数组元素，则将所有用到该 Load 结果的指令的对应参数替换为到达 ins 处的变量的定义（Store 或 Phi）
- 重新遍历每条指令 ins：
  - 如果 ins 为 Store，将所有指令对其的使用替换为要原本存回内存的参数。

实现 Mem2reg 后，可以提交到 llvm 赛道测试正确性。

3. 函数内联

(1) 判断能否内联

我选择将所有不会进行递归的函数进行内联。由于 SysY 中只有函数定义，没有函数声明，所以只会出现直接递归。所以满足以下条件的函数可以内联：

- 自己不调用自己

(2) 函数内联

对于可以内联的函数（设其为 F），对于所有调用该函数的指令（设其为 C，所在函数为 G），进行以下操作：

- 将该指令所在的基本块从该指令出拆分为两个基本块，设其分别为 B1 和 B2。
- 对于所有 phi 指令，将其参数中的 B1 改为 B2
- 将 F 中的所有基本块及内部指令复制一份，将复制出来的基本块都加入 G 中。复制时，指令的操作数也要替换为复制后的操作数
- 将复制来的基本块中的 alloca 指令移动到 G 的第一个基本块的首部
- 将复制来的基本块中的指令，如果原操作数为函数的参数，则应将其替换为函数调用指令所传递的参数
- 对于复制来的基本块中的 ret 指令，均需替换为 br B2，此外还需要进行以下操作：
    - ret void 无需额外操作
    - 只有一条 ret，则将对 C 的使用换为对 ret 返回值的使用
    - 有多条指令时，需在 B2 首部添加一条 phi 指令，操作数为 ret 所在基本块及返回值
- 最后将原本的函数调用语句删去，并在 B1 结尾添加一个 br 指令，跳转到复制来的 F 的入口基本块

4. 常量折叠

常量折叠就是在编译时把常量表达式的结果计算出来。

最基本的操作是：在遍历指令的过程中，判别指令的所有操作数是否为常数值，并用计算结果替代该表达式。

此外，对于只有一个操作数是常数，且满足交换律的指令，将操作数放在右边。

还可以利用一些代数性质进行优化：

- $a + 0 = a$
- $a - 0 = a$
- $a * 0 = 0$
- $a * 1 = a$
- $a / 1 = a$
- $a \% 1 = 0$
- $a \% (-1) = -a$

在实现时，思路为：对于可以算出值的表达式，使用其值替换所有用到该指令的指令的操作数，之后删除该指令即可。依赖于 useList，必须确保正确。

5. GVN和GCM

全局值编号（GVN）是指为函数内每条指令（的结果）进行编号，实现全局的消除公共表达式。落实到实现时，编号等同于对每条指令进行 hash，将根据哈希值取出相同的指令。

GVN 会带来问题，如：

```
int main() {
    int a, b = 0, c = 0;
    a = getint();
    if (a) {
        b = a * 2;
    }
    c = a * 2;
    printf("b=%d, c=%d\n", b, c);
    return 0;
}
```

经过 Mem2reg 后获取的 LLVM IR 为：（只贴出主函数内容）

```
define dso_local i32 @main() {
    %1 = call i32 @getint()
    %2 = icmp ne i32 %1, 0
    br i1 %2, label %3, label %5
3:
    %4 = mul i32 %1, 2
    br label %5
5:
    %6 = phi i32 [ 0, %0 ], [ %4, %3 ]
    %7 = mul i32 %1, 2
    call void @putstr(i8* getelementptr ([3 x i8], [3 x i8]* @.str, i32 0,
i32 0))
    call void @putint(i32 %6)
    call void @putstr(i8* getelementptr ([5 x i8], [5 x i8]* @.str.1, i32 0,
i32 0))
    call void @putint(i32 %7)
    call void @putstr(i8* getelementptr ([2 x i8], [2 x i8]* @.str.2, i32 0,
i32 0))
    ret i32 0
}
```

进行 GVN 会导致 `%4 = mul i32 %1, 2` 和 `%7 = mul i32 %1, 2` 合并：

```
define dso_local i32 @main() {
    %1 = call i32 @getint()
    %2 = icmp ne i32 %1, 0
    br i1 %2, label %3, label %5
3:
    %4 = mul i32 %1, 2
    br label %5
5:
    %6 = phi i32 [ 0, %0 ], [ %4, %3 ]
    call void @putstr(i8* getelementptr ([3 x i8], [3 x i8]* @.str, i32 0,
i32 0))
    call void @putint(i32 %6)
```

```
    call void @putstr(i8* getelementptr ([5 x i8], [5 x i8]* @.str.1, i32 0, i32 0))
    call void @putint(i32 %4)
    call void @putstr(i8* getelementptr ([2 x i8], [2 x i8]* @.str.2, i32 0, i32 0))
    ret i32 0
}
```

可以看到 %4 不是必然能执行的，所以出现了错误。

为了纠正这一错误，需要使用全局代码移动（GCM），根据Value之间的依赖关系，**将代码的位置重新安排**，可以纠正 GVN 导致的错误，还可以实现类似循环不变量外提的效果。

> GVN 导致的错误是由于合并的指令，除了进行 GCM 外，还可以进行保守的值合并，比如：
>
> - 仅在基本块内部进行值编号，合并基本块内部的表达式，这种方法称为局部值编号（LVN）
> - 还可以根据支配树进行值编号，如果基本块 B 直接支配基本块 C，即 C 在支配树中是 B 的子结点，那么使用 B 中指令替换 C 中指令不会出现问题。

经过 GCM 后，结果将为：

```
define dso_local i32 @main() {
0:
    %1 = call i32 @getint()
    %2 = mul i32 %1, 2
    %3 = icmp ne i32 %1, 0
    br i1 %3, label %4, label %5
4:
    br label %5
5:
    %6 = phi i32 [ 0, %0 ], [ %2, %4 ]
    call void @putstr(i8* getelementptr ([3 x i8], [3 x i8]* @.str, i32 0, i32 0))
    call void @putint(i32 %6)
    call void @putstr(i8* getelementptr ([5 x i8], [5 x i8]* @.str.1, i32 0, i32 0))
    call void @putint(i32 %2)
    call void @putstr(i8* getelementptr ([2 x i8], [2 x i8]* @.str.2, i32 0, i32 0))
    ret i32 0
}
```

可以看到 `mul i32 %1, 2` 被外提。

(1) GVN

GVN 的核心是哈希函数的设计。设计如下：

- 对于数值字面量，其哈希值就是该数字
- 对于指令 ins，其哈希值的计算方法参考了 java 中 List 的哈希函数，具体如下：
  - 如果指令不可合并（如函数调用、phi等），直接使用 Objects.hash 方法计算
  - 如果指令符合交换律，则其哈希值为每个参数的哈希值之和（递归计算即可）
  - 如果不符合交换律，则计算哈希值时，为每个参数的哈希值赋予不同的权重，第一个参数权重为 1，第二个为 31，第三个为 31^2……，最终哈希值乘上权重求和。

哈希值相同并不代表可以合并，还需要定义一个判断指令参数是否相等的方法。

- 指令类型不同肯定不等。
- 对于不符合交换律的指令，判断对应位置参数是否相等即可
- 如果指令符合交换律，通过两重循环判断：
    - 对于指令 A 的每个参数 arg：
        - 如果 B 中没有相同的参数，指令肯定不相等
        - 否则，取出 B 中一个相同的参数
    - 如果 B 仍有参数未匹配，指令肯定不相等
    - 否则，两个指令相等

之后，只需遍历合并指令即可。

(2) GCM

GCM 的大致步骤为：

- 构建支配树
- 利用循环分析计算循环深度
- 调度指令

Ⅰ 构建支配树

构建 CFG 时，已经求出了直接支配者，而基本块的直接支配者就是其支配树的父结点。

Ⅱ 计算循环深度

这里采用了一个不具有普适性的方法，仅对于 SysY 生成的 LLVM IR 可以保证正确。

- 初始化所有基本块的循环深度为 0
- 对支配树进行 DFS，维护一个 visiting 数组，进入递归进入到结点时，将其加入，结束时移除。如果 visiting 中包含当前所在基本块的后继块，则把这两个基本块之间的基本块的循环深度加 1。

Ⅲ 调度指令

调度指令分为几个流程：find_Pinned_Insts，schedule_Early，schedule_Late，select_block。

find_Pinned_Insts 不必实际执行，只需可以判断指令能否被调度即可。不能被调度的指令有跳转（br、ret）、函数调用（call）、phi、load、alloca。

schedule_Early 是尽可能的把指令前移，确定每个指令能被调度到的最早的基本块，同时不影响指令间的依赖关系。当把指令向前提时，限制它前移的是它的操作数，即每条指令最早要在它的所有操作数定义后的位置。伪代码如下：

```
forall instructions i do
  if i is pinned then          // Pinned instructions remain
    i.visit := True;           // ... in their original block
    forall inputs x to i do    // Schedule inputs to pinned
      Schedule_Early( x );     // ... instructions

// Find earliest legal block for instruction i
Schedule_Early( instruction i ) {
  if i.visit = True then       // Instruction is already
    return;                    // ... scheduled early?
  i.visit := True;             // Instruction is being visited now
  i.block := root;             // Start with shallowest dominator
  forall inputs x to i do      //
    Schedule_Early( x );       // Schedule all inputs early
    if i.block.dom_depth < x.block.dom_depth then
      i.block := x.block;      // Choose deepest dominator input
}
```

schedule_Late 尽可能的把指令后移，确定每个指令能被调度到的最晚的基本块。每个指令也会被使用它们的指令限制，限制其不能无限向后移。伪代码如下：

```
forall instructions i do
  if i is pinned then          // Pinned instructions remain
    i.visit := True;           // ... in their original block
    forall uses y of i do      // Schedule pinned insts outputs
      Schedule_Late( y );

// Find latest legal block for instruction i
Schedule_Late( instruction i ) {
  if i.visit = True then       // Instruction is already
    return;                    // ... scheduled late?
  i.visit := True;             // Instruction is being visited now
  Block lca := NULL;           // Start the LCA empty
  forall uses y of i do {      // Schedule all uses first
    Schedule_Late( y );        // Schedule all inputs late
    Block use := y.block;      // Use occurs in y's block
    if y is a PHI then {       // ... except for PHI instructions
      // Reverse dependence edge from i to y
      Pick j so that the jth input of y is i
      // Use matching block from CFG
      use := y.block.CFG_pred[j];
    }
    // Find the least common ancestor
    lca := Find_LCA( lca, use );
  }
  ...use the latest and earliest blocks to pick final position
}
```

```
// Least Common Ancestor
Block Find_LCA( Block a, Block b ) {
  if( a = NULL ) return b;              // Trivial case
  // While a is deeper than b go up the dom  tree
  while( a.dom_depth < b.dom_depth )
    a := a.immediate_dominator;
  // While b is deeper than a go up the dom  tree
  while( b.dom_depth < a.dom_depth )
    b := b.immediate_dominator;
  while( a ≠ b ) {                      // While not equal
    a := a.immediate_dominator;         // ...go up the dom tree
    b := b.immediate_dominator;         // ...go up the dom tree
  }
  return a;                             // Return the LCA
}
```

select_block 就是确定指令的最终位置。在确定每个指令可以被灵活调度的空间后，找到循环深度尽可能浅且尽可能的靠前的位置作为最终位置。伪代码如下：

```
... Found the LCA, the latest legal position for this inst.
... We already have the earliest legal block.
  Block best := lca;          // Best place for i starts at the lca
  while lca ≠ i.block do      // While not at earliest block do...
    if lca.loop_nest < best.loop_nest then
      // Save deepest block at shallowest nest
      best := lca;
    lca := lca.immediate_dominator; // Go up the dom tree
  }
  i.block := best;            // Set desired block
```

需要注意的是：select_block 必须在 schedule_Late 中被调用。

二、后端优化

1. 消phi

消 phi 和mem2reg基本上按照《the SSA book》伪代码实现就行。

先插入并行复制指令消去 phi：

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
```

插入的并行复制指令需要新定义一个指令类 PC 表示。

插入并行复制指令 PC 之后，应当把 phi 指令移除。但是，该 phi 指令会被其他指令用作操作数，应该将这个操作数替换为什么呢？为了解决这个问题，我选择了定义一个新指令 Empty，仅用于引用一个新的变量，用 Empty 替换 phi 指令。

再将并行复制指令串行化，转换为串行复制：

Algorithm 21.6: Parallel copy sequentialization algorithm

Data: Set P of parallel copies of the form $a \mapsto b$, $a \neq b$, one extra fresh variable n
Output: List of copies in sequential order

```
1  ready ← [] ; to_do ← [] ; pred(n) ← ⊥
2  forall (a ↦ b) ∈ P do
3      loc(b) ← ⊥ ; pred(a) ← ⊥                          ▷ initialization
4  forall (a ↦ b) ∈ P do
5      loc(a) ← a                                        ▷ needed and not copied yet
6      pred(b) ← a                                       ▷ (unique) predecessor
7      to_do.push(b)                                     ▷ copy into b to be done
8  forall (a ↦ b) ∈ P do
9      if loc(b) = ⊥ then ready.push(b)                  ▷ b is not used and can be overwritten
10
11 while to_do ≠ [] do
12     while ready ≠ [] do
13         b ← ready.pop()                               ▷ pick a free location
14         a ← pred(b) ; c ← loc(a)                      ▷ available in c
15         emit_copy(c ↦ b)                              ▷ generate the copy
16         loc(a) ← b                                    ▷ now, available in b
17         if a = c and pred(a) ≠ ⊥ then ready.push(a)   ▷ just copied, can be overwritten
18
19     b ← to_do.pop()                                   ▷ look for remaining copy
20     if b ≠ loc(pred(b)) then
21         emit_copy(b ↦ n)                              ▷ break circuit with copy
22         loc(b) ← n                                    ▷ now, available in n
23         ready.push(b)                                 ▷ b can be overwritten
```

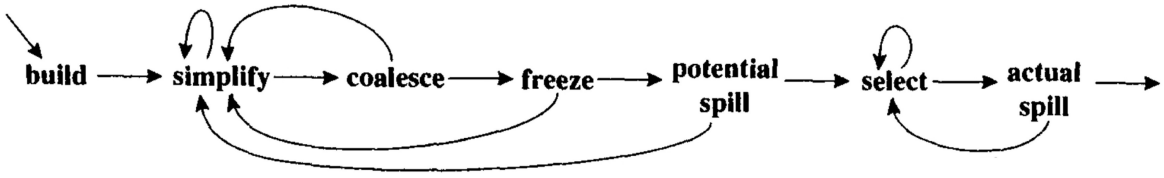
串行复制需要新定义一个指令类 Move 表示，对应于 mips 的伪指令 move $t1, $t2。消除 phi 的过程中，可能需要引入一个新的变量，这个变量的引入同样使用 Empty 实现。

2. 图着色寄存器分配

大体上基于《现代编译原理：C语言描述》中介绍的图着色寄存器分配策略进行寄存器分配，但去除了重新开始这一步。

完整的步骤包括：

- 构造（build）
- 简化（simplify）
- 合并（coalesce）
- 冻结（freeze）
- 溢出（spill）
- 选择（select）



对于 mips 的 32 个寄存器，我将 $s0 - $7 这 8 个寄存器通过图着色寄存器分配的方法分给变量；$t0 - $t9 这 10 个寄存器为临时寄存器，供图着色时没有分到寄存器的变量使用；$a0 和 $a1 供常数使用，$a2 和 $a3 在翻译指令时存储一些中间结果。

(1) 构造

本阶段的主要任务是进行活跃变量分析，从而构建冲突图。此外，为了之后的合并操作

活跃变量分析分两个阶段进行，首先进行基本块粒度的数据流分析，然后再进行指令粒度的数据流分析并构建冲突图。数据流方程为：

$$in[B] = use[B] \cup (out[B] - def[B])$$
$$out[B] = \bigcup_{s \in succ[B]} in[s]$$

基本块粒度的数据流分析的过程与理论课所讲相同，先求出每个基本块 def 和 use 集合，然后不断根据数据流方程进行迭代，最终求得每个基本块的 in 和 out 集合。

Ⅰ预处理

- **对于函数调用 call 指令**：
  - LLVM IR 中，该指令含有若干参数，实际上这些参数不需同时占用寄存器，为了避免冲突图错误地增加边，我将 call 指令拆分为了若干条自定义的压栈指令 pushstack 以及一条空参数的 call 指令。
- **对于函数的参数：**
  - memreg 后，使得函数参数的使用好像普通变量，但是其实每次使用这些参数，都需要从内存中读取（我在生成 mips 时，选择将所有参数压栈），需要补充上这些 load 指令。

Ⅱ 基本块

构建基本块 def 和 out 集合时，def 集合就是基本块内部所有定义先于使用的变量（LLVM IR 里是指令），而 out 集合是基本块内部所有使用先于定义的变量。具体实现思路为：

- 对于每个基本块，遍历其内部的指令
  - 对于指令的操作数，如果不在 def 集合中，则将其添加到该基本块的 use 集合中。需要注意一些特殊的操作数：
    - 常数、undef 不参与图着色分配寄存器，直接交替分配 $a0 和 $a1 即可。
    - 字符串常量、alloca、全局变量、函数名、基本块等标签/地址不参与寄存器分配
    - **自定义的 Move 指令**，它有两个操作数，其实是在用第二个操作数给第一个操作数赋值，因此第一个操作数应添加到 def 集合（需满足定义先于使用），第二个应添加到 use 集合（需满足使用先于定义）。
  - 对于指令，如果其参与寄存器分配（有结果且结果需要寄存器保存），并且不再 use 集合中，则将其加入 def 集合。需要注意一些特殊的指令：
    - alloca 指令不参与寄存器分配
    - **Empty 指令**仅用于引入一个新的变量，直接忽略即可，其是否加入 def 或 use 由其他指令决定。

之后，需要迭代计算每个基本块的 in 和 out 集合，为了尽快收敛，最好从函数的出口块逆序迭代（顺序不会影响结果）。

Ⅲ 指令

获取到基本块的 in 和 out 集合之后，需要进行指令粒度的数据流分析。这部分需要从基本块最后一条指令开始，逆序遍历基本块的指令，对于每条指令，引用数据流方程：

$$in[i] = use[i] \cup (out[i] - def[i])$$
$$out[i] = in[i+1]$$

最后一个指令的 out 集合就是基本块的 out 集合，其他指令的 out 集合为下条指令的 in 集合。

对于每条指令，如果其（结果）参与寄存器分配，则 $def[i] = i$，否则为空；而 $use[i]$ 则是其所有操作数。和基本块不同，指令的 $def[i]$ 和 $use[i]$ **不需要考虑**使用先于定义还是定义先于使用。此外，有以下内容需要额外考虑：

- 对于指令的操作数：
  - 字符串常量、alloca、全局变量、函数名、基本块等标签/地址，以及常数和 undef 不参与计算
  - **自定义的 Move 指令**，第一个操作数应添加到 def 集合，第二个应添加到 use 集合
- 对于指令本身：
  - 如果为 alloca 指令或 **Empty 指令**，直接跳过就行

在遍历构建每条指令的 in 和 out 集合时，可以同时建立冲突图：

- 变量定义处所有出口活跃的变量和定义的变量是互相冲突的：即**指令的 def 集合中的变量和指令 out 集合中的变量相冲突**
- 同一条指令的出口变量互相之间是冲突的：即**指令 out 集合中的变量两两冲突**（这条不考虑也不影响结果，因为每个需要分配寄存器的变量都有明确的定义点）

冲突图采用一个无自环的无向图来表示即可。

IV 传送有关结点

在消 phi 的过程中，我们引入了很多的 move 指令，而如果 move 指令的操作数所使用的寄存器相同，翻译为 mips 时，就可以删去该条指令。合并（coalesce）这一步骤，就是为了实现这个目的，让传送有关结点（move 指令的操作数）共用一个寄存器。

为此，需要先记录所有的传送有关结点。这可以通过一次遍历实现。为了记录这些结点，以及它们的相关关系，可以采用和冲突图相同的数据结构，即无自环的无向图。两个结点为传送相关结点，则这两个结点之间有一条边。

(2) 简化

这个步骤和理论课介绍的图着色算法类似：对于冲突图中的一个**非传送有关结点**，如果它的度数（相邻结点数）**小于**可用寄存器数量（这里为 8），那么将它从冲突图当中删除，并且将其压入栈中。

经过简化，会减少其他结点的度数，从而产生更多的简化的机会，并且也为之后的合并提供更多机会。

不断地进行简化，直到无可简化结点，之后，进入合并阶段。

(3) 合并

很容易想到，两个传送有关结点能合并的必要条件是：这两个结点不冲突，即冲突图中两结点间无边。但这并不足够，因为两个结点合并后引入的新结点，和其冲突的结点是合并前两个结点的冲突节点的并集。因此，一张图很有可能在合并之前是可 K 着色的，盲目合并之后就不再是可 K 着色的。为了避免这种情况，有以下两种合并策略（K 为可用寄存器数量，这里为 8）：

- Briggs：如果结点 a 和 b 合并产生的结点 ab 的高度数（即度 ≥ K）邻结点的个数少于 K，则结点 a 和 b 可以被合并。这样的合并可以保证不会将一个可 K 色着色的图变成非可 K 色着色的，因为在简化阶段将所有度小于 K 的结点从图中移走之后，被合并的结点将只能与高度数的结点相邻。因为这些结点的个数少于 K，通过简化便可以将这个合并的结点从图中移走。因此，如果原来的图是可着色的，则保守的合并方案不会改变这个图的可着色性。
- George：结点 a 和 b 可以合并的条件是：对于 a 的每一个邻居 t，或者 t 与 b 已有冲突，或者 t 是低度数（度 < K）的结点。通过下述推理可以证明这种合并是安全的。令 S 为原图中结点 a 的度小于 K 的邻结点组成的集合。若不进行合并，简化可以移去 S 中的所有结点，得到一个变小了的图 G1。如果进行合并，则简化也可以移去 S 内的所有结点，得到图 G2。但是，G2 是 G1 的子图（结点 G2 中的 ab 对应于 G1 中的 b），因此它至少会比 G1 更容易着色。

在实现时，我选用的 Briggs 条件判断能否合并。

这一阶段，就是遍历所有的传送有关结点，如果遇到可以合并的结点，则将其合并，之后返回简化阶段。两个结点合并包括：冲突图中两个结点要合并，表示传送相关结点的图中两个结点也要合并。

合并操作是穿插在简化之间的，每次合并只合并一次的效果较好。

> 我一开始进行图着色算法时，加入合并操作就会有样例点出现冲突结点共用寄存器的情况。排查很久才发现，原因在于，为了便于后面的选择阶段分配寄存器，我将冲突图备份了一份，但是结点合并时，**备份的冲突图中的两个结点没有合并**，最后导致寄存器错误分配。
>
> 使用了合并操作后，在竞速的测试点里没体现出太大区别（裂开）：

3819.0	1846.0	129984.0	56303.0	21470112.0	2096573.0	104099.0	410067.0
3819.0	1846.0	129984.0	56302.0	21470112.0	2096573.0	104099.0	410066.0

(4) 冻结

如果简化和合并都无法进行后，进入冻结阶段。

简化时，从冲突图中移除低度数的非传送有关结点。冻结，则是**从冲突图中移除一个低度数的传送有关结点，加入栈中**，放弃将其与其他结点合并。

需要注意：从冲突图中移除后，也要将其从表示传送相关结点的图中移除。

之后，重新返回简化阶段。

(5) 溢出

如果简化、合并、冻结都无法进行，进入溢出阶段。

此时冲突图中只剩下高度数的结点，在冲突图中**选择一个高度数的结点，将它标记为溢出，然后将它压入栈中**。这个时候其他结点的度数降低，可以继续进行简化。

(6) 选择

冲突图为空后，进入选择阶段。

选择阶段，就是为栈中的每个变量分配寄存器。只需要从栈中依次弹出变量，为其分配一个和冲突图中相邻结点不同的寄存器即可。对于之前标记为溢出的结点，实际上只是潜在溢出，完全有可能出现其若干个相邻结点分配了同一寄存器，使得其可以分配到寄存器。因此，判断一个结点是否可以分配寄存器的原则为：

- 其相邻结点分配的寄存器种类数小于 K（此处为 8）。

不满足上述条件的结点为实际溢出，无法分配全局寄存器，等到翻译为 mips 时动态分配即可。

(7) 关于重新开始

《现代编译原理：C语言描述》中的算法还有一步：重新开始。如果选择阶段出现了实际溢出结点，则将实际溢出结点的使用改为内存存取式的，即为这些变量在栈中分配空间，在每次使用前将其从内存中取出，每次修改后存会内存当中。这样，就将一个变量拆分为几个变量，然后重新从构造开始运行整个算法。反复迭代，直到不再有实际溢出，最终为所有变量提前分配了寄存器。

我一开始还是使用了重新开始这种方法，但是后来发现，会在翻译为 mips 时引入一些麻烦，最后放弃了。使用寄存器池为实际溢出结点动态分配寄存器也可以有一个不错的效果。

3. 循环结构优化

对于 SysY 中的 for 循环，以下述代码为例：

```
int main() {
    int i;
    for (i = 0; i  < 10; i = i + 1) {
        printf("%d\n", i);
    }
    return 0;
}
```

我之前翻译出的 LLVM IR 如下：

```
define dso_local i32 @main() {
    %1 = alloca i32
    store i32 0, i32* %1
    br label %2
2:
    %3 = load i32, i32* %1
    %4 = icmp slt i32 %3, 10
    %5 = zext i1 %4 to i32
    %6 = icmp ne i32 %5, 0
    br i1 %6, label %7, label %12
7:
    %8 = load i32, i32* %1
    call void @putint(i32 %8)
    call void @putstr(i8* getelementptr ([2 x i8], [2 x i8]* @.str, i32 0,
i32 0))
    br label %9
9:
    %10 = load i32, i32* %1
    %11 = add i32 %10, 1
    store i32 %11, i32* %1
    br label %2
12:
    ret i32 0
}
```

可以将循环结构进行调整，调整为 do-while 形式的，即循环体及更新语句执行完毕后，不再跳转回入口处的条件判断基本块，而是在更新语句之后添加循环条件判断语句。如下：

```
define dso_local i32 @main() {
    %1 = alloca i32
    store i32 0, i32* %1
    br label %2
2:
    %3 = load i32, i32* %1
    %4 = icmp slt i32 %3, 10
    %5 = zext i1 %4 to i32
    %6 = icmp ne i32 %5, 0
    br i1 %6, label %7, label %16
7:
    %8 = load i32, i32* %1
    call void @putint(i32 %8)
```

```
    call void @putstr(i8* getelementptr ([2 x i8], [2 x i8]* @.str, i32 0, i32 0))
    br label %9
9:
    %10 = load i32, i32* %1
    %11 = add i32 %10, 1
    store i32 %11, i32* %1
    %12 = load i32, i32* %1
    %13 = icmp slt i32 %12, 10
    %14 = zext i1 %13 to i32
    %15 = icmp ne i32 %14, 0
    br i1 %15, label %7, label %16
16:
    ret i32 0
}
```

这样，可以对于一个循环可以减少约一半的跳转。

4. 跳转优化

(1) 无条件跳转优化

LLVM IR 中，每个基本块都以一条跳转指令结束。而在 mips 中，指令是顺序执行的。因此，对于一条无条件跳转语句，如果跳转到的基本块是相邻的下个基本块，则这条无条件跳转语句可以删去。如：

```
1:
.........
br %5
5:
.........
```

上述示例中，"br %5" 可以删去。

更进一步的，如果无条件跳转语句跳转到的基本块是不是相邻的下个基本块，可以通过调整基本块的排列顺序，以达到去除该无条件跳转语句的目的。

(2) 条件跳转优化

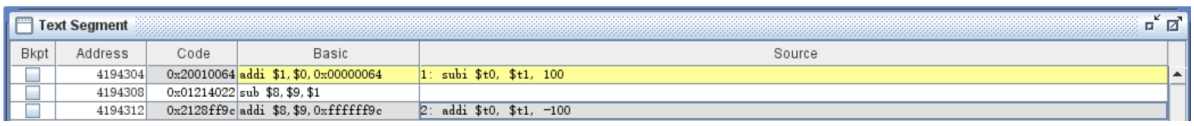
在常量折叠时，条件跳转的跳转条件可能已经计算出来了，对于这样的跳转语句可以将其更改为无条件跳转语句，之后可以继续执行上述的无条件跳转优化。

5. 指令选择

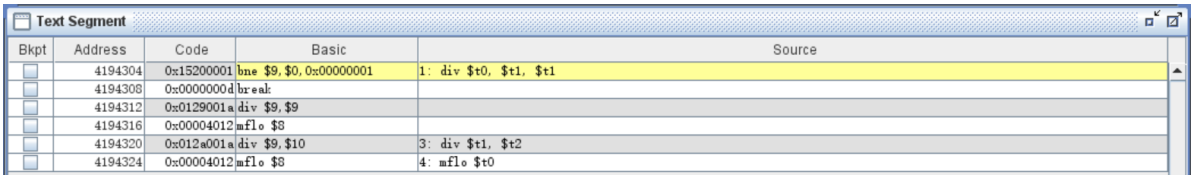
Mars 的一些伪指令会带来效率的降低，应避免使用。

发现的低效伪指令有：

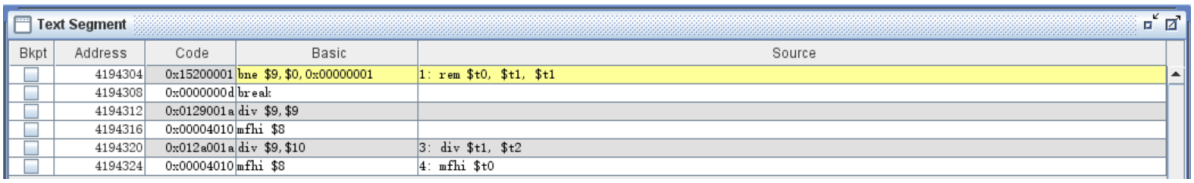
- subi $t0, $t1, 100：使用 addi $t0, $t1, -100 代替



- div $t0, $t1, $t2：会生成检查除数是否为 0 的逻辑，不要使用。



- rem $t0, $t1, $t2：同上



此外，为防止数据溢出带来结果异常，使用 addu 替代 add；addiu 替代 addi；subu 替代sub。

6. 乘除法优化

乘法和除法的代价相对较高，因此想办法将乘常数、除常数的指令替换为加、减、移位。

(1) 乘法优化

如果乘常数可以进行如下乘法优化，假设计算 $a \times b$，其中 b 是常数：

- 如果 $b = 0$，结果为 0。这个在常数折叠时应该已经优化过了。
- 如果 $b = 1$，结果为 a。在常数折叠时应该已经优化过了。
- 如果 $b = -1$，结果为 $-a$，转换为减法。
- 如果 $|b| = 2^k$，转换为 $a << k$。b 为负数，记得最后进行一次减法取相反数。
- 如果 $|b| = 2^k + t$，且 $t \leq 2$，转换为 $a << k再减少t个a$。b 为负数时，记得最后进行一次减法取相反数。（之所以要求 $t \leq 2$是因为乘法的代价为 4，而移位、加减的代价为 1）
- 如果 $|b| = 2^k - t$，且 $t \leq 2$，转换为 $a << k再减去t个a$。b 为负数时，记得最后进行一次减法取相反数。
- 其他情况，只能直接计算 $a \times b$。

(2) 除法优化

由于除法为整除，所以除常数可以进行除法优化，转化为乘法指令和移位指令：

$$quotient = \frac{dividend}{divisor} = (dividend * multiplier) >> shift$$

合理选择 mutiplier 和 shift。具体实现稍微有点绕，直接借鉴了别人的代码。

理论部分参见：Division by Invariant Integers using Multiplication