编译课程设计申优文档

求求你们这样写编译器

这个编译器是**写的很烂的**，不要学习，虽然靠着一些技巧，在编译优化方面，能跑的比较快。

主要的缺点就是没有实现抽象语法树（AST），且中间代码设计很差，与llvm ir有很大的差距

所以，参考llvm官方的基础教程，我对课程的小编译器的设计建议是：

- 分离语法分析与语义分析，也就是**一定要有AST树来表示语法结构**，方便差查错和代码翻译，并且可以提高扩展性
- 对于中间代码，可以**参考llvm ir的设计思路**，抽象出 **模块--函数--基本块--语句** 这几个层次，方便中间代码向目标代码的翻译，与优化
- 优化部分，**写成不同的优化pass函数**，方便排查错误，解耦代码

- 对于AST，设计一个公共的接口/父类，所有的各类AST继承这个类，通过统一的接口（虚函数）调用
- 静态单一赋值SSA可以考虑实现
- **前端可以参考llvm的官方教程的实现方法 http://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html 构造lexer, parser, AST, codegen**
- 后端设计不再赘述

词法分析参考：http://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html

语法分析与抽象语法树的生成：http://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl02.html

抽象语法树的语义分析与错误处理：很简单，构造好符号表

中间代码代码翻译：这里是翻译成LLVM IR，翻译成自己的IR同理 http://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html 抽象出 **模块--函数--基本块--语句** 这几个层次

~~如教程所说，这里不讨论软件的设计模式，就按照教程来写，很方便的~~

（优势：避免了由于first集重复过多而回退，比如变量定义和函数定义，要读到第三个才能确定，但是按照教程的设计思路就无需理会这个）

编译器的总体设计

语法分析

问题：文法的改写

课程设计给的文法各个语法成分存在许多$FIRST$集相交的情况，其中大部分可以通过课程中讲的方法，即通过对文法的改写，改写成若干子句，使得各个非终结符的头符号集合不相交，这样才能实现不带回溯的递归下降子程序的分析方法。

大部分的$FIRST$集相交的语法成分可以通过多读一个词，根据词法成分类型来判断接下来进入哪个分析子程序。但是对于“有返回值的函数声明”与“变量说明”这两个语法成分难以区分。

$$
\begin{aligned}
&<\text{变量说明}> ::= <\text{变量定义}>;\{<\text{变量定义}>;\}\\
&<\text{变量定义}> ::= <\text{类型标识符}>(<\text{标识符}>|<\text{标识符}>'['<\text{无符号整数}>']')\{,(<\text{标识符}>|<\text{标识符}>'['<\text{无符号整数}>']')\}\\
&<\text{有返回值函数定义}> ::= <\text{声明头部}>'('<\text{参数表}>')''{'<\text{复合语句}>'}'\\
&<\text{声明头部}> ::= int<\text{标识符}>|char<\text{标识符}>
\end{aligned}
$$

只有词法分析读到左括号时，才能判断是有返回值的函数声明；词法分析读到逗号或分号时，才能判断这是个变量说明，而由于会预读一个符号，此时再输出语法成分是后于预读的符号的，这样的结果是错误的。

这里我采取的方式是：当进入变量说明分析子程序时，**向后预读3个词**；如果是左括号，说明这应该是有返回值的函数定义；否则继续执行变量定义分析子程序。

符号表的设计与错误处理

问题：符号表的管理与设计

符号表贯穿整个编译的整个过程，无论是语义分析、错误处理、中间代码、目标代码等环节，所以符号表的设计非常重要。

根据C0文法的特性，可以抽象出符号表应该由三个类别组成：**①常量、②变量、③函数**

符号表是以上三种符号的集合，其中，函数又是变量和常量的集合。

问题：符号表如何落实到代码层面

面向对象的思维方法。

以上三种类别可以抽象出一些共同的属性，例如：

- 名字$name$（一个字符串）
- 类别$kind$（类别只能是常量$constant$、变量$variable$、函数$function$三者之一）
- 类型type（类型只能是$void$、int、$char$三者之一，表示函数返回值的类型或者变量常量的类型）

感谢C++提供的面向对象的编程方法。可以抽象出一个$Item$类，拥有上述三种共同的属性，常量类$ConstantItem$、变量类$VariableItem$、函数类$FunctionItem$分别继承$Item$类。

这样面向对象的设计方法为后续符号表的填表、查询等操作带来了很大的便利性。

问题：让符号表的填表、查询等操作更快、更简单

由于C0文法或者C语言是分程序结构语言，但都不存在函数的嵌套定义的情况，所以在总体的符号表内，设置一个**指针**$nowFunctionItem$，指向当前编译的函数对应的符号表项，查询时先查本函数的局部变量，再查全局变量或函数定义。

只需在进入某个函数时设置这个指针即可，查询、填表等操作无需关心具体的查询过程，这些都由符号表实现。

符号表的填表、查询的操作很多，栈式符号表的时间负责度比较高，代码也比较复杂。为了提高效率，这里使用C++ STL中的map，以符号项的名字作为索引key，实现快速查询和信息的获取，将查表的时间复杂度降低至$O(logn)$

问题：错误处理中，如何识别表达式的类型

根据课程组给出的定义，一个表达式是$char$类型当且仅当以下情况：

- 表达式由<标识符>或＜标识符＞'['＜表达式＞']'构成，且<标识符>的类型为char，**即char类型的常量和变量、char类型的数组元素。**
- 表达式仅由一个<字符>构成，即字符字面量。
- 表达式仅由一个有返回值的函数调用构成，且该被调用的函数返回值为$char$型

只在表达式计算中有类型转换，字符型一旦参与运算则转换成整型，包括小括号括起来的字符型，也算参与了运算，例如('c')的结果是整型。

所以语法分析时，**<因子>、<项>、<表达式>分析子程序需要逐层向上返回分析到的因子的类型**，如果因子是表达式则返回int类型。

<项>和<表达式>的类型是$char$型当且仅当：只有一个因子或项是$char$型时。也就是表达式或项中只要出现任意运算类型或括号表达式，都是int型。

语义分析与中间代码

问题：算数表达式的翻译方法

课上讲的属性翻译文法来翻译算术表达式，但是我实在没有想出来怎么把属性翻译文法用程序实现。求助万能的百度，“表达式转四元式”，给出了一种很简便的、容易理解的翻译方法。

利用栈来保存操作数和运算符，根据运算符的优先级决定入栈或者出栈运算，有点类似于数据结构课上学的中缀表达式转逆波兰式。

细化到我们的文法，<因子>将变量名、常数、表达式、函数返回值或数组的临时变量等运算数压入栈中，上一级的<项>根据运算符$*$，/号进行运算输出中间代码，并将运算结果压栈；再上一级的<表达式>再根据运算符$+$，$-$运算并压栈。

举个例子：

```
x = a * (b + c)
```

利用上述方法，翻译为

```
#temp_var_1 = b + c
#temp_var_2 = a * #temp_var_1
x = #temp_var_2
```

当分析完一个表达式以后，**栈中有且仅有一个操作数**，只需将这个操作数弹出，进行接下来的赋值、传参、比较等操作即可。

直到学完后面的**算符优先分析法**，我才明白上述方法其实就是算符优先分析法的一种简单情况，或者说咱们文法中表达式相关的部分式符合算符优先文法的。

问题：条件表达式的翻译方法

有两个方案：

1. 根据条件表达式的布尔值跳转。

   按照C语言的语义，**条件表达式是一种布尔运算**，返回值是0或1，所以可以算出条件表达式对应的**布尔值**，分支指令再根据这个布尔值进行跳转。$MARS$模拟器中正好也为我们提供了这六种关系运算符对应的指令，所以这个方案是可行的。

   条件运算符和伪指令的对应关系如下表：

关系运算符	<	<=	>	>=	==	!=
mips指令	slt/slti	sle	sgt	sge	seq	sne

   这样的好处是：语义逻辑于C语言一致，生成的中间代码与C语言直接对应，符合人的直观思维。

2. 根据语义直接在分支语句中比较并跳转。

   用到了<条件>的由$if-else$分支语句，以及$do-while$，$while$，for循环语句。其中，除了$do-while$结构式条件的作用是**满足条件则向上跳转继续循环**，**其他都是不满足向下跳转结束循环或执行else语句**。所以如果直接在分支语句中比较，就会产生两种跳转的形式，一种是条件为真时跳转（$do-while$），另一种是**条件的非为真**时跳转。（这是跳转方向和语义共同决定的）

   下面用一段C语言代码说明：

   ```
   do {
       a = a + 1;
   } while (a != 10);
   ```

   上述代码翻译成汇编应该是

   ```
   LABEL:
   addiu $t0 $t0 1
   # 如果按照方式一翻译应该是
   # sne $t1 $t0 10
   # bne $t1 0 for_end
   bne $t0 10 LABEL # 不同点，这里是a不等于10则向上跳继续循环
   ```

   但是对于for循环，就不太一样

```
for (a = 1; a != 10; ) {
    a = a + 1;
}
```

翻译成汇编应该是

```
li $t0 1
for_begin:
    # 如果按照方式一翻译应该是
    # sne $t1 $t0 10
    # beq $t1 0 for_end
    beq $t0 10 for_end # 不同点，这里是a==10，也就是条件的非为真时，向下跳转跳出循环
    addiu $t0 $t0 1
    j for_begin
for_end:
```

为了保证中间代码的可读性和与源代码的一致性，我把这部分留到了优化部分，后面会有详细的说明。

好处是：对应的指令少，效率高

下表是条件跳转与mips指令的对应关系：

条件跳转	等于跳转	不等于跳转	小于跳转	小于等于跳转	大于跳转	大于等于跳转
mips指令	beq	bne	blt	ble	bgt	bge

问题：中间代码的保存形式和输出方法

按照要求，中间代码应该是四元式形式的。为了方便目标代码翻译和输出，我实现一个中间代码的类，使用了一个**枚举类**作为中间代码的操作符OP，用三个字符串保存中间代码的操作数$first, second$和结果$result$。在这个类中实现一个$toString()$方法，根据各个操作符输出该四元式对应的字符串。

这么做的好处是方便添加、修改中间代码的操作符，同时将**中间代码的输出和保存形式分开**，一面利于编译器后端对中间代码的分析，另一面方便人对中间代码的阅读，便于DEBUG和优化。

目标代码栈的设计

问题：函数栈的作用和需要保存的东西

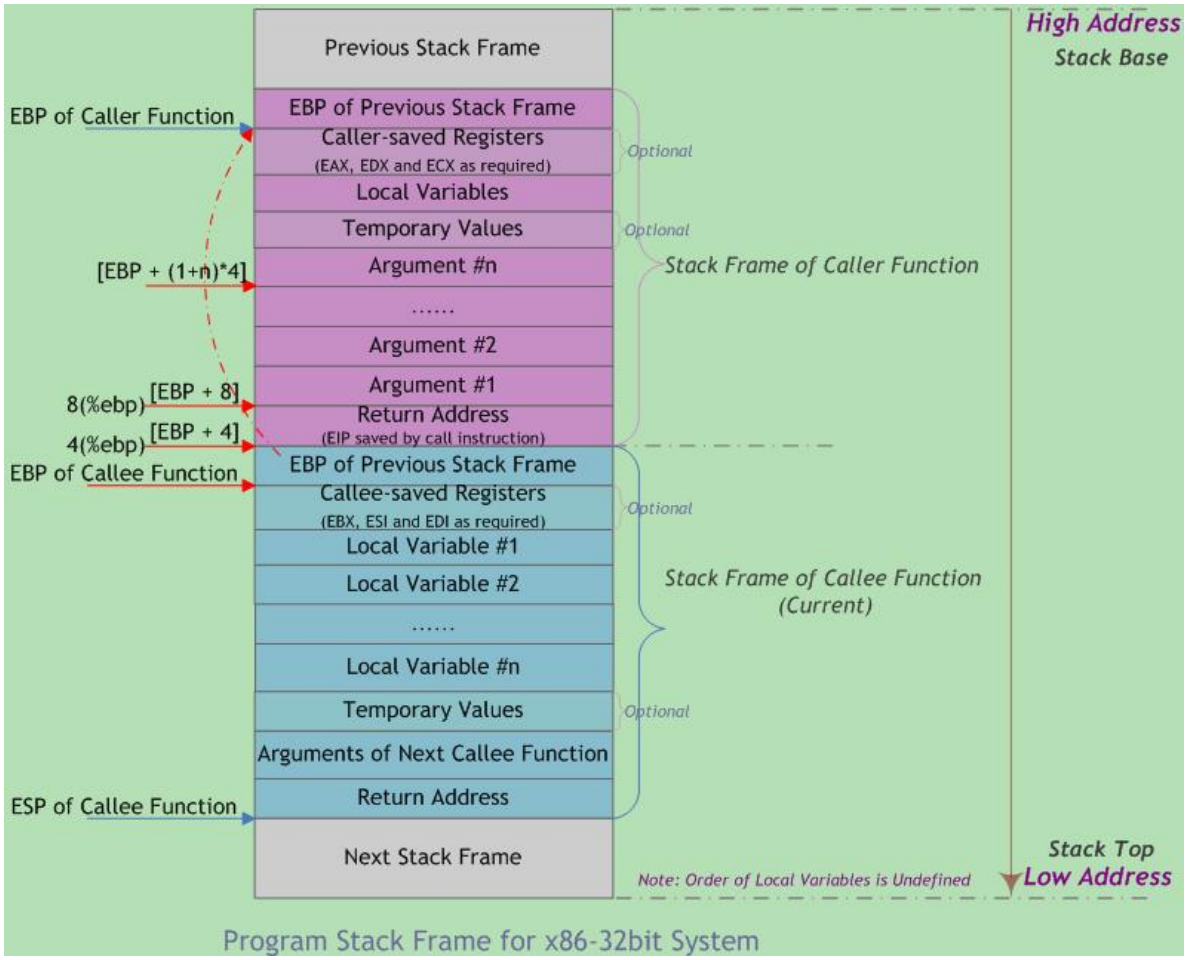
首先要考虑清楚mips的$sp, $fp和$gp这三个寄存器的作用。

每次函数调用的栈帧保存了如下的信息（函数栈里面有啥）：

- 函数的返回地址
- 函数的参数
- 函数的本地变量（包含数组）
- 中间代码生成的临时变量
- 调用者$callee$保存的寄存器
- 上一个栈帧的帧指针$\$fp$

这部分主要是参考书上和这幅图：

> https://blog.csdn.net/summonlight/article/details/81123785



编译器的优化

常数合并和传播

这个优化的宗旨就是：在编译过程中，利用编译器计算出所有可以计算的常数值。

在翻译成中间代码的阶段，所有的字符常量和整形常量，都使用常量值进行替换。如果表达式翻译阶段，栈中取出的两个操作数均是常数，则根据运算符把运算结果入栈，而不输出中间代码。

这样可以减少ALU指令，并且常量将不占用栈空间。

举个例子：

```
const char char_a = 'b';
A = 9 / 4 * 2 + (char_a - 'a') + A;
```

经过常数合并和传播，会被翻译为

```
const char char_a = 'b';
A = 5 + A;
```

死代码删除

对于条件表达式，如果条件的布尔值是可知的（表达式都是常数值），可以根据条件的真假判断是删除if分支还是$else$分支。

举个例子：

```
const int debug = 0;
int testFunctionCall() {
    if (debug != 0) {
        printf("test function call");
        return 1;
    } else
        return 1;
}
```

经过死代码删除，上述代码会被翻译为

```
int testFunctionCall() {
    return 1;
}
```

对循环结构的死代码删除同理，可以将分支语句替换为无条件跳转语句或者删除循环体。

这样做的好处是：

- 减小代码体积
- 减少分支语句

循环结构的优化

以for循环为例，直观的for循环翻译方法如下：

```
var1 = t0
start_label:
t1 = <condition>
beq t1 0 end_label
<statement>
var2 = var3 <op> <step>
goto start_label
end_label:
```

这种结构很直观，但是仔细思考，end_label前的$goto$语句显然是多余的。三种循环结构中，只有$do-while$结构只需要一条分支指令和一个标签。为什么不在goto的时候就直接判断循环能否继续？如何去掉这条多余的$goto$语句，把$for/while$循环转化成$do-while$循环的结构。

这两类循环结构的差异就是：循环体至少执行几次。$for/while$至少执行0次，而$do-while$至少执行1次。那么在外面加一个判断条件，就能使for/while循环转化为do-while循环。

如果写成C语言，这两种形式是等价的。

> while循环

```
i = 0;
while (i < 100){
    // .......
}
```

> 转为do-while循环

```
i = 0;
if (i < 100) {
    do {
        // .......
    } while (i < 100);
}
```

这样可以大大减少循环中的跳转j指令，缺点是会增加代码量，可读性变差。

寄存器的分配

函数的参数寄存器

调用函数时，将函数的前三个参数依次放入寄存器$\$a1,\ \$a2,\ \$a3$（寄存器$\$a0$保留给系统调用使用），多余的参数压入栈中保存在内存里。

局部变量分配寄存器

对局部变量的寄存器分配采用了比较简单的引用计数法，对于处在循环体内的变量会适当增大权重，增加其分配到寄存器的可能性。按照变量引用次数的降序依次分配8个s寄存器，无法分配到寄存器的变量分配对应的栈空间。对于全局变量则不分配寄存器，因为全局变量在函数调用过程中可能被修改，分配寄存器有可能导致数据不一致的问题。

临时寄存器

可以证明，临时变量的作用域不会超过一个基本块。维护一个临时寄存器池，使用中间变量的时候，如果临时寄存器池没有满，则将寄存器分配给该临时变量，在这个基本块内，这个临时寄存器只能对应的临时变量使用。同时，10个临时寄存器需要保留$\$t8,\$t9$两个临时寄存器，作为缓冲用来完成内存中的变量的计算。

临时寄存器可能存在跨基本块的情况，比如条件中，两个表达式都是函数调用的返回值，则需要特殊处理

函数内联

小函数的内联可以大大减少函数调用的开销。

函数内联的条件

为了方便实现，我对内联函数的条件比较苛刻。一个函数是可以内联，必须同时满足如下条件：

- 可以内联的函数必须是**叶子函数**（不需要保存$\$ra$）
- 函数**参数不能超过4个**（以mips参数寄存器个数为标准）
- **不能定义变量**（不需要开辟栈空间保存本地变量）
- **不能使用到全局变量或数组**（为了避免内联后**全局变量与同名局部变量的访问冲突问题**）
- （非必要条件：）**不存在分支、跳转语句**（**解决标签的冲突和函数的多出口问题即可**）

长话短说，可以内联的函数是：只包含ALU指令、没有定义变量、没有使用全局变量、参数少于4个的叶子函数。

所以，可以内联的函数的理想形式可以是这样：

```
int exp2(int x) {
    return (x * x);
}
```

其实内联函数条件的最后一条我认为是可以删除的，为了避免标签冲突，只需要对函数内的标签重新编号即可。例如常见的abs绝对值函数，去除上述条件的最后一条的限制，就可以实现内联。

函数内联的实现

以上述$exp2(int\ x)$为例，展示内联函数的实现过程。

内联前的中间代码：

调用语句

```
push -2
call exp2
x = retValue
```

exp2函数的中间代码

```
func int exp2 ()
    para int x
    #temp_var_0 = x * x
    #ret_var_v0 = #temp_var_0
    return
```

- 扫描中间代码，发现函数调用语句时，查表获取函数是否可以内联。首先判断出$exp2$函数是可以内联的。
- 获得$exp2$函数的参数表，并将实际参数与形式参数对应起来
- 删除传参和调用语句，用内联函数的语句插入替换
- 插入替换过程中，用实参代替形参，并对中间代码的临时变量重命名（重命名的作用是防止内联函数中的临时变量与调用函数的临时变量产生冲突）

内联后的中间代码

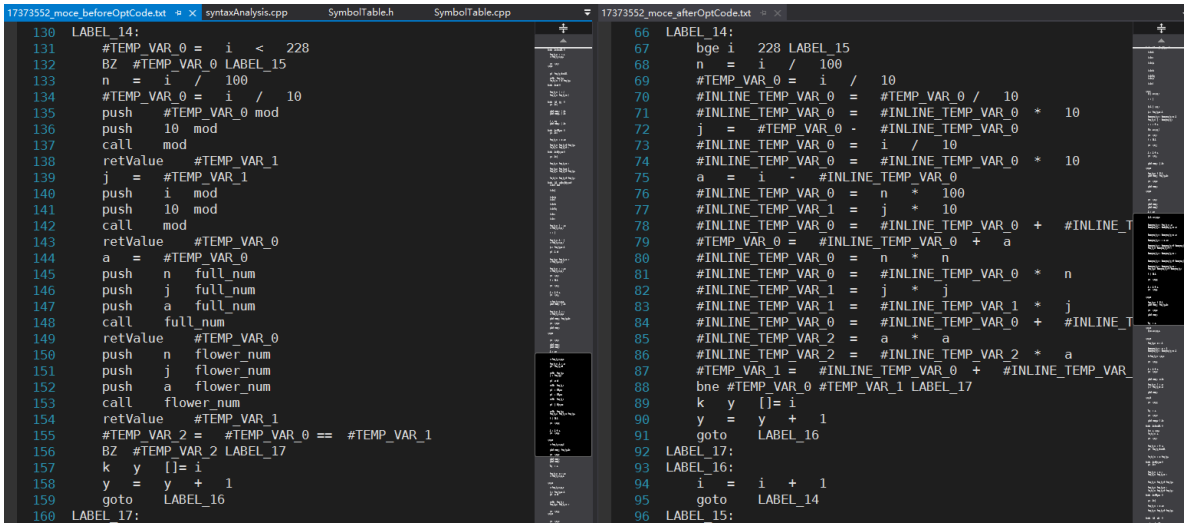
```
#inline_temp_var_0 = -2 * -2
#inline_ret_var_v0 = #inline_temp_var_0
x = #inline_ret_var_v0
```

上述代码还可以进行复制传播、常数合并，进行进一步优化。需要注意的是，如果内联函数的实际参数是常数，且函数体中出现了修改函数参数的情况，需要特殊处理。

内联前后的对比

以公布的竞速排名程序为例，其中mod、$full\_num$、$flower_num$函数是可以内联的。优化前后的中间代码如下图所示：

> 左边是内联前，右边是内联后

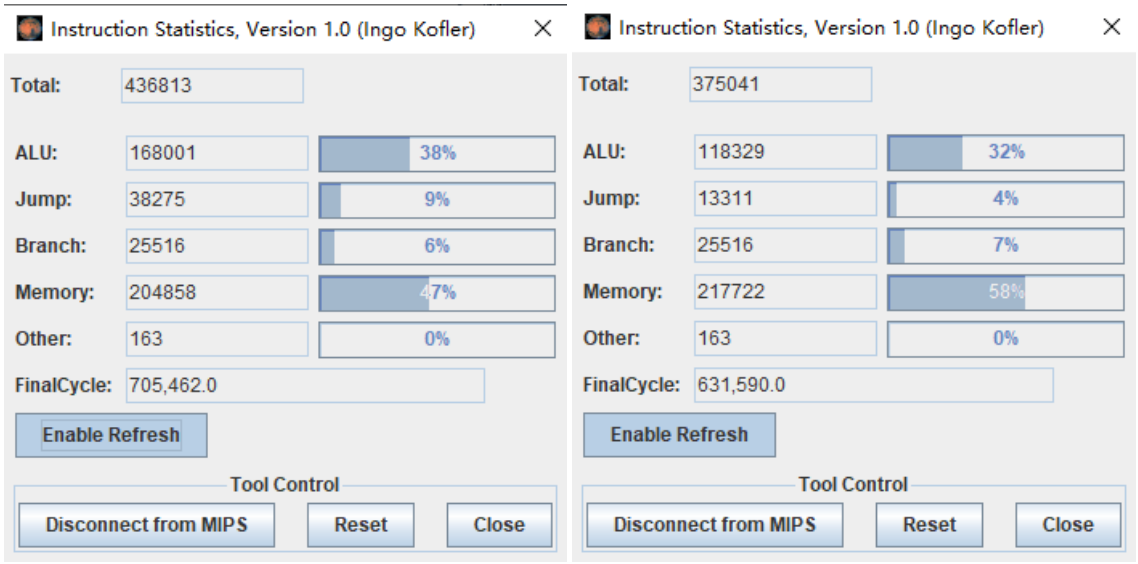


```
LABEL_14:
    #TEMP_VAR_0 =   i   <   228
    BZ  #TEMP_VAR_0 LABEL_15
    n   =   i   /   100
    #TEMP_VAR_0 =   i   /   10
    push    #TEMP_VAR_0 mod
    push    10  mod
    call    mod
    retValue    #TEMP_VAR_1
    j   =   #TEMP_VAR_1
    push    i   mod
    push    10  mod
    call    mod
    retValue    #TEMP_VAR_0
    a   =   #TEMP_VAR_0
    push    n   full_num
    push    j   full_num
    push    a   full_num
    call    full_num
    retValue    #TEMP_VAR_0
    push    n   flower_num
    push    j   flower_num
    push    a   flower_num
    call    flower_num
    retValue    #TEMP_VAR_1
    #TEMP_VAR_2 =   #TEMP_VAR_0 ==  #TEMP_VAR_1
    BZ  #TEMP_VAR_2 LABEL_17
    k   y   []= i
    y   =   y   +   1
    goto    LABEL_16
LABEL_17:
```

```
LABEL_14:
    bge i   228 LABEL_15
    n   =   i   /   100
    #TEMP_VAR_0 =   i   /   10
    #INLINE_TEMP_VAR_0  =   #TEMP_VAR_0 /   10
    #INLINE_TEMP_VAR_0  =   #INLINE_TEMP_VAR_0  *   10
    j   =   #TEMP_VAR_0 -   #INLINE_TEMP_VAR_0
    #INLINE_TEMP_VAR_0  =   i   /   10
    #INLINE_TEMP_VAR_0  =   #INLINE_TEMP_VAR_0  *   10
    a   =   i   -   #INLINE_TEMP_VAR_0
    #INLINE_TEMP_VAR_0  =   n   *   100
    #INLINE_TEMP_VAR_1  =   j   *   10
    #INLINE_TEMP_VAR_0  =   #INLINE_TEMP_VAR_0  +   #INLINE_T
    #TEMP_VAR_0 =   #INLINE_TEMP_VAR_0  +   a
    #INLINE_TEMP_VAR_0  =   n   *   n
    #INLINE_TEMP_VAR_0  =   #INLINE_TEMP_VAR_0  *   n
    #INLINE_TEMP_VAR_1  =   j   *   j
    #INLINE_TEMP_VAR_1  =   #INLINE_TEMP_VAR_1  *   j
    #INLINE_TEMP_VAR_0  =   #INLINE_TEMP_VAR_0  +   #INLINE_T
    #INLINE_TEMP_VAR_2  =   a   *   a
    #INLINE_TEMP_VAR_2  =   #INLINE_TEMP_VAR_2  *   a
    #TEMP_VAR_1 =   #INLINE_TEMP_VAR_0  +   #INLINE_TEMP_VAR_
    bne #TEMP_VAR_0 #TEMP_VAR_1 LABEL_17
    k   y   []= i
    y   =   y   +   1
    goto    LABEL_16
LABEL_17:
LABEL_16:
    i   =   i   +   1
    goto    LABEL_14
LABEL_15:
```

Instruction Statistic 可以看出，整体代价有明显下降，达到了约10%

> 左边是内联前，右边是内联后



可以看出，ALU指令减少为原来的$\frac{2}{3}$，$JUMP$指令减少为原来的$\frac{1}{3}$.原因是**函数内联减少了函数参数传递和取出返回值**的过程。

$Memory$指令略微增长的原因是：由于此时关闭了分配寄存器，**内联带来的函数代码膨胀**会增加栈空间和临时变量数目，使访存次数增多。分配寄存器以后，整体代价还可以进一步继续下降。

调用叶子函数的改进与函数保存现场

叶子函数是不调用其他函数、处在函数调用链最末端的函数。

所以对于叶子函数，其返回地址寄存器$\$ra$不会改变，叶子函数使用到的寄存器也是可以确定的。

针对叶子函数的以上特性，我做出了两点改进：

- 叶子函数的函数栈内不需要保存寄存器$\$ra$
- 调用叶子函数时，需要保存的寄存器是：调用者和被调用的叶子函数使用寄存器的交集

对于不可内联的调用频繁的叶子函数这可以降低很多$Memory$指令。

对于普通函数的调用，需要保存的寄存器只有当前用到了的寄存器，不需要保存分配的所有寄存器。

指令的选择和窥孔优化

条件语句的优化

MARS模拟模拟器提供的布尔运算伪指令翻译为基础指令时，少则1条，多则4条ALU指令。但是使用MARS提供的6条条件跳转指令（伪指令），则至多只需要2条ALU指令就能完成跳转。

按照此前对条件语句翻译的分析，只需判断中间代码分支语句$BZ,\ BNZ$和他们之前布尔运算，就能化简为MARS提供的分支指令。

举个例子，while循环，循环条件是等于则继续循环，优化前的mips指令如下：

```
seq $t1 $t0 10 # ==
beqz $t1 while_end # 不满足条件则跳出循环
```

这会被MARS翻译为

```
addi $at, $0, 10
subu $t1, $t0, $t1
ori $at, $0,  1
stlu $t1, $t1, $at
beqz $t1 while_end
```

上述一条ALU伪指令被翻译为了四条基础指令，大大增加了运算量。优化后，使用mips提供的指令就可以完成上述操作（只需要一条加载立即数的指令）：

```
bne $t0 10 while_end
```

加、减、乘、除法

- 多用**带有立即数的指令**，比如addiu。由于C语言不会处理溢出异常，所以要**选择不会产生溢出异常**的指令。
- 常数0使用0号寄存器代替。
- 根据$mips$指令的特点，没有减去立即数的原生指令，所以当减数是一个常数时，可以转化为加上这个常数的相反数，使用$addiu$指令，此时要注意这个常数必须大于$INT_MIN=0x8000_0000$，否则会溢出。
- 乘以一个立即数且这个立即数是2的幂时，可以使用左移位指令代替，这个在数组寻址中非常常见，可以节省一条加载立即数的ALU指令。
- 需要注意的时，除以一个2的幂立即数**不能**使用逻辑右移指令代替，右移就要考虑符号为了，一定要用算术右移sra。

删除多余跳转语句

多余的跳转语句指的是：无条件跳转语句的目标语句是该跳转语句的下一句，也就是目标语句减去跳转语句的PC值为4。这种情况非常常见，对于条件语句$if-else$，当else语句缺省时，就会在if语句块结束的时候产生一条多余的跳转语句，可以这条无用的跳转语句直接删除。

对模运算的优化

由于文法中没有定义模运算，但是公开的测试代码中又用到了模运算，所以简单探讨一下对模运算的优化。

对形如这样的源代码：

```
local_var_0 = x - y * (x / y);
```

对应的中间代码为：

```
/*expr1*/ #temp_var_0 = x / y;
/*expr2*/ #temp_var_1 = y * #temp_var_0
/*expr3*/ local_var_0 = x - #temp_var_1
```

可以看出这个表达式是为了求x/y的余数，也就是模运算，所以可以在中间代码中识别出这样的式子，直接替换为模运算。替换必须满足以下的充分条件：

- 这三个中间代码**不能使用或修改任何全局变量**
- 前两个表达式的结果$expr1.result,\ expr2.result$**只能是中间代码的临时变量**，不能是局部变量

这样替换才能保证结果的正确性。替换后的中间代码为：

```
/*expr3*/ local_var_0 = x % y
```

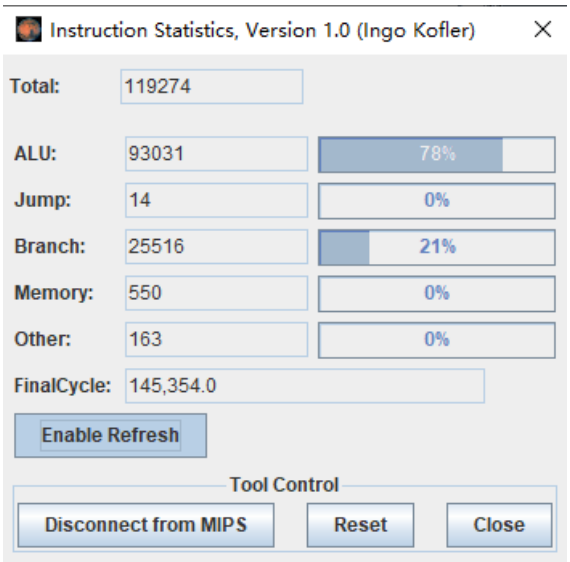
对于汇编指令，从原来的4条汇编指令减少为2条

```
#优化前
div $a0 $a1
mflo $t0
mul $t1 $a1 $t0
subu $s2 $a0 $t1
#优化后
div $a0 $a1
mfhi $s2
```

（其实我个人感觉可以把模运算加入文法，模运算的优先级与乘法除法相同，并且mips中也有相应的取余数的指令）

总结

> 测试程序最终优化效果



终于编译器课程设计算是告一段落。与另一大6系神课计算机组成原理相比，我认为编译课设的难度其实是高于计组课设的。组成原理课设更多的是按照课程组和实验指导书的要求，去实现一个没有错误的小CPU，很多的方法、方案都是固定好了的，我们需要做就是实现它，“照猫画虎”。但编译课设不一样，理论课只给我们讲了基础的东西，并没有限制我们怎么从零开始实现一个小编译器。发的参考手册也只是提供一些建议，具体的实现过程每个人都用自己不同的思考和想法；编译器的优化也有不同的角度、不同的实现方式。这给我们提供了大量独立思考、创新和试错的机会。

课程设计中碰到了很多困难，例如中间代码的表示和优化、如何处理函数内联、栈的设计以及很多代码实现层面的问题，好在一一克服后，还是取得了不错的效果，从第一版FinalCycle为一百一十万优化到最后的不到二十万。当然还是有很多做的不够充分的地方，比如寄存器分配的效果不够理想等。

编译原理课程设计是将课堂上学习的知识转化为实践的最好的机会，一方面理论课的学习为课程设计提供了原理上的支持，另一方面通过课程设计的实践又巩固了理论课的学习，加深了对理论课知识的理解。

最后祝愿编译课设的课程改革成功顺利，给后来者带来更好的学习体验。也希望我自己能在理论课期末考试中取得一个好的成绩。