基于C0文法的编译器V4.0

10011603 2016302731 冯宇

**一.主要功能**

将中间代码（四元式）转化为MIPS代码

**二.代码实现**

主要添加target.cpp文件将四元式转化成MIPS代码

//index of var in static \*4

inline int getGlobalOffset\_old(string varName)

//-(8+local var.addr\*4)

inline int getLocalOffset\_old(string varName)

inline void insertCode\_old(string code)

//name[0] is letter or '#' true，判断是否是变量名（#代表是临时变量）

inline bool isIdentifier\_old(string name)

// Transform first operand to $t1

void firstOperandToRegister\_old(string operand1)

// Transform second operand to $t2

void secondOperandToRegister\_old(string operand2)

// Transform first two operands to registers

void operandsToRegister\_old(infixNotation notation)

// Transform third operand to $t0

void thirdOperandToRegister\_old(string operand3)

// Store result of $t0 (third operand) in memory

void storeThirdOperand\_old(string operand3, string registerName)

// Push one zero to local stack where $fp + offset points

void pushLocalStack\_old()

// Push zeros to local stack with size of count(cout\*4)

void pushLocalStack\_old(int count)

// Manage variable and store it into stack from $gp or $fp

//将static\_table中某一函数的变量var加入其中

void mipsVarDef\_old(bool isGlobal)

// Store $fp and $ra into stack

void funcHead\_old()

// Recover $fp and $ra

void funcTail\_old()

// Mange codes inside func/main definition

void funcContent\_old(){

对于函数输入的不同四元式有不同的处理，中心思想是将结果存储在$t0指令中，其余的值存储在$t1,$t2存储器中进行已写死的操作，不考虑寄存器的分配问题。

加减乘除操作，如果是变量，则从idTable 中找到对应的变量，如果是值则将值赋给$t1，$t2，将运算结果保存在$t0中，再将$t0赋给对应变量

赋值语句ASSIGN，如果是函数返回值赋值，将函数返回值存储器$v1赋值给$t0,再将$t0赋给对应变量。如果是数字赋值使用li指令。其余的变量之间的赋值采用lw指令,从对应存储位置将值付给存储器$t0

条件跳转指令均采用将比较值付给$t1，$t2，将判断值存进$t0，生成指令的方法

return语句：如果直接返回常量，使用li语句存储到$t0,是变量的话使用lw语句将对应位置的值存储在$t0中。再将$t0值存储到$v1中，使用j指令跳转到函数末尾

PUSH每push一个参数$sp-4，并存储

CALL语句$sp先减去所需栈空间，再使用jal语句跳转到对应函数

scanf和printf均调用系统中的固定方法

}

void mipsFuncDef\_old(){

定义该函数的LABEL

调用funcHead\_old()对fp,sp初始化

翻译四元式

尾部处理insertCode\_old(currentFunc\_old + "Tail:")

funcTail\_old();//还原$fp和$ra

}

//main函数入口，主要是对$sp,$fp指针初始化

void mainHead\_old()

//main函数转化为MIPS指令

void mipsMainDef\_old(){

定义main函数的LABEL

调用mainHead\_old()对fp,sp初始化

翻译四元式

尾部处理insertCode\_old("mainTail:")

}

//使用j指令跳转到输入的label

void mipsJ\_old(string label)

//翻译入口

void mipsProgram(string mipsFileName){

打开文本指针

插入”.data” insertCode\_old(".data"); 将printf语句中的字符串存储到该区域

插入语句部分insertCode\_old(".text");

先跳过infixtable中所有的全局变量声明和constant类型变量声明

mipsVarDef\_old(true); // Push all global variables from static table into global stack

mipsJ\_old(labelMain\_old);//跳到main（）函数入口

for (; infixIndex\_old < infixTable.size() && infixTable[infixIndex\_old].ioperator == "FUNC"&& infixTable[infixIndex\_old].operand3 != "main";) {

mipsFuncDef\_old();//将函数的声明语句的四元式转化为MIPS

}

mipsMainDef\_old();//将main函数中的语句转化为四元式

}

**三.测试结果**

|  |  |  |
| --- | --- | --- |
| Input code | 四元式 | MIPS |
| int mm(int x)  {  return (x+1);  }  int main()  {  int a=1,b=2;  mm(a);  switch(a){  case 1:b=b+1;  case 2:b=b-1;  default:b=0;  }  a=a\*4;  return;  } | int mm()  param int x  #t0 = x + 1  return #t0  int main()  int a  a = 1  int b  b = 2  push a  call mm  #t1 = #RET  goto LABEL1, if a != 1  #t2 = b + 1  b = #t2  jmp LABEL0  LABEL1  goto LABEL2, if a != 2  #t3 = b - 1  b = #t3  jmp LABEL0  LABEL2  b = 0  LABEL0  #t4 = a \* 4  a = #t4  return | .data  .text  j main  mmTail:  lw $ra, -4($fp)  move $sp, $fp  lw $fp, ($sp)  jr $ra  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  li $t0, 1  sw $t0, -8($fp)  li $t0, 2  sw $t0, -12($fp)  sub $sp, $sp, 8  lw $t0, -8($fp)  sw $t0, ($sp)  sub $sp, $sp, 4  jal mm  move $t0, $v1  sw $t0, -16($fp)  lw $t1, -8($fp)  li $t2, 1  bne $t1, $t2, LABEL1  lw $t0, -12($fp)  sw $t0, -20($fp)  lw $t1, -20($fp)  li $t2, 1  add $t0, $t1, $t2  sw $t0, -12($fp)  j LABEL0  LABEL1:  lw $t1, -8($fp)  li $t2, 2  bne $t1, $t2, LABEL2  lw $t0, -12($fp)  sw $t0, -20($fp)  lw $t1, -20($fp)  li $t2, 1  sub $t0, $t1, $t2  sw $t0, -12($fp)  j LABEL0  LABEL2:  li $t0, 0  sw $t0, -12($fp)  LABEL0:  lw $t0, -8($fp)  sw $t0, -28($fp)  lw $t1, -28($fp)  li $t2, 4  mul $t0, $t1, $t2  sw $t0, -8($fp)  move $v1, $t0  j mainTail  mainTail: |
| Input code | 四元式 | MIPS |
| int main(){  int ax,bx;  int i;  bx=0;  i=0;  for(ax=0;i<=2;i=i+1){  ax=bx+i;  }  ax=ax\*2;  return;  } | int main()  int ax  int bx  int i  bx = 0  i = 0  ax = 0  LABEL1  #t0 = i <= 2  goto LABEL0, if #t0 != 0  goto LABEL2, if #t0 == 0  LABEL3  i = i + 1  jmp LABEL1  LABEL0  #t1 = bx + i  ax = #t1  jmp LABEL3  LABEL2  #t2 = ax \* 2  ax = #t2  return | .data  .text  j main  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  li $t0, 0  sw $t0, -8($fp)  li $t0, 0  sw $t0, -16($fp)  li $t0, 0  sw $t0, -12($fp)  LABEL1:  lw $t1, -16($fp)  li $t2, 2  sle $t0, $t1, $t2  sw $t0, -20($fp)  lw $t1, -20($fp)  li $t2, 0  bne $t1, $t2, LABEL0  lw $t1, -20($fp)  li $t2, 0  beq $t1, $t2, LABEL2  LABEL3:  lw $t0, -16($fp)  sw $t0, -24($fp)  lw $t1, -24($fp)  li $t2, 1  add $t0, $t1, $t2  sw $t0, -16($fp)  j LABEL1  LABEL0:  lw $t1, -12($fp)  lw $t2, -16($fp)  add $t0, $t1, $t2  sw $t0, -8($fp)  j LABEL3  LABEL2:  lw $t0, -8($fp)  sw $t0, -28($fp)  lw $t1, -28($fp)  li $t2, 2  mul $t0, $t1, $t2  sw $t0, -8($fp)  move $v1, $t0  j mainTail  mainTail: |
| Input code | 四元式 | MIPS |
| int main(){  int ax =2;  int i=0;  while(i <=3){  i=i+1;  }  ax=ax+5;  return;  } | int main()  int ax  ax = 2  int i  i = 0  LABEL1  #t0 = i <= 3  goto LABEL0, if #t0 != 0  goto LABEL2, if #t0 == 0  LABEL0  #t1 = i + 1  i = #t1  jmp LABEL1  LABEL2  #t2 = ax + 5  ax = #t2  return | .data  .text  j main  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  li $t0, 2  sw $t0, -8($fp)  li $t0, 0  sw $t0, -12($fp)  LABEL1:  lw $t1, -12($fp)  li $t2, 3  sle $t0, $t1, $t2  sw $t0, -16($fp)  lw $t1, -16($fp)  li $t2, 0  bne $t1, $t2, LABEL0  lw $t1, -16($fp)  li $t2, 0  beq $t1, $t2, LABEL2  LABEL0:  lw $t0, -12($fp)  sw $t0, -20($fp)  lw $t1, -20($fp)  li $t2, 1  add $t0, $t1, $t2  sw $t0, -12($fp)  j LABEL1  LABEL2:  lw $t0, -8($fp)  sw $t0, -24($fp)  lw $t1, -24($fp)  li $t2, 5  add $t0, $t1, $t2  sw $t0, -8($fp)  move $v1, $t0  j mainTail  mainTail: |
| Input code | 四元式 | MIPS代码 |
| int main(){  int T0,T1,T2,T3,T4,T5,T6;  int A,B;  int R,r;  R=2;  r=3;  T0=5;  T1=2\*T0;  T2=R+r;  A=T1\*T2;  B=A;  T3=2\*T0;  T4=R+r;  T5=T3\*T4;  T6=R-r;  B=T5\*T6;  printf(B);  return;  } | int main()  int T0  int T1  int T2  int T3  int T4  int T5  int T6  int A  int B  int R  int r  R = 2  r = 3  T0 = 5  #t0 = 2 \* T0  T1 = #t0  #t1 = R + r  T2 = #t1  #t2 = T1 \* T2  A = #t2  B = A  #t3 = 2 \* T0  T3 = #t3  #t4 = R + r  T4 = #t4  #t5 = T3 \* T4  T5 = #t5  #t6 = R - r  T6 = #t6  #t7 = T5 \* T6  B = #t7  print B  return | .data  .text  j main  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  li $t0, 3  sw $t0, -20($fp)  li $t0, 2  sw $t0, -16($fp)  lw $t1, -16($fp)  lw $t2, -20($fp)  sub $t0, $t1, $t2  sw $t0, -36($fp)  lw $t1, -16($fp)  lw $t2, -20($fp)  add $t0, $t1, $t2  sw $t0, -28($fp)  li $t0, 5  sw $t0, -8($fp)  li $t1, 2  lw $t2, -8($fp)  mul $t0, $t1, $t2  sw $t0, -24($fp)  lw $t1, -24($fp)  lw $t2, -28($fp)  mul $t0, $t1, $t2  sw $t0, -32($fp)  lw $t1, -32($fp)  lw $t2, -36($fp)  mul $t0, $t1, $t2  sw $t0, -12($fp)  lw $a0, -12($fp)  li $v0, 1  syscall  move $v1, $t0  j mainTail  mainTail: |

**四. Debug以及C++知识点**

**1.什么是正确的MIPS代码**

通常而言，linux汇编完成的代码都是X86，如果需要转化为MIPS代码，需要安装额外的工具<https://blog.csdn.net/qq_41595874/article/details/88754760#mips_70>

MIPS的寄存器以及常用指令<https://blog.csdn.net/king523103/article/details/45967991>

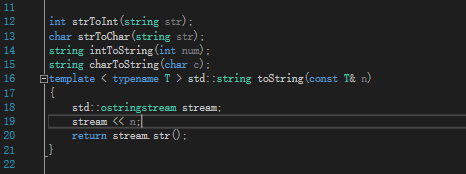
根据以上信息，对四元式进行翻译，生成可执行的MIPS代码。需要注意的是，本程序需要通过的是Mars软件的测试，所以$fp,$sp的操作可以与linux中的不一致，只需要自洽即可。保证程序运行结果的正确性。

**2.C++中的模板**

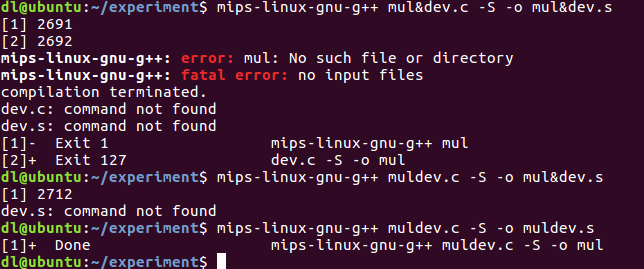
模板是C++支持**参数化**多态的工具，使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数、返回值取得任意类型**使用模板的目的就是能够让程序员编写与类型无关的代码。**比如编写了一个交换两个整型int 类型的swap函数，这个函数就只能实现**int** 型，对**double**，字符这些类型无法实现，要实现这些类型的交换就要重新编写另一个**swap**函数。使用模板的目的就是要让这程序的实现与类型无关，比如一个**swap**模板函数，即可以实现**int** 型，又可以实现double型的交换。

需要注意的是**注意：模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在main函数中声明或定义一个模板**

这种结构能大大减少重复功能函数的使用，使程序更加简洁，本程序主要是在tool.h文件中添加一个模板实现类型的转换：

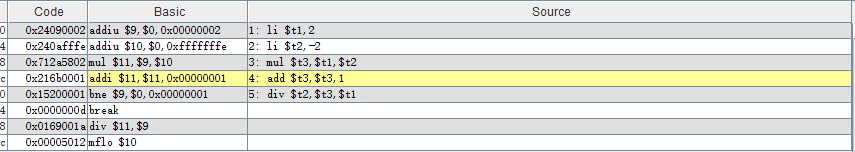


**3.linux中文件的命名不能包含&字符，否则会报错**



**4.Mars中的乘除法**

在Mars中乘除法指令形式为：



其中乘法将$t1和$t2中数字相乘的结果存储到$hi和$lo寄存器中，高位在$hi中，低位在$lo中，$t0默认从$lo中取数。而除法中Mars自动会对被除数是否为0进行判断，并进行相应处理，所以在翻译为汇编代码的时候忽略这一部分。