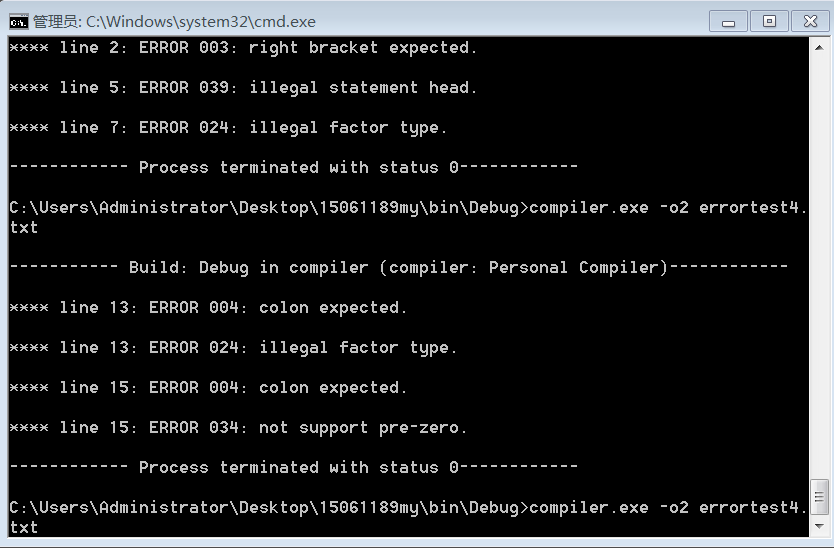
编译课设申优文档

1. 编译器工作流程



首先在cmd中打开compiler.exe，选择参数：

“-no”：不进行优化

“-o1”：只进行局部优化

“-o2”：进行局部以及全局优化

例：

compiler.exe –o2 test.txt

首先将待编译文件打开，若文件不存在，则输出

Unable to open the file，

若是不符合标准的输入，则输出

too many parameters

Usage:\t [filepath][mode](absolute or relative)

正确输入后，先输出开始符号，此时进行编译，首先调用词法分析程序预读一个单词，再进行语法分析，语法分析采用递归下降子程序法，根据成分调用相应的子程序，并在过程中进行语义分析以及生成中间代码，之后调用DAGopt进行局部优化，消除公共子表达式，再之后调用Globalopt进行全局优化，全局优化使用定义使用链，生成冲突图，并用图着色算法进行全局寄存器分配，之后通过mips.cpp生成目标代码，若没有编译错误，会输出结束符号。生成的目标代码文件存储在midcode.txt中，目标代码文件存储在tarcode.asm中。

1. 词法分析详细设计

这部分代码在wordAnalysis.cpp中，定义全局二维字符数组reservedWord存储保留字，operatorSet存储运算符，num存储当前数字token的整数值，ch存储当前单字符token的值，利用token字符数组存储分析时的单词，利用整数symbol存储单词的种类，关键函数是getsym()，首先预读文件中的下一个字符，之后清空token数组，防止之前的分析对之后产生影响，然后跳过空白字符，包括换行符，tab符，空格，直到一个非空白符出现。

之后对其进行分析：

若为字母或者下划线，那么读取多个非空白字符进而组成一个字符组，此时将文件指针后退一个字符的位置，之后因为文法要求是不区分大小写的，因此使用tolower将其变为小写，之后查找是不是保留字。若是保留字，将symbol赋值为相应的保留字宏定义值；若不是保留字，symbol赋值为IDENTIFIER这个宏定义值，表示当前字符为标识符。

若为数字，则继续读取数字，得到数字token，此时将文件指针后退一个字符的位置，将token数组转化为数字存储在num中，同时将symbol赋值为INTCONST，即整数常量。

若为单引号，则得到单字符token，将token中的字符存储到ch中，symbol赋值为CHARCONST。即字符常量，同时检查引号有没有闭合。

若为双引号，则得到字符串token，将symbol赋值为STRING，即为字符串，同时检查引号有没有闭合。

若为EOF，即文件结束符，返回-1。

若为其他符号，比如冒号，分号，逗号等，都将symbol赋值为相应的宏定义值。

至此一个单词的分析结束。

1. 语法分析详细设计

语法分析严格按照文法进行设计，采用递归下降子程序法，每一个非终结符都对应着一个函数。程序执行过程通过文法即可确定。以下是函数与非终结符的对应关系：

void addOperator();--<加法运算符>

void assignmentStatement();--<赋值语句>

void caseStatement(char tempIdName[]);--<情况语句>

void caseTable(char tempIdName[]);--<情况表>

void character();--<字符>

void charString();--<字符串>

void compoundStatement();--<复合语句>

void condition();--<条件>

void conditionalStatement();--<条件语句>

void constant();--<常量>

void constDeclr();--<常量声明>

void constDef();--<常量定义>

void declrHead();--<声明头部>

void defaultStatement();--<缺省>

void expression();--<表达式>

void factor();--<因子>

void identifier();--<标识符>

void item();--<项>

void integer();--<整数>

int letter(char s);--<字母>

void loopStatement();--<循环语句>

void multOperator();--<乘法运算符>

void mainFunc();--<主函数>

int notZeroNum(char s);--<非零数字>

int number(char s);--<数字>

void parameter();--<参数>

void paramTable();--<参数表>

void program();--<程序>

void readStatement();--<读语句>

void relaOperator();--<关系运算符>

void retFuncCall();--<有返回值函数调用>

void retFuncDefine();--<有返回值函数定义>

void retStatement();--<返回语句>

void situationStatement();--<情况语句>

void statement();--<语句>

void statementColumn();--<语句列>

void typeIdentifier();--<类型标识符>

int unsignInteger(char s[]);--<无符号整数>

void valueParamTable();--<值参数表>

void varDeclr();--<变量声明>

void varDef();--<变量定义>

void voidFuncCall();--<无返回值函数调用>

void voidFuncDefine();--<无返回值函数定义>

void writeStatement();--<写语句>

在这部分中我们还生成了中间代码，具体的四元式设计如下：

1. 函数声明

源码形如：

Int func1(int para1, int para2){}

Void func2{}

中间代码：

Func, int, 调用层数, func1

Para, int, , para1

Para, int, , para2

Func, void, , func2

1. 函数调用

源码形式：

i = func1(para1, para2)

func2;

中间代码：

vpara, int, 调用层数, para1

vpara, int, 调用层数, para2

call, func1, 调用层数, $临时寄存器

=, $临时寄存器, , i

call, func2, 调用层数,

1. 函数返回

源代码：

return (x);

中间代码：

Ret, , , $临时寄存器

1. 变量声明

源代码：

int x;

char y;

int z[10];

中间代码：

int, , , x

char, , , y

int, , 10, z

1. 常量声明

源代码：

Const int x = 10;

Const char y = ‘a’;

中间代码：

Const, int, 10, x

Const, char, 97, y

1. 表达式

源代码：

X = a\*(b+c)

中间代码：

+, b, c, $0

\*, a, $0, $1

=, $1, x,

1. 条件判断

源代码：

X == y

中间代码：

==, x, y,

1. 条件跳转

中间代码：

Jmp, , , LABEL //无条件跳转

Jne, , ,LABEL //不满足条件跳转

1. 标号语句

中间代码：

Lab, , , <name>

1. 数组取值赋值

源代码：

a[10] = 10;

v = a[10];

中间代码：

[]=, 10, a, 10

=[], a, 10, $1

=, $1, , v

1. 输入输出语句

源代码：

Scanf(a);

Printf(“a”, a);

中间代码：

Scan, , int, a

Print, “a”，a, int

1. 优化部分详细设计

局部优化：

采用的是生成DAG图消除公共子表达式，定义了以下两个结构体：

typedef struct dagnode{

char name[100];

vector<int> Lparent; //Left parent node index

vector<int> Rparent; //Right parent node index

int Lson; //Left son node index

int Rson; //Right son node index

int flag; //表示是否在结点表中

}DAGNODE;

typedef struct dagtabnode{

char name[100];

int index;

int flag; //表示是否已被导出

}DAGTABNODE;

Dagnode是DAG图节点，dagtabnode是DAG节点表节点，两个结构体的意义在注释中已经说明，关键函数是DAG(int, int)以及DAGopt()。在DAGopt中首先我们划分基本块，划分条件有isBlockHead以及isBlockEnd确定，是基本块首语句或者末尾语句的返回true，否则返回false。

划分好基本块之后，调用DAG，将基本块的头和尾传进去，首先定义DAG图以及节点表，如果中间代码的op是运算符，那么进行分析，如果操作数不在结点表中，就新建一个节点加入节点表中；之后在DAG图dag中寻找op==dag[i].name && dag[i].Lson = arg1.index && dag[i] == arg2.index的dagnode，如果找到，记录下i，否则就新建一个这样的子树，记录下新建的index，之后对midcode的res进行分析，如果在dagtab中找到，将相应的index改为之前记录的op的数值，否则在dagtab中新建一个节点，记录数值。

构建完DAG图之后要进行中间代码的导出，使用书上的启发式算法，即遍历dagtab，如果在dag中找到对应的节点且flag为0，且它的Lson和Rson对应的节点的flag均为1，说明可以导出这个节点的中间代码。定义一个vector<int> Queue存储要进行计算的临时变量的顺序，导出时逆序检索就可以。

这样局部优化就做完了。

全局优化：

使用定义使用集合进行分析，建立冲突图，使用图着色算法进行全局寄存器分配。使用数组来存储冲突图以及染色队列，冲突图节点以及染色队列节点的结构体如下：

typedef struct conflictnode{

string name;

vector<int> active; //active blocknum

set<string> link; //conflict nodes

}CONFLICTNODE;

typedef struct dyenode{

string name;

int reg; //alloced reg num

}DYENODE;

首先还是进行基本块的划分，以一个函数作为划分区间，每划分一个基本块，就求出它的use和def集，直到划分到函数结束，即midcode.op == <endop>时，使用书上求in，out集合的方法求出集合，直到in和out集合不再随着计算次数变化而变化。

之后就是构建冲突图，在冲突定义上，我采取了比较保守的策略，即每个基本块in中出现的变量的均冲突，因为我暂时无法确保其他的冲突定义可以完备的处理所有情况，因此采用了最保守的策略。这样就要分析每个in来求出冲突图。

得到冲突图之后，使用图着色算法进行染色，具体算法书上很详细，这里不多赘述。将分配好的染色队列保存至全局，以便在生成中间代码时使用。

至此全局优化完成。

1. MIPS代码生成详细设计
2. initasm()

将全局常量和变量存放到内存的.data中，并在内存中开辟空间，将其存入lvar数组中，lvar结构体数组元素有name以及offset两个属性，name是名字，offset就是相对于当前函数首地址的偏移。

1、add\_mips(), sub\_mips(), mul\_mips(), div\_mips()

将四元式中的arg1和arg2分别加载到对应的寄存器，如果是常量或者全局变量，就从内存中取值加载到$v0或$v1中，若是分配了全局寄存器，就直接使用相应的寄存器，若是未分配全局寄存器的局部变量，就从内存中读取。将计算结果存储在$v0中，如果res有全局寄存器且没有函数调用，那么就只更新全局寄存器中的值，否则就要同时更新全局寄存器以及内存中的值。不涉及优化的就将全局寄存器的部分去掉即可。

2、assign\_mips()

将四元式中的arg1加载到$v0中，将其存储在内存的相应位置，如果res有全局寄存器且没有函数调用，那么就只更新全局寄存器中的值，否则就要同时更新全局寄存器以及内存中的值。

3、func\_mips()

将$fp, 以及调用函数的$fp记录下来，提前设置好$sp。

1. para\_mips()

为一个参数在运行栈中开辟空间

1. vpara\_mips()

为值参数提前开辟空间

1. ret\_mips()

将返回值载入$a2，在调用之后进行赋值

1. end\_mips()，vend\_mips()，exit\_mips()

函数结束时的标记符

1. jne\_mips()，jmp\_mips()

生成跳转指令

1. scanf\_mips()，printf\_mips()

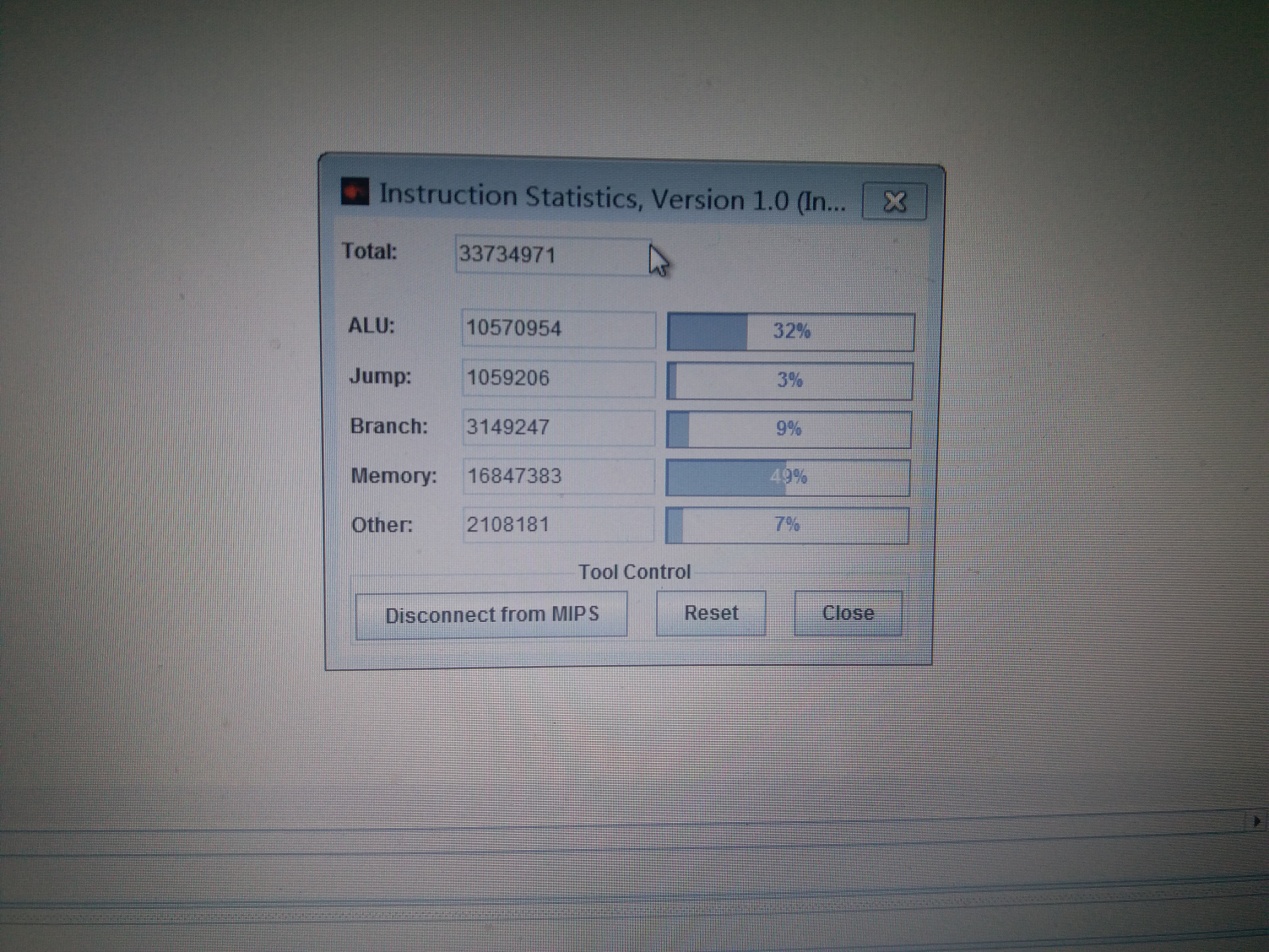
系统调用处理输入输出指令。对于printf，如果arg1或者arg2有全局寄存器且没有函数调用，那么就从全局寄存器中取值，否则就要从内存中取值。不涉及优化的部分，去掉全局寄存器即可。对于scanf，如果res有全局寄存器且没有函数调用，那么就只更新全局寄存器中的值，否则就要同时更新全局寄存器以及内存中的值。不涉及优化的就将全局寄存器的部分去掉即可。

10．Equal\_mips()，notequal\_mips()…greatorequal\_mips()

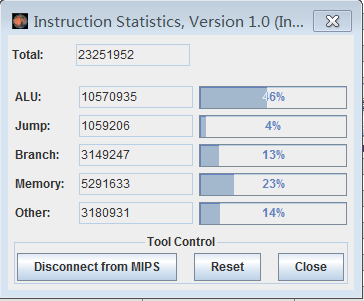
将条件两端载入寄存器进行比较，如果arg1或者arg2有全局寄存器且没有函数调用，那么就从全局寄存器中取值，否则就要从内存中取值。不涉及优化的部分，去掉全局寄存器即可。

六、性能对比

优化前：



优化后：



可以看到Memory指令减少了1000w次左右，优化效果较为显著。