一种基于 C0 文法的 MIPS32/64 指令集下的编译器及针对龙芯真机 MIPS 运行的探索

唐家祺

西北工业大学计算机学院

一. 总体设计

1.介绍

我本次实验完成了一种基于 MIPS32/64 的编译器，并使 32 位的指令集通过了仿真，64 位的指令集部分通过并运行在龙芯派。我使用的文法为 C0 文法，它是一种几乎可以实现 C 语言完备功能的文法，实现了通用的分支，选择和循环结构，具备完善的指令性质，可以清晰实现代码逻辑。

我选择的开发环境是 Microsoft visual studio 2019，这是一种功能强大的 IDE 可以方便的完成开发工作，我最终提交的代码也是一 VS 的工程形式提交，如果版本正确可以直接在 VS 上运行。

在本次开发中，我没有使用任何工具辅助我进行设计，从词法分析到最终的代码生成的全过程都是由自己编写的。在最终阶段我使用了 Mars 工具进行了仿真，同时采用了 gcc 进行了交叉编译和真机实验。

2.总述

图 1 为此编译器的流程图。

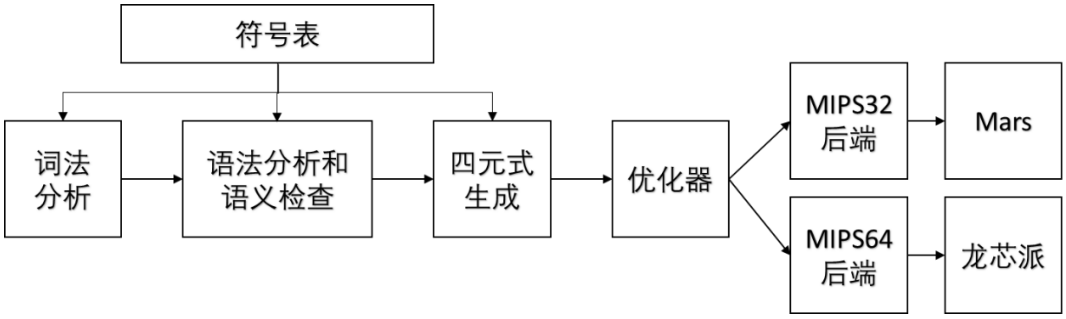


图 1 编译器流程图

我的编译器会将满足 C0 文法的代码输入到词法分析器中，最终经过一系列处理获得了 MIPS32 和 MIPS64 两种不同的汇编代码。

在前端部分，首先经过词法分析器，将单词拆分成为对应的单词及其对应的单词符号和序号；紧接着，进入语义分析和语义检查器，使用 LL(1)型文法，构造递归下降分析器，判断是否其符合语法，及其是否具有语义错误；最后，进入四元式生成器，其可以生成四元式，获得文法的中间表示。
在优化部分，主要的实现了五个基本的功能：基本块划分，数据流分析，常数传播，消除公共子表达式和窥孔优化（删除死代码）。
在后端部分，我最终完成了一个 MIPS32 的后端，它生成 32 位的 MIPS 代码，可以再 Mars 上仿真运行；同时，为了适应龙芯派的 64 位系统，我同样完成了 MIPS64 的后端代码，部分测试案例可以在龙芯派上直接运行。

3.完成时间和进度安排

时间	完成情况
2021.5.16	初步设计与词法分析
2021.5.23	语法分析
2021.5.31	语义分析和中间代码生成
2021.6.7	MIPS32 汇编语言生成
2021.6.15	优化器和 Debug
2021.6.25	MIPS64 汇编器调试
2021.6.30	龙芯派调试和运行
2021.7.2	教师检查（第一个）
2021.7.15	报告整理，代码整理，提交

注：在每一次进度结束时，我都根据当时的完成情况，提交了一份进度报告在课堂派上。

4.代码清单

资源文件	代码行数	头文件	代码行数
main.cpp	65	head.h	19
Lex.cpp	802	Lex.h	22
Syntax.cpp	2110	Syntax.h	56
Error.cpp	115	Error.h	51
IR.cpp	121	IR.h	58
Optim.cpp	463	Optim.h	16
Symbol_Table.cpp	25	Symbol_Table.h	72
Asm_Code.cpp	1451	Asm_Code.h	215
Asm_Code_MIPS64.cpp	1254	Asm_Code_MIPS64.h	54

经过统计，最终的代码情况为：

代码全部的行数	**6722 行**
不含注释的代码行数	**4128 行**

二. 测试案例

我设计了三种测试案例，分别针对不同的问题进行测试。在功能测试中，我主要测试了是否我的编译器可以正确的运行，包括多种语句的赋值和变量的赋值，输入输出，函数的递归调用等。在综合测试的部分，主要测试的是复杂情况下是否编译器可以正常运行。在性能和优化测试部分，主要测试的是我们的编译器优化是否具有效果。

最终覆盖的测试用例如下：

功能测试

Test_F1.txt	**Printf**
Test_F2.txt	**Scanf+Printf**
Test_F3.txt	**数组**
Test_F4.txt	**多变量赋值**
Test_F5.txt	**判断 if**
Test_F6.txt	**分支 switch**
Test_F7.txt	**循环 While**
Test_F8.txt	**Char 和变量名较长**
Test_F9.txt	**函数的调用，返回值**
Test_F10.txt	**斐波拉契数列**

综合测试（来自北航，CSDN，github 等用例）

Test_S1.txt	**加减乘除，printf，scanf 混合测试**
Test_S2.txt	**语句嵌套，函数，复杂过程**
Test_S3.txt	**比较过程，数组，函数调用**

优化或性能测试

Test_O1.txt	**窥孔优化，消除公共子表达式**
Test_O2.txt	**常数传播**

三. 词法分析

1.词法定义

我采用一种联合体的形式定义单词，每一个单词自动指定了一个数字作为分配的标记，我们最终的词法分析会将代码中的单词识别出来。

```
enum LEX_NUM
{
        lex_error_id,   //0//error
        lex_int_id,     //1//int
        lex_char_id,    //2//char
        lex_float_id,   //3//float
        lex_double_id,  //4//double
        lex_bool_id,    //5//bool
        lex_void_id,    //6//void
        lex_const_id,   //7//const
        lex_if_id,      //8//if
        lex_else_id,    //9//else
        lex_for_id,     //10//for
        lex_while_id,   //11//while
        lex_do_id,      //12//do
        lex_switch_id,  //13//switch
        lex_case_id,    //14//case
        lex_default_id, //15//default
        lex_printf_id,  //16//print
        lex_scanf_id,   //17//scanf
        lex_return_id,  //18//return
        lex_main_id,    //19//main
        lex_indentifer_id,//20//identifer,标识符
        lex_add_id,     //21//+
        lex_subtract_id, //22//-
        lex_multiply_id, //23//*
        lex_divide_id,  //24//,///
        lex_percent_id, //25//%
        lex_less_id,    //26//<
        lex_greater_id, //27//>
        lex_lessequal_id,//28//<,//
        lex_greaterequal_id,//29//>,//
        lex_equalequal_id,//30//,//,//
        lex_notequal_id, //31//,//!,//
        lex_equal_id,   //32//,//
        lex_comma_id,   //33//,
```

```
        lex_semicolon_id,//34//;
        lex_colon_id,   //35//:
        lex_dot_id,     //36//.
        lex_singlemark_id,//37//'
        lex_doublemark_id,//38//"
        lex_leftparenthesis_id,//39//(
        lex_rightparenthesis_id,//40//)
        lex_leftmiddle_id,//41//[
        lex_rightmiddle_id,//42//]
        lex_leftbraces_id,//43//{
        lex_rightbraces_id,//44//}
        lex_NotZeroHeadNum_id,//45//不以 0 开头数字
        lex_string_id,  //46//字符串
        lex_character_id, //47//字符
        lex_letter_id,  //48//_
        lex_ZeroToNine,  //49//0~9
        lex_Zero_id,    //50//0
        lex_OneToNine_id, //51//1~9
        lex_ZeroHeadNum_id,//52//0 开头数字
        lex_struct_id,  //53//结构体
};
```

2.代码设计

词法分析部分的代码设计如图所示：

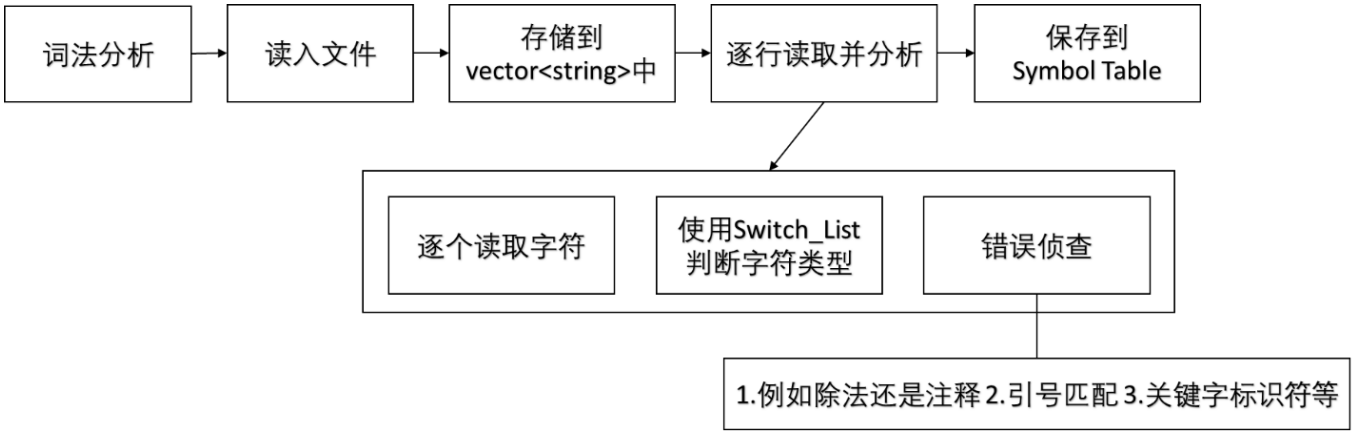


图2 词法分析设计

词法分析直观来看就是读入所需要的文件，然后将文件的全部词法信息输出出来。

void Lex_Aanlysis_main(string File)，为我设计的入口函数，其中主要实现的功能为：

1.读入文件

2.将读入的文件转化为 vector 数组，其中每一行分别对应代码中的每一有用的代码段

3.读取下一个字符串，将其存入全局变量中，这个全局变量就是存储的变量

4.当文件还未遍历完成时，对每一个单词进行词法分析 lex_Analysis

主程序即为 lex_Analysis，其主要的工作为：

1.它先读取了一个字符串，然后如果存储的字符串已经全部读完，即返回，如果没有，则对读到的字符串进行语法分析操作。

2.在这里我们构建了针对每一个字符串的局部转移表，Switch_list，因为字符串往往不是唯一确定的状态转移方式，而是多种状态，包括前一个状态和后一个状态共同决定的，我们在表中计算出所有可能的情况，即可清楚的判断哪一种状态是正确的状态。

lex_nextchar 用于得到下一个字符串

lex_recordCH 用于记录上一个字符串

lex_rollbackCH 用于回滚回上一个字符串

3.得到 Switch_list 之后，就可以用所给的值进行 Switch 语句分支选择了，一共具有 23 个分支，即可词语究竟属于哪一个分支。

4.我们使用的字符表的数据结构同样为单个字符结构体的 vector。Word 为单个字符的结构体，其中记录了 ID 号，字符名称（string），和字符所处的位置。而 word 的 vector 则将其链接为一个向量，这样的向量可以动态的增减符号表的长度。

5.最后，将所完成的词法打印出来，保存在 LEX.txt 文件中

四. 符号表

单词结构体：单个符号存储的结构。

```
typedef struct word
{
        int lex_symbol_ID;      //单词 ID
        string lex_symbol;      //单词符号
        int lex_indexOfFile;    //单词在文件中的索引
        int lex_indexOfLine;    //单词在文件中每一行的索引
};
```

符号表：

//使用 vector 管理符号表，可以方便的实现**读取，入栈和出栈**操作

```
vector<word> Lex_Table;
```

这就是我设计的符号表。符号表将标识符和其类型、位置关联起来，当我们去处理变量，函数的声明时，就是将这些信息组织（绑定）起来，放在表里，当需要知道这些函数，变量的意义时，就去这个表里查。
这样我们即可结构化的管理符号表并且方便的进行下一步的语法分析。

五. 语法分析

1.文法定义

我采用了 C0 文法，主要参考的为北航的课程设计。这个文法可以进行递归下降分析，非常容易上手。这是一种合适的文法，通过实现这种文法，即可容易的完成语法分析的操作。
文法定义经过我的改进和设计后，如下：

< 加法运算符 > ::= + | -
< 乘法运算符 > ::= * | /
< 关系运算符 > ::= < | <= | > | >= | != | ==
< 字母 > ::= _ | a | . . . | z | A | . . . | Z
< 数字 > ::= 0 | < 非零数字 >
< 非零数字 > ::= 1 | . . . | 9

< 字符 > ::= ' < 加法运算符 > ' | ' < 乘法运算符 > ' | ' < 字母 > ' | ' < 数字 > '

< 字符串 > ::= " {十进制编码为 32,33,35-126 的 ASCII 字符} "

< 程序 > ::= [< 常量说明 >] [< 变量说明 >] [<结构体说明>]
{ < 有返回值函数定义 > | < 无返回值函数定义 > }
< 主函数 >

< 常量说明 > ::= const < 常量定义 > ;{ const < 常量定义 > ;}

< 标识符 > ::= < 字母 > { < 字母 > | < 数字 > }

< 常量定义 > ::= int < 标识符 > = < 整数 > {, < 标识符 > = < 整数 > }
| float < 标识符 > = < 实数 > {, < 标识符 > = < 实数 > }
| char < 标识符 > = < 字符 > {, < 标识符 > = < 字符 > }

< 无符号整数 > ::= < 非零数字 > { < 数字 > }
< 整数 > ::= [+ | −] < 无符号整数 > | 0
< 小数部分 > ::= < 数字 > { < 数字 > } | < 空 >

< 实数 >　　::= ［+ | −］ < 整数 > [. < 小数部分 >]

< 声明头部 >　::= int < 标识符 > |float < 标识符 > |char < 标识符 >

< 变量说明 >　::= < 变量定义 > ;{ < 变量定义 > ;}

< 变量定义 >　::= < 类型标识符 > (< 标识符 > | < 标识符 > '[' < 无符号整数 > ']'){,(< 标识符 > | < 标识符 > '[' < 无符号整数 > ']')}

< 可枚举常量 >　::= < 整数 > | < 字符 >
< 类型标识符 >　　::= int | char

< 有返回值函数定义 >　::= < 声明头部 > '(' < 参数 > ')' '{' < 复合语句 > '}'
< 无返回值函数定义 >　::= void < 标识符 > '(' < 参数 > ')''{' < 复合语句 > '}'

< 语句列 >　::= { < 语句 > }
< 复合语句 >　::= ［< 常量说明 > ］［< 变量说明 > ］< 语句列 >

< 参数 >　::= < 参数表 >
< 参数表 >　::= < 类型标识符 > < 标识符 > {, < 类型标识符 > < 标识符 > }| < 空 >

< 主函数 >　::= void main'('')''{' < 复合语句 > '}'

< 表达式 >　::= ［+ | −］ < 项 > { < 加法运算符 > < 项 > }

< 项 >　::= < 因子 > { < 乘法运算符 > < 因子 > }

< 因子 >　::= < 标识符 >
　　| < 标识符 > '[' < 表达式 > ']'
　　|'(' < 表达式 > ')'
　　| < 整数 >
　　| < 实数 >
　　| < 字符 >
　　| < 有返回值函数调用语句 >
< 语句 >　::= < 条件语句 >
　　| < 循环语句 >
　　| < 情况语句 >
　　| < 读语句 > ;
　　| < 写语句 > ;
　　| < 返回语句 > ;
　　| '{' < 语句列 > '}'

| < 赋值语句 > ;
| < 有返回值函数调用语句 > ;
| < 无返回值函数调用语句 > ;
| < 空 > ;

< 赋值语句 > ::= < 标识符 > = < 表达式 >
| < 标识符 > '[' < 表达式 > ']'= < 表达式 >
< 条件语句 > ::= if '(' < 条件 > ')' < 语句 > [else < 语句 >]
< 条件 > ::= < 表达式 > < 关系运算符 > < 表达式 > | < 表达式 >
< 循环语句 > ::= while '(' < 条件 > ')' < 语句 >
< 情况语句 > ::= switch '(' < 表达式 > ')' '{' < 情况表 > < 缺省 > '}'
< 情况表 > ::= < 情况子语句 > { < 情况子语句 > }
< 情况子语句 > ::= case < 可枚举常量 > : < 语句 >
< 缺省 > ::= default : < 语句 > | < 空 >
< 有返回值函数调用语句 > ::= < 标识符 > '(' < 值参数表 > ')'
< 无返回值函数调用语句 > ::= < 标识符 > '(' < 值参数表 > ')'

< 读语句 > ::= scanf '(' < 标识符 > {, < 标识符 > }')'
< 写语句 > ::= printf '(' < 字符串 > , < 表达式 > ')' | printf '(' < 字符串 > ')'| printf '('
< 表达式 > ')'

< 返回语句 > ::= return['(' < 表达式 > ')']

< 值参数表 > ::= < 表达式 > {, < 表达式 > } | < 空 >

2.设计

语法分析的流程如下图所示：

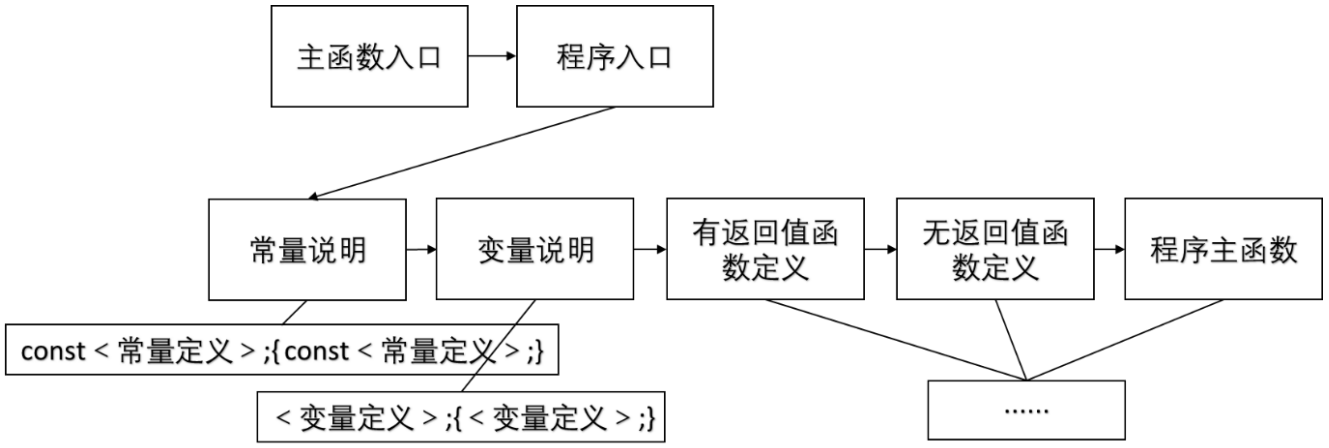


图3 语法分析

语法分析采用了递归下降分析法。这种算法在文法合适的情况下非常简单，易用。是一种主流的分析方式。
我们写的程序是符合 C0 文法的程序，所以当我们将程序输入到主函数中，此时我们已经经过词法分析，明确了我们的程序究竟该使用哪一种方式对程序进行读取，因为我们已经将单词打上标签，标签会告诉我们目前读到哪里了。

那么当我们递归下降分析时，我们就可以清楚的知道哪一种程序可以被我们的程序读到，这样递归下降下去，直到分析完整个符号表。

在 main 函数中接入入口参数，并运行:
首先进入初始化函数 **syn_init**，这个函数将我们需要的变量进行了初始化操作，包括语法分析的结果表。
syn_nextsymbol 用于进入了一个新的词语。
syn_record 用于记录当前位置。
syn_rollback 用于回退到之前的位置。
syn_read_ahead 用于提前读取下一个 symbol。

程序初始化之后将会读入第一个字符，此时将进入 syn_program 函数，也就是程序的语法分析的开始程序中。这个程序将通过递归下降的方式，分析出程序的语法结构，并在结构不为实现的结构时，产生报错信息（语义检查）。

程序结构如下:
＜程序＞ ::= ［＜常量说明＞］［＜变量说明＞］[〈结构体说明〉]｛＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞

不同的文法将被送入不同的函数中进行进一步分析，我针对 C0 的每一个文法都进行了这样的分析，以适应语法分析的要求。
例如:
＜常量说明＞ ::= const＜常量定义＞;{ const＜常量定义＞;}

六. 语义检查

1.语义检查的类型

我设计的语义检查在语法分析中，主要检查是否在语法生成过程中能否产生语义错误。当发现有错误时，就会报错，此时语义分析器会打印错误信息，语义分析随之停止。
主要解决的是以下几类错误:

- **声明与定义**

变量未声明
函数未声明
变量重定义
函数重定义
恒定常数重复定义

- **表达式**

类型不匹配
缺乏标识符

- **语句方面**

与函数返回值不匹配
企图更改常量的值

2.设计

语义检查的流程如下图所示：

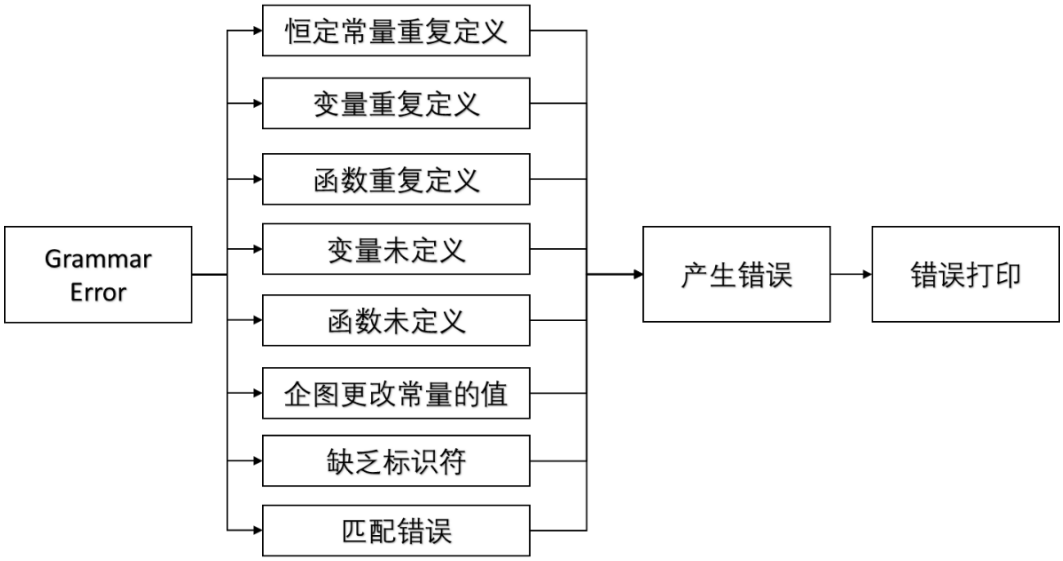


图4 语义检查

通过语义分析检查错误，如果观察到程序发生了错误。则发生报错，并且将错误打印在屏幕上。这时程序观察到不正确，即可完成语义检查。
当发生语义错误时，我们在屏幕上会观察到错误发生为代码行，位置和错误类型。

七. 四元式设计和中间代码生成

1.四元式设计

我设计的四元式结构，采用四个单词指示代码设计：

Op	Scr1	Sc2	Res	
IsRepeat	Which_Block	isentrence	isexit	islableorfunc

前四个是四元式的基本结构，后面是为了优化设计提供支撑的结构。

四元式的定义如下：

```
typedef struct Quadruple
{
        string op;
        string src1;
        string src2;
        string res;

        int IsRepeat=0;      //指示是否重复
        int Which_Block;   //指示基本块
        bool isentrance;    //入口语句
        bool isexit;          //出口语句
        bool islableorfunc;//是否是标签或者函数
};
```

四元式的前部分用于指示四个操作符号，后面用于优化器设计。

- Op 为功能操作符，这个主要指明这个四元式的功能是什么
- Src1, Src2 指出了操作的操作数
- Res 指出了标识符

2.中间代码的表示

中间表示 IR，是一个四元式的结构体数组，定义如下：

Quadruple **IR_List**[MAX_NUM];

中间代码在语句通过语法和语义分析检查后自动生成。当我们获得了一个合适的语句之后（即我们的语法分析完成），即可生成中间代码。

最终我们生成的代码，例如：

< < , 10 , a , Function_ > 对 IF 语句处理

< printf, "BIGGER", , > 对 Printf 语句的处理

< jmp , , , Function_1 > 跳入函数
< lab , , , Function_ > 设置 label
< ret , , , k > 返回值
< exit , , , F1 > 退出 label

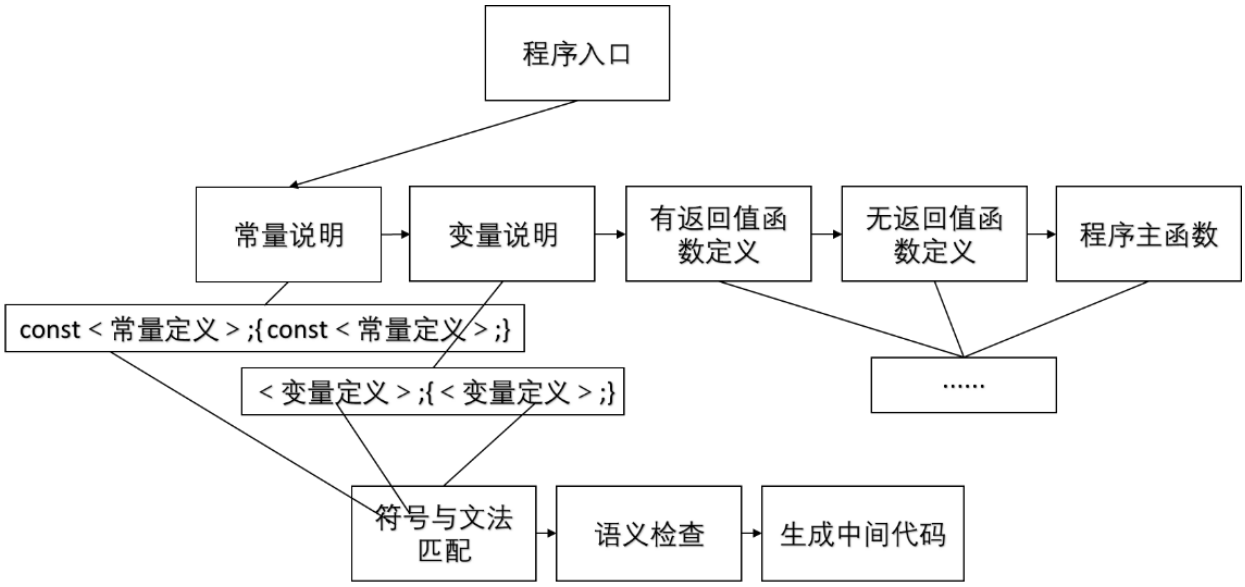


图5 中间代码生成

最后，此时程序已经获得了一种符合语法的表示结构，将输入到 MIPS 后端中生成代码，但是由于未经优化，代码效率可能不高，于是我进行了优化。

八. 优化器设计

我主要针对四元式做了一下几部分优化:

- 基本块划分
- 数据流分析
- 常数传播
- 消除公共子表达式
- 窥孔优化（删除死代码）

1.基本块划分:

针对基本块，我们首先将四元式输入到基本块的入口函数中，遍历四元式，进入基本块。通过对基本块入口的判断，获得基本块的入口，出口相关的信息。如果是基

本块的入口则给出一个新的基本块。如果是基本块的出口则在下一条语句中给一个基本块。如果不是入口或者出口则跳转到下一个语句中，不分配基本块。

下图是我的基本块划分的流程：

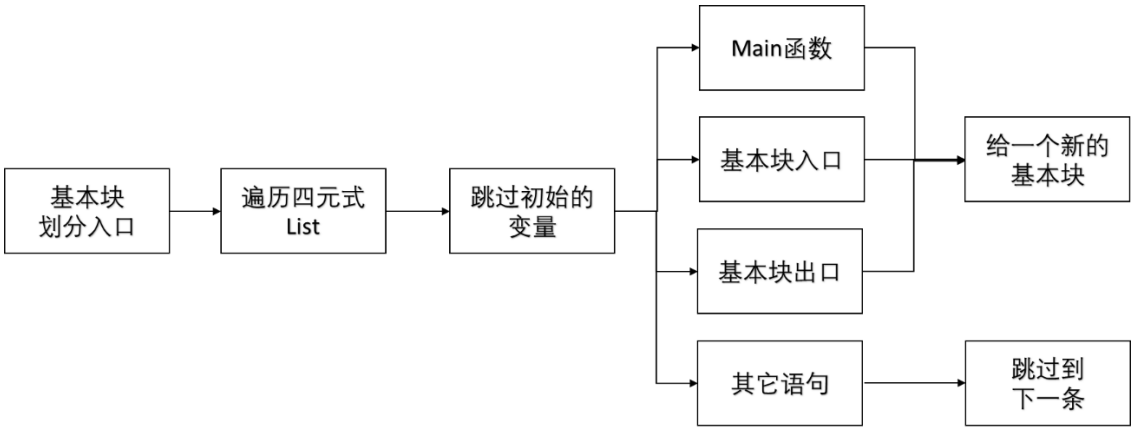


图6 基本块划分

最终，当我们生成一段代码，基本块及如下所示：
< call , Fib ， , Var_ > ---Block: 10
< int ， ， , Var_2 > ---Block: 11
当我们前一条语句跳转到了函数时，那么其和下一条语句即不属于同一个基本块。

基本块划分如上所示，基于基本块，我们可以分析数据流、并进行优化操作。

2.数据流分析：（部分实现）

针对每一个基本块都要进行数据流分析，我们需要清楚的知道数据流的流向，才能知道究竟基本块如何跳转。
我本次的设计思路是，通过遍历每个基本块，然后找到其需要跳转的基本块，这样就可以将它们连接起来。当只是普通的语句，而不是跳转，调用，退出等语句时，则跳转到对应的基本块。但是，当其是普通语句时，不跳转。

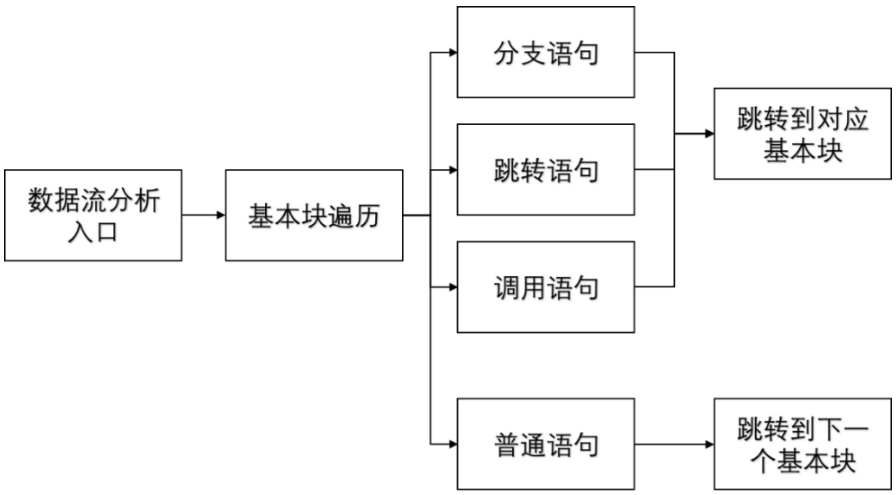


图7 数据流分析

3. 常数传播

常数传播是将 const 常数到四元式中，使其不用再通过访问内存的方式提高开销，所以可以极大程度的提高性能。

在常数传播部分，主要方法为：
1.遍历一次四元式，构造一个 const 常数表
2.在遍历一次四元式，将对应的数字在四元式中进行置换

这样在未来我们提取常数时，即可直接在汇编代码中生成常数，而不用在栈空间中取出常数，提高了代码运行的效率。

4. 消除公共子表达式

遍历四元式的每一个，消除掉没有使用价值的公共子表达式。注意必须在基本块内部消除。

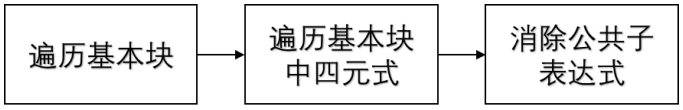


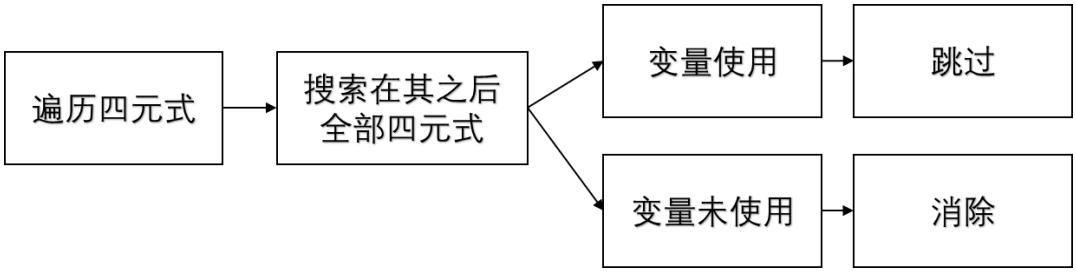
图8 消除公共子表达式

我们首先遍历每一个基本块，在基本块的每一个语句中，遍历基本块中的四元式。
如果四元式中产生了相同的公共子表达式，即将四元式删除。
最终即可消除掉全部的公共子表达式。

5. 窥孔优化（删除死代码）

当某个参数只是定义而最终没有使用时，可以认为这个参数是一个死代码，意味着其没有需要其作为参数完成对应的操作。
因此我们通过窥孔消除这些参数，即可极大程度提高程序效率。

主要的实现逻辑为：



- 如果最后没有使用这个变量，则消除。
- 如果最后使用了这个变量，则不消除。

这样即可实现窥孔优化，从而消除死代码。

九. MIPS32 后端和仿真

1.寄存器分配

在后端的设计中，首要考虑的是寄存器的分配问题，**我采用了固定分配的方式分配寄存器，这种方式的好处是不会因为函数的递归调用造成寄存器分配不足，而缺点也比较明显，就是在性能上并没有办法提高很多。**

最终我们在代码上看到的效果为：

```
dsubi   $sp,$sp,4
li   $2,5
syscall
sd   $2,-4($fp)
```

这段代码定义了一个变量，使用 scanf 得到这个变量，然后我在这里就直接将变量存放在了内存中。而在需要的时候，再将变量取出，例如：

```
ld   $25,-4($fp)
```

在这个过程中，寄存器始终会被释放，所以我们只需要固定的寄存器即可完成分配问题。

2.汇编语言生成设计

我们看到的目前是优化过后的四元式，四元式目前已经经过了高度的优化，达到了很高的性能。只需要将每一个四元式逐个转化为 MIPS 语言即可。

代码逻辑图如下：

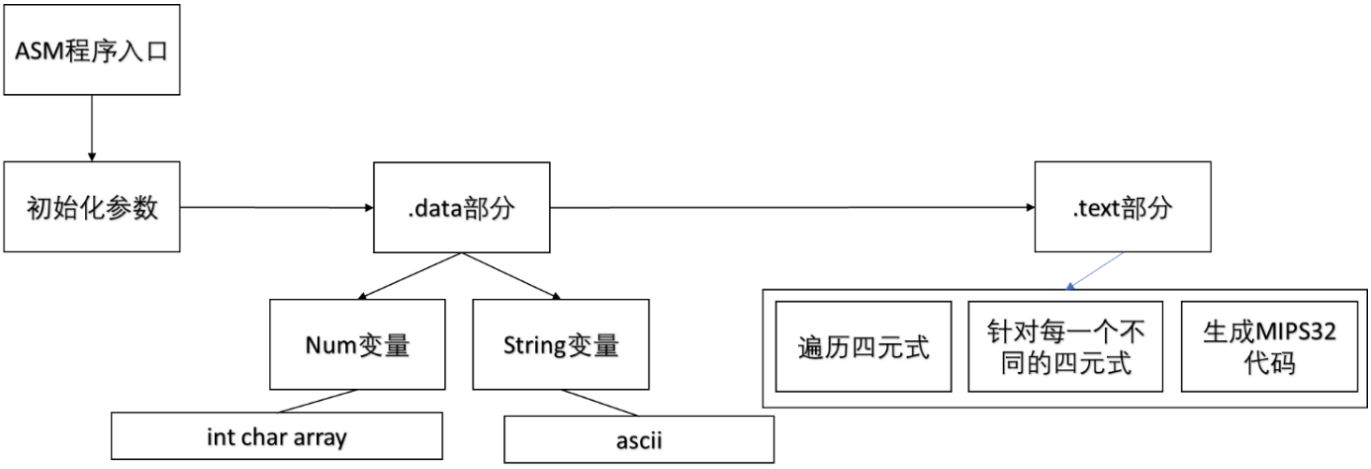


图9 汇编语言生成

MIPS 汇编通常分为.data 以及.test 部分。
我首先通过遍历，针对不同的变量生成不同的全局 data，例如 int，char 和 array 部分均生成不同的变量；在 string 部分生成 ascii 编码，这样即可生成需要的 data 部分代码。
例如生成部分代码如下：

```
.data:
a:   . word    0
charb:   . word    0
String_:   . ascii    "\n------\n\0"
```

那么，如何生成 MIPS32 代码呢？我们先得识别出究竟是那一条语句，才能跳转到对应的函数中去。

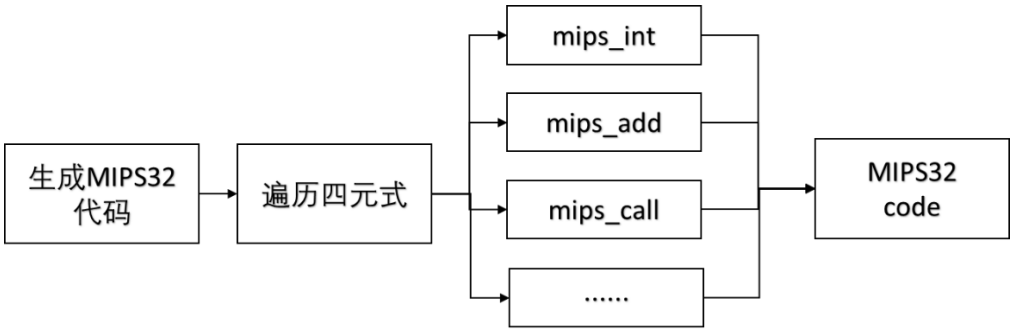


图10 生成代码的流程

每一条生成的 MIPS32/64 程序都是通过这种方式逐渐实现的。我们将四元式直接转化成对应的语句。

例如 Func：

```
#< func   ,int    ,      ,Fib     >

Fib:
```

例如 IF：

```
#< !=     ,n      ,0      ,Function_ >

    lw    $24,8($fp)
    li    $25,0
    bne    $24,$25,Function_
```

例如 Return：

```
#< ret    ,      ,      ,0       >
    li    $9,0
    move    $2,$9
    jr    $ra
```

按照这种方式，我们生成了全部的 MIPS 代码。

3.仿真
我们的代码最终在 Mars 上进行了仿真，通过了前面全部的测试样例。

十. MIPS64 后端的设计变化及龙芯派运行

由于 MIPS64 在指令和逻辑上和 MIPS32 存在很多不同，我最后为了在龙芯派上直接运行，设计了 MIPS64 指令系统下的后端。其代码逻辑和 MIPS32 类似，但是必须要遵循 gcc 编译器的规范，才可正确运行。

1.指令变化
指令由原来的 32 为全字指令，变成了半字的指令，因为 MIPS64 是 64 位的系统，而定义的字符的语义规范是不变的，因此我们必须知道的是我们需要使用半字的指令。

例如：
lw 变化为了 ld

addi 变化为了 daddi

我将 MIPS64 的文档中的定义总结了一下，保存在了 MIPS64.txt 中，可供查阅。
lb reg,imm(reg) - load byte
lbu reg,imm(reg) - load byte unsigned
sb reg,imm(reg) - store byte
lh reg,imm(reg) - load 16-bit half-word
lhu reg,imm(reg) - load 16-bit half word unsigned

我的设计由此开始，首先，将 MIPS32 的指令转化成了 MIPS64 的指令，完成了指令这一部分的转化。

2.GCC 汇编器

龙芯派上对应版本的 gcc 可将对应的 MIPS64 代码汇编，链接为实际的可执行程序。
具体的转化方法如下：

Gcc .s -o .o
Gcc .o a.out

a.out 即为可执行程序。
Gcc 汇编器在编译 MIPS64 时，遵循定义的语义规范，所以必须加入一个语句指示其 main 函数：

```
.globl  main
```

这样我们可以通过，并生成.o 文件。

但是 MIPS 在运行时遵循对应的开辟栈空间的规范，否则会引起段错误或者总线错误，我们得开辟部分栈空间提供给函数，防止函数产生段错误。

```
.frame  $fp,96,$31              # vars= 64, regs= 3/0, args= 0, gp= 0
.set    noreorder
.set    nomacro
sd      $28,72($sp) 开辟 72 字节空间
move    $fp,$sp
```

3.针对 scanf 和 printf 的处理
输入和输出在真机上十分复杂，涉及到协处理器 C0 和 C1 的不同定义和参数，非常麻烦。我们只能按照规范完成简单的输入输出：

这是协处理器的定义，用于输入和输出

```
.LC0:
        .ascii    "%d\000"
        .align    3         \\对齐的内存单元
.LC1:
        .ascii    "a\000"
        .align    3
.LC2:
        .ascii    "%s\000"
```

然后使用语句输出：

```
        lw        $3,20($fp)
        ld        $2,%got_page(.LC0)($28)
        daddiu    $4,$2,%got_ofst(.LC0)
        move      $5,$3
        ld        $2,%call16(printf)($28)
        move      $25,$2
        .reloc    1f,R_MIPS_JALR,printf
1:      jalr      $25
        Nop
```

使用语句输入：

```
        daddiu    $3,$fp,20
        ld        $2,%got_page(.LC0)($28)
        daddiu    $4,$2,%got_ofst(.LC0)
        move      $5,$3
        ld        $2,%call16(__isoc99_scanf)($28)
        move      $25,$2
        .reloc    1f,R_MIPS_JALR,__isoc99_scanf
1:      jalr      $25
        Nop
```

十一. 总结

最终实现的是一种基于 C0 文法的，可以根据 MIPS32/64 生成汇编语言的 C0 文法编译器。其支持的测试用例比较完备，并支持在 mars 和龙芯派上进行测试。同时，其在生成时完全没有使用工具，所有的过程均为自行设计完成的，从课程设置来看也符合我们对编译器的理解和掌握情况。

我在此过程中进行了编译器的开发，并且每一次开发的代码均保存到了网上，根据进度逐渐提交自己的代码，同样也是一次项目或者工程开发的训练过程。这种完整的训练过程值得我们每个人尝试与学习。另一方面，我采用 C0 文法开发的编译器后端采用了两种不同的 MIPS32 和 MIPS64 进行模拟，这样不仅可以实现在仿真平台上的运行，同时也可以在含有 gcc 平台的龙芯环境下进行运行。

我认为最终我的编译器达到了编译原理课程实践的基本要求。

十二. 自我评价

以上是我本次实验完成的全部内容，在这个过程是漫长而艰辛，在我看来也同样有价值，有意义。他让我学习到了编译这门技术及其最重要的使用和应用价值，对计算机系统的原理的编译环节进行了深刻的体会和掌握。

最后，感谢老师可以提供编译原理试点班的选项，让我们可以从实践中掌握编译器的原理，并探索自己的兴趣。