编译器课程设计申优文档

19376273 陈厚伦

一、写在前面

当写下第一句话的时候，我意识到我已经初步完成了一个完整的编译系统，可以说是自我选择计算机专业以来完成的最大的工程了，还是感觉非常有成就感的。设计与实现过程中，经历了调研、构思、论证、编码、测试等一系列环节，也经历了许多挫折与挑战，现在回想已经战胜了一个个困难，对复杂系统的驾驭能力有了显著的提高，专业本领取得提高的同时，自己在面对困难时的心态也更加坦然与平和，“回首向来萧瑟处，归去，也无风雨也无晴”。

在我个人看来，编译器设计与大二下学期的面向对象课程是紧密衔接的，在本学期编译器设计中，我将面向对象的思想深深地融入到工程的架构设计中，开闭原则、依赖倒转原则、接口隔离原则等设计原则均有体现，也能够适时灵活地使用工厂模式、单例模式等设计模式，这使得我整个编译器书写过程中，都能够较为轻松地驾驭自己的代码，并且通常也能够很快地定位程序错误。在代码优化阶段，好的架构

设计才真正体现出巨大威力，在优化的过程中，我轻松地实现了模块化优化设计，对每个历史版本均可兼容。

由于本文章为申优文档，所以本文的重点将放在代码优化部分，将会详细阐述优化的设计与实现细节，对于编译器其余功能模块，由于设计文档中已经进行了详细的说明，本文则着重讲解遇到的困难、解决方案和设计亮点，以及随着学期推进对各个功能模块的迭代性补充，并用编译器全局的视角进行分析。

二、功能模块设计

（一）词法分析

词法分析中遇到的第一个困难是**回滚机制的设计以及词法分析如何结束结束的问题。**词法分析设计了读字符和回滚机制，在读字符和回滚中实时对当前行信息进行维护，为了适应不同操作系统下换行符不同的情况，我在读源程序的时候统一了换行符。因为词法分析为了得到一个词，必然要读到第一个不属于这个词的字符，于是判断源程序最后一个词的时候必然要超出源程序，为此我在源程序的末尾手动添加了几个空白符，并通过捕捉文件尾异常来结束词法分析。

词法分析中遇到的第二个困难是**清洗注释。**初步设计中我是一边解析词汇一边记录是否进入注释状态，但是发现使我的词汇分析过程判断语句会变得特别多，增加了许多边界条件的分析。于是在后面的迭代中将清洗注释作为单独一遍进行，将源文件拷贝成一份没有注释的代码。注意的是注释虽然删掉，但是注释占用的行信息仍要记录，否则会影响错误处理的行数，我的做法是每删除一行注释就手动增加一个换行符`\n`；其次考虑了注释与字符串的关系，若注释符处于字符串中则不能认为是注释。

词法分析中遇到的第三个困难是`Token`**对象的设计**，即词法分析返回的词对象要包含什么信息，这一点是在后面的迭代中不断补充的，词对象包含的信息越多，越能简化后面的语义分析与错误处理，我的`Token`类中包括：词的原始字符串、词的类型、词的行号、词的值（如果是`number`）、词中占位符的个数（如果是`FormatString`）。

（二）语法分析

我的语法分析使用的是递归下降法。

语法分析中遇到的第一个困难是**文法预处理。**实验指导手册提供的文法存在左递归，为此我使用扩充的`BNF`表达式消除了左递归，并进行了公共部分提取等其他的化简。

语法分析遇到的第二个困难是**预读与回滚机制的设计。**语法分析的预读与回滚比词法分析的预读与回滚复杂得多，在我的实现中为了使语法分析更加清晰，每次得到一个语法成分之后都会将额外读取的进行回滚，这样每进入一个新的语法成分的分析程序中均是从第一个词开始解析。我的语法分析实现中不仅会预读多个词汇，还会预读多个语法成分，所以除了一般的读取一个词与回滚一个词之外，我还设计了`mark()`和`restore()`方法，遇到复杂的预读问题时，通过`mark()`标记起点，使用`restore()`可以直接回到预读的起点，这样可以解决预读语法成分时的回滚问题。预读的过程中还有一个困难是因为程序可能存在错误，所以在读取可有可无的语法成分时，我的程序预读这个成分的`FIRST`集，而不是读取下下个成分的`FIRST`集，因为下下个成分可能出现错误。

语法分析遇到的第三个困难是`AST`**的构建。**我将每种语法成分视为树的节点，通过节点之下悬挂子节点对文法进行抽象表达，由于文法中一个非终结符的文法可能有多条，所以一个语法成分可能悬挂不同的子成分，于是我在一边构建`AST`时一边记录分支信息，同时也尽可能记录了当前语法成分的语义信息，这样后面递归遍历`AST`就能够非常容易地建立符号表并进行中间代码生成。

（三）符号表管理

为了实现前后端解耦，我实现了两个符号表：栈式符号表和汇编符号表。在生成中间代码之前使用栈式符号表，将中间代码翻译成汇编符号表时使用汇编符号表。栈式符号表主要方便错误处理与简单的前端预处理功能，汇编符号表主要用来管理目标程序代码的运行时空间。

栈式符号表是在语法分析结束之后递归遍历`AST`构建的。栈式符号表使用单例模式，一边遍历`AST`一边更新栈式符号表。**在这一部分我实现的亮点是建立符号表的同时对变量进行了重命名**，`block`内的变量与其外面的变量重名时会将外面的变量暂时覆盖掉，对后面的生成中间代码非常不方便，于是我对所有的变量进行了重命名，这样在后面各个阶段均可以通过变量名唯一确定变量。声明变量时对变量建立别名并进行统一标号，当引用变量时查找栈式符号表，找到距离最近的符号表项，生成中间代码的时候用最近的符号表项中记录的别名。特别的，我也对全局变量的别名使用`-GLOBAL-`进行标记，在后面代码优化阶段需要频繁判断变量是否是全局变量，由于构建符号表阶段已经进行了标记，这一步会变得非常方便。

由于`AST`递归访问完毕后，栈式符号表已经弹空，于是为了翻译MIPS，我构造了一个汇编符号表，将会在目标代码生成中描述。

（四）错误处理

作业中定义的错误包含词法错误、语法错误和语义错误。分别在词法分析、语法分析和语义分析进行。当进行完语义分析之后，如果错误列表非空，则启动报错并停止编译程序的运行。

错误处理的一个难点是**报错顺序。**由于我是按照词法错误、语法错误、语义错误的顺序进行报错，并不是按照源程序的书写顺序，但是由于规定了每一行至多有一个错误，所以最后输出错误的时候对错误列表按照行号进行排序输出即可。

错误处理的另一个难点是**如何跳读**使得编译器能够在检测出第一个错误后继续检测剩下的错误，这主要体现在词法错误和语法错误中，我的做法是：对词法错误，报错之后将其视作一个正确的词；对语法错误，缺少`)`，`]`时报错之后认为其存在。

错误处理阶段我的一个亮点是对**每个错误设定专门的报错接口**，减少大量的冗余代码。

（五）语义分析与中间代码生成

语义分析与中间代码生成的重点和难点是中间代码的设计，我的设计中建立了非常多的中间代码种类，每个中间代码的功能都是单一且独立的，这对于后面生成目标代码与代码优化非常有益。

我的中间代码是在递归访问`AST`的同时一边建立栈式符号表一边创建的，根据`AST`可以非常容易地以递归的方式得到中间代码。

本部分我的设计亮点之一便是**中间代码的设计与管理。**为了对中间代码进行更好的分类管理，`MidCode`类管理所有中间代码的公共属性和公共方法，各中间代码类继承`MidCode`类。中间代码可以有不同的分类方法，我使用不同种类接口对中间代码进行分类，接口中定义了相应大类的中间代码应当实现的方法。我设计了如下的抽象接口对中间代码进行管理：

序号	接口名称	接口作用
1	**ABSTRACT_DECLARE**	管理所有声明语句
2	**ABSTRACT_DEF**	管理所有定义语句
3	**ABSTRACT_JUMP**	管理所有控制流语句
4	**ABSTRACT_LABEL**	管理所有标签语句
5	**ABSTRACT_OUTPUT**	管理所有输出语句
6	**ABSTRACT_USE**	管理所有使用语句

访问某一类中间代码时只需要按照该类的接口进行访问，顶层不需要关心具体是哪一种中间代码，底层对象对上层透明，减少了大量的判断语句，从根本上杜绝了判断语句遗漏部分条件而出现错误的现象。我的中间代码设计如下：

序号	种类	域1	域2	域3	语义	实现接口
1	**ADD**	和	加数1	加数2	加法	DEF、USE
2	**SUB**	差	被减数	减数	减法	DEF、USE
3	**MULT**	积	乘数1	乘数2	乘法	DEF、USE
4	**DIV**	商	被除数	除数	除法	DEF、USE
5	**MOD**	余	被除数	除数	模	DEF、USE
6	**EQL**	结果	操作数1	操作数2	相等置1	DEF、USE
7	**GEQ**	结果	操作数1	操作数2	大于等于置1	DEF、USE
8	**GRE**	结果	操作数1	操作数2	大于置1	DEF、USE
9	**LEQ**	结果	操作数1	操作数2	小于等于置1	DEF、USE
10	**LSS**	结果	操作数1	操作数2	小于置1	DEF、USE
11	**NEQ**	结果	操作数1	操作数2	不等于置1	DEF、USE
12	**NEG**	结果	操作数		取相反数	DEF、USE
13	**NOT**	结果	操作数		取逻辑反	DEF、USE
14	**ASSIGN**	结果	操作数		赋值	DEF、USE
15	**ARRAY_ADDR_INIT**	数组指针			为数组指针赋地址初值	DEF
16	**CONST_ARRAY_ADDR_INIT**	数组指针			为常量数组指针赋地址初值	DEF
17	**BEGIN_WHILE**				while的开始标签	LABEL
18	**END_WHILE**				while的结束标签	LABEL
19	**IF**				if的开始标签	LABEL
20	**ELSE**				else的标签	LABEL
21	**END_IF**				if的结束标签	LABEL
22	**LABEL**				自定义标签	LABEL

序号	种类	域1	域2	域3	语义	实现接口
23	**SUPPLEMENT_LABEL**				用于基本块划分的辅助标签	LABEL
24	**BEZ**		条件	指定标签	等于0跳转到指定标签	JUMP、USE
25	**BREAK**			指定标签	break	JUMP
26	**CONTINUE**			指定标签	continue	JUMP
27	**GOTO**			指定标签	无条件跳转	JUMP
28	**RETURN**		返回值（可无）		函数返回	JUMP、USE
29	**PRINT_INT**		待打印操作数		打印操作数	OUTPUT、USE
30	**PRINT_STR**		待打印字符串		打印字符串	OUTPUT
31	**CONST**	常量名			声明常量	DECLARE
32	**CONST_ARRAY**	数组常量名	维度1	维度2	声明数组常量	DECLARE
33	**PARA_DECLARE**	形参名			声明形参	DECLARE
34	**PARA_ARRAY_DECLARE**	数组形参名	维度1	维度2	声明数组形参	DECLARE
35	**VAR**	变量名			声明变量	DECLARE
36	**VAR_ARRAY**	数组变量名	维度1	维度2	声明数组变量	DECLARE
37	**CALL_INT**		函数名		调用int函数准备工作	

序号	种类	域1	域2	域3	语义	实现接口
38	**CALL_VOID**		函数名		调用void函数准备工作	
39	**END_CALL_INT**	目标变量	函数名		调用int函数，将返回值赋给目标变量	
40	**END_CALL_VOID**		函数名		调用void函数	
41	**PUSH**		操作数		将操作数压栈	USE
42	**LOAD_ARRAY**	dst	src	index	`dst = src[index]`	DEF、USE
43	**STORE_ARRAY**	dst	index	src	`dst[index] = src`	USE
44	**FUN**		函数名		创建函数	
45	**PARA**		形参名		从栈取形参的值	DEF
46	**PARA_ARRAY**		形参名		从栈取数组形参的（地址）值	DEF
47	**GETINT**	dst			`dst = getint()`	DEF
48	**START_PROGRAM**				程序入口	
49	**END_PROGRAM**				程序出口	

每个中间代码都有高度的独立性，每个中间代码也记录了其所在的函数信息，不仅简化了每一条中间代码翻译成`mips`的复杂度，也为翻译与优化提供了足够细粒度的信息，优化阶段对中间代码的添加与删除也变得更加容易，减少了出现风险的可能，中间代码的可扩展性好，在代码优化中能够得到体现。

本部分的另一个难点是**数组**。在我的设计中，数组是按照**指针常量**的方式处理的，每声明一个数组便创建同名的指针变量，并将数组的首地址赋值给这个指针变量，这样的好处是简化了数组传参，将数组指针当做一个普通变量即可。数组的模板被记录下来，在生成中间代码时，已经消解了数组维度的概念，在中间代码中只留下基地址和偏移两个概念，减轻了目标代码生成的压力。

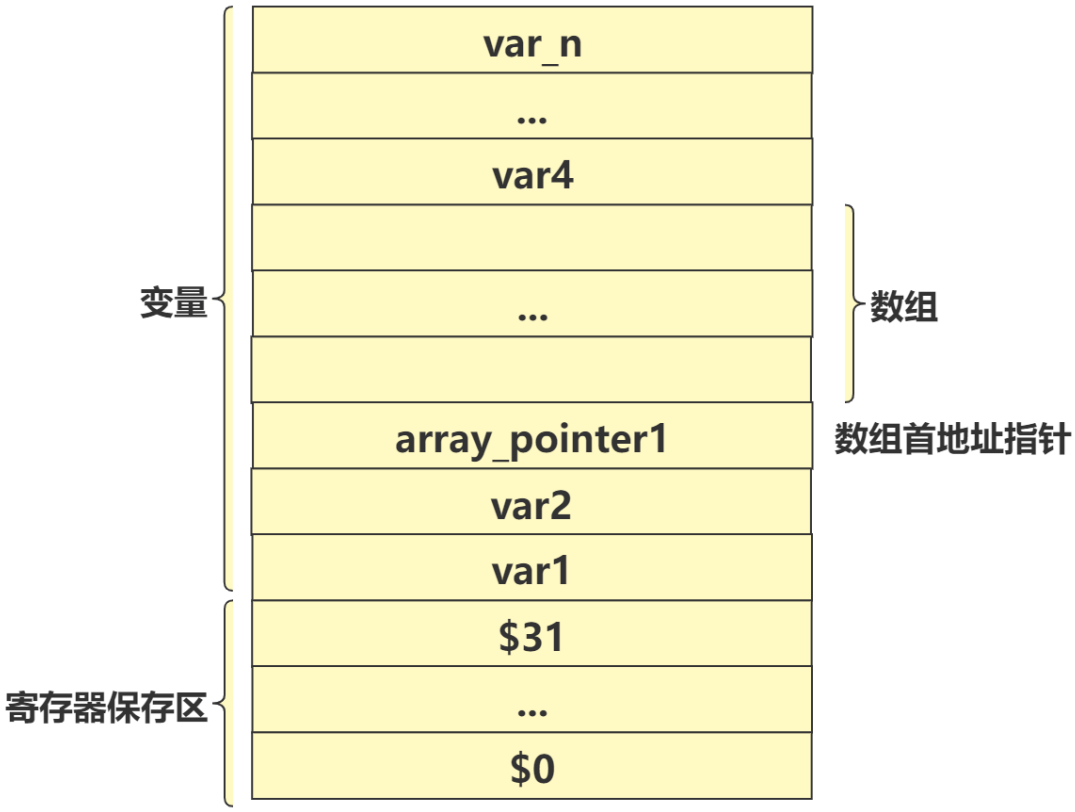
本部分的最后一个难点是**短路求值**。此部分有一定的技巧性，我采取了控制流法进行了实现，即在逻辑上将表达式拆解成一层一层的`if`，将`&&`视作`if`的嵌套，将`||`视作`if else`，在具体的实现上通过在适当的位置添加`JUMP`语句和`LABEL`实现。

（六）目标代码生成

目标代码直接对每条中间代码进行翻译得到，其中打印的字符串根据`%d`切割，字符串常量放在`.data`段，全局变量放在`$gp`位置，局部变量放在函数的调用栈中。

目标代码生成的一个难点是**运行时空间管理**。我的编译器在此部分的一个亮点是实现了一个单独的汇编符号表用来进行运行时空间管理。由于栈式符号表查找速度慢，符号表随着遍历不断动态变化，而且语法分析阶段的符号表中很多信息并不在目标代码生成阶段使用，为了更好地解耦合，我设计了汇编符号表，全局有一个静态符号表，每个函数也有一个符号表，因为之前已经对变量进行了重命名，所以汇编符号表均使用`HashMap`组织，加快了查询速度。每个函数的调用栈预先分配`32*4`字节空间用来保存寄

存器，在此基础上给变量分配空间，通过扫描中间代码中的`ABSTRACT_DECLARE`语句，进行空间分配，当遇到变量的时候分配`4`个字节，当遇到数组时，先分配`length*4`个字节空间，再分配`4`字节作为数组的指针，对数组的访问转化成对指针的访问。函数栈的结构可如下图所示：



运行时空间管理

目标代码生成的另一个难点是**寄存器分配**，在代码生成作业中，我实现了变量读写只通过`lw`和`sw`的版本以保证编译程序的正确性，寄存器分配将在代码优化阶段展开。

三、代码优化

在优化阶段，我的总的设计理念是**开闭原则**，尽量不对其余部分进行修改，通过扩充模块实现新的功能，虽然可能会有冗余部分使得编译器性能有所下降，但是增加了各个模块之间的独立性，有很好的可维护性。

（一）数据流分析

1、基本块划分与数据流图的构建

数据流分析对于全局优化有着基础性的意义，而数据流分析的第一步也是关键一步便是数据流图的构建。我的数据流图构建分为基本块划分与数据流图构建部分。

在架构设计层面，由于函数调用要保存现场，所以我认为函数调用不破坏基本块，每个函数都是一个相对独立且封闭的部分，应当拥有一个独立的数据流。所以我用两个类进行数据流的管理，`Flow`类和`SubFlow`类，`Flow`管理全局数据流并定义了全局数据流的方法，并对`SubFlow`进行管理，`SubFlow`管理函数内的数据流并定义了函数内数据流的方法，全局也作为一个函数名为`&global`的假函数。基本块的划分与数据流图也均对每个函数单独进行，所以先将中间代码根据其所处的域分配到相应函数的`SubFlow`.

基本块划分的难点是在顺序排列的中间代码中找到基本块的入口与出口，同时我的基本块划分是基于标签的划分方法，在中间代码生成的过程中生成标签，每遇到一个标签便切割一次基本块，基本块的名称即标签的名称，由于跳转语句也会使用标签，因此这样做连接基本块会变得容易。基本块的切分方法如下：

1、生成分支/跳转语句时在其后面插入一条`LABEL`，分支/跳转语句的一个`dst`已经是语句中的`LABEL`（`RETURN`设置为`"#Exit"`），如果生成的是分支语句，则将分支语句的另一个`dst`设定为其后面的`LABEL`。

2、对函数的中间代码进行扫描。如果遇到`ABSTRACT_JUMP`语句，则找到部分基本块的终点，将基本块的`dst`分别设定为`ABSTRACT_JUMP`的`dst`；如果遇到`ABSTRACT_LABEL`语句，则找到新的基本块的起点，如果前面的基本块不以`ABSTRACT_JUMP`结束，则将前面的基本块的`dst`设定为新的基本块的名称。

基本块划分过程中已经对每个基本块的`dst`信息进行了记录，所以构建数据流图只需要遍历基本块，将各个基本块通过前驱与后继关系关联起来即可。

对每个函数的数据流图中补充一个`#Exit`的结束基本块，所有`return`结尾的基本块均指向`#Exit`基本块，至此每个函数的流图均为单入口单出口。函数声明语句由于处于全局作用域中被分到了`&global`函数中，于是再将函数声明语句从`&global`中转移到各个函数的入口基本块处，至此将各个基本块顺序拼接即可得到全部的中间代码。方便后面的优化对基本块中间代码进行增删后能够将合成新的中间代码。

2、活跃变量分析

我的编译器中实现了以语句为颗粒度的活跃变量分析。按照课本上的算法可以以基本块为单位计算出在每个基本块的`in`和`out`，即基本块入口和出口处的活跃变量。接下来我对基本块内进行遍历得到了基本块内的活跃变量信息，最终得到了每条中间代码处的活跃变量，算法如下：

对于基本块`B`，初始化集合`willUse = out[B]`，**逆序遍历**基本块`B`中所有的中间代码，对于每条中间代码`m`，如过`m`是`ABSTRACT_DEF`并且`willUse`中有`DEF`语句产生的`def`，则从`willUse`中移除`def`，然后将`willUse`加入到`m.willUse`中，如果`m`还是`ABSTRACT_USE`，则将其所有的`use`加入到`willUse`中。

再初始化集合`alreadyDef = in[B]`，**顺序遍历**基本块`B`中所有的中间代码，对于每条中间代码`m`，将`alreadyDef`加入到`m.alreadyDef`中，如果`m`是`ABSTRACT_DEF`，则将`m`产生的定义`def`加入到`alreadyDef`中。**语句`m`处的活跃变量`m.active`则为`m.alreadyDef`和`m.willUse`的交集。**

在上述算法计算`willUse`时体现出了定义语句的`kill`作用，原因是因为一个变量在基本块内可能被赋值多次，**如果在基本块内实现SSA**，即在一个基本块内，一个变量至多被赋值一次，则可以简化掉定义语句的`kill`，这一点是容易做到的，需要通过下文的到达定义分析。

3、到达定义分析与du关系

我的编译器中实现了以语句为颗粒度的到达定义分析，最终得到了**du链**和**ud链**，即`ABSTRACT_DEF`语句以及所有其可达的`ABSTRACT_USE`语句，和`ABSTRACT_USE`语句以及所有可能到达其的`ABSTRACT_DEF`语句。按照课本上的算法可以得到基本块间的到达定义数据流和所有的`du`链，对基本块内的代码进行遍历对`du`链进行基本块内的**延拓**，在得到`du`链的同时记录`ud`链，并将`ud`链的信息保存在每条中间代码中。

通过`du`链可以得到网，当同一个变量的多个`du`链拥有公共的使用点时可以合并为一个网。这个问题可以**转变为求连通分量的图问题**。给定一个变量的所有`du`链，每条`du`链均视为图的节点，如果两个`du`链有公共的使用点，则认为两个`du`节点之间存在一条边，通过`BFS`可以求得无向图的所有连通分量，即可得到变量的所有网。

一个变量的不同的网可以视作不同的变量，根据这一思想，**我对每个网得到的变量进行了重命名**，如此每个基本块内实现了SSA。重命名之后可能会存在没有被重命名的变量，说明这些变量的使用没有可以到达的定义，或是变量没有赋值直接使用，或者变量所在的语句是不可达语句。对于前者，因为变量没有赋值直接使用，所以取得任何值都是合法的，对于后者，由于语句不可达，将语句直接删除即可。

重命名之后，由于产生的新的变量，所以对于新的变量要在重命名之前该变量的位置补充变量声明语句，让`MIPS`翻译器为其分配空间，如果重命名的是形参，`PARA_DECLARE`中的形参名称要与`PARA`的形参名称一致，对于形参的其他重命名，这些重命名的使用无法使用到函数调用传进来的值，这些重命名严格来讲不属于形参，所以在其使用处补充一条`VAR`声明即可。

进行了重命名之后再进行活跃变量分析，能够使得活跃变量分析的结果更加准确。

（二）中间代码优化

1、常量替换

在生成中间代码的阶段，对非数组的常量进行替换，因为常量都是编译可求值的，所以在常量定义处将常量的值记录下来存入符号表，在生成中间代码时，访问符号表将常量用其值替换。

2、局部复制传播与常量合并

局部的复制传播与常量合并是结合在一起的，首先先进行常量合并，将尽可能多的语句转化为赋值语句。对于运算类型语句，如果参与运算的操作数均为常数，则编译期可求值，可以转化为赋值语句。如果其中有一个操作数是常数，还可以进行如下的合并操作：

```
a = b + c && c =  0  ==> a = b;
a = b - c && c =  0  ==> a = b;
a = b * c && c =  0  ==> a = 0;
a = b * c && c =  1  ==> a = b;
a = b / c && c =  1  ==> a = b;
a = b / c && b =  0  ==> a = 0;
a = b % c && c =  1  ==> a = 0;
a = b % c && c = -1  ==> a = 0;
a = b % c && b =  0  ==> a = 0;
```

局部的赋值传播算法如下，对基本块顺序遍历，如果当前语句是`a = b`，则将其后面`a`被`kill`之前所有对`a`的使用替换成`b`，遍历结束之后，再对所有的代码进行一次常量合并。

3、全局常量传播

全局的常量传播基于`ud`链，对于中间代码`m`中的使用`use`，如果只有一条`DEF`可以到达`use`，并且`DEF`是`ASSIGN`语句且赋值的右值是常数，则可以将常数替换`m`中的`use`，实现全局的常量传播。

通过实验发现，先进行一次局部复制传播与常量合并，再进行一次全局常量传播，最后再进行一次局部复制传播与常量合并，就已经可以达到非常不错的效果。

4、局部死代码删除

局部死代码删除基于活跃变量分析，以基本块为单位进行，如果一个变量在其定义之后没有使用，即变量定义之后在基本块内没有使用，而且不在基本块的`out`中，则可以删除其定义语句。进行完一遍之后，再做活跃变量分析，再进行局部死代码删除，如此循环迭代，直至中间代码不再发生变化。

对代码的删除有时并不能直接删除，因为有许多代码有副作用，如函数调用，输出输出，形参接收等，于是我对删除的代码设置删除状态，这样在翻译`MIPS`的时候如果中间代码处于删除状态时，则执行其删除后行为。

5、全局死代码删除

在局部死代码删除中，发现对于如下情况：

```
while (n < NUM) {
    d = d * d % 1000;
    n = n + 1;
}
```

当这段代码之后变量d如果没有被输出，没有参与函数传参，没有作为控制流判断信息等，`d = d * d % 1000;`是可以删除的，但是根据局部死代码删除策略，无论迭代多少次，都是无法删除的，因为语句自己使用了自己的定义。所以这里提出了**全局死代码删除**策略。

全局死代码删除基于`du`链和`ud`链条。当一个变量没有在副作用语句中使用时，则其定义点的语句可以删除，如果一个变量在副作用语句中使用时，则所有可以到达这个使用的定义语句也标记为副作用语句。

原子的副作用语句包括输入输出、函数调用、对全局变量的赋值、对数组元素的改变、控制流语句，具体展开如下：

- `ABSTRACT_DEF`且定义的变量是全局变量
- `STORE_ARRAY`
- `PRINT_INT`
- `PRINT_STR`
- `GETINT`
- `CALL_INT`
- `CALL_VOID`
- `PUSH`
- `END_CALL_INT`
- `END_CALL_VOID`
- `ABSTRACT_JUMP`

算法如下：

初始化集合`Side`为空，遍历中间代码找到所有的原子的副作用语句，将这些副作用语句加入`Side`，重复进行如下操作，直到`Side`不再变化：遍历`Side`中语句，如果语句是`ABSTRACT_USE`，将其所有的`use`根据`ud`链，找到能够到达这个使用的定义语句，加入`Side`中。

最终，将不在上述`Side`集合中的`ABSTRACT_DEF`语句删去即可。

6、无用循环删除

首先对前文副作用代码的定义进行松弛，**只将跳转类中间代码中的`RETURN`视为副作用代码**，对于**其余跳转语句另行讨论。**

如果一个循环中**不含有副作用代码**，且**当前层循环中的定义语句无法传播到本层循环之外的跳转类语句**，则本层循环可以删除，即在循环的开头添加`GOTO`语句，`GOTO`到循环结束的`label`。如果本层循环下无循环，且本层循环下的子循环已经被删除，则可对当前层循环进行同样操作，直至离开最外层的循环。

判断当前层循环中的定义语句是否传播到本层循环之外的跳转类语句算法如下：定义集合`S`，初始化为空，对每个非副作用的定义语句并放入`S`，如果是`USE`，找到其所有的`use`，并将这些`use`放入`S`，对新添加的代码，如果是`DEF`，找到其所有的`use`，放入`S`，执行同样操作，直至`S`收敛。如果最终`S`中含有不在循环中的跳转语句，则可以判定：当前层循环中，有定义语句传播到了本层循环之外的跳转类语句。

（三）目标代码优化

目标代码优化最重要的是寄存器分配。寄存器分配是影响全局的一个重要因素，寄存器分配得好有时可以将其他优化的性能更好地释放出来，达到四两拨千斤的效果。我在这一部分实现了课本上的主要方法之后，又进行了诸多**精细化处理**，尽可能将寄存器分配的效率释放出来。

整体架构上，我实现了如下接口，具体的实现将在临时寄存器一节展开。

```
// 为name申请一个读寄存器
Register applyReadReg(String name, MidCode midCode);
// 为name申请一个写寄存器
Register applyWriteReg(String name, MidCode midCode);
// 暂时借用一个寄存器
Register borrowReg(MidCode midCode);
// 归还一个暂用的寄存器
Register returnReg(Register reg, MidCode midCode);
// 清空已经借出的寄存器
Register clearBorrow(MidCode);
```

1、全局寄存器分配

全局寄存器分配使用了图着色分配算法。根据到达定义建立定义使用网，对变量重命名，再进行活跃变量分析，根据“**一个变量定义处另一个变量是否活跃**”原则建立冲突图，进行全局寄存器的图着色分配。

在实现中，我以每个函数作为进行冲突图构建与全局寄存器分配的基本单位。只有作用域跨越基本块的变量才有资格竞争全局寄存器，**作用域跨越基本块的变量是函数的所有基本块的`in`与`out`的并集**，且全局变量不能够使用全局寄存器，否则伴随函数调用可能出现值不一致的错误。

可以当做全局寄存器分配的有如下寄存器，如果全局寄存器没有被用尽，剩下的在函数内可以“借出”作为临时寄存器使用。

```
$s0,$s1,$s2,$s3,$s4,$s5,$s6,$s7,$v1,$a1,$a2,$a3,$k0,$k1
```

用冲突图分配寄存器中，我在算法的每一步执行前都将节点按照`degree`从大到小排序，在冲突图中寻找第一个连接边数小于`K`的节点可以使用二分，在标记不分配全局寄存器节点时，我**选择的是当前`degree`最大的节点**，从而移走这个节点后最大限度减少冲突，提升全局寄存器的利用率。

分配时尽可能使用最少数目的全局寄存器，具体实现上每次寻找可用寄存器时都**按照固定的顺序**找，则一定能够使用最少数目的寄存器，从而增加临时寄存器的数目。

2、临时寄存器分配

临时寄存器分配有很多门道，小细节做好可以很好地提升性能。架构设计上，我对每一个函数建立了一个寄存器分配器，由寄存器分配器管理当前函数的临时寄存器池，全局变量分配完毕后，初始每个函数的临时寄存器池，每个函数可用的临时寄存器包括如下的几个临时寄存器和当前函数没有使用到的全局寄存器。

```
$t0,$t1,$t2,$t3,$t4,$t5,$t6,$t7,$t8,$t9,$fp
```

临时寄存器部分非常难的一部分是**临时寄存器池的管理**。

临时寄存器池的管理包括很多方面，包含但不限于**临时寄存器的替换策略，寄存器的申请方式，临时寄存器的写回策略，寄存器的保存，临时寄存器池的清空**等。

我的所有寄存器申请相关的接口均传入了`MidCode`，如此可以**利用当前`MidCode`的上下文信息，更好地分配临时寄存器。**

临时寄存器是一个随用随申请的资源，因为中间代码的上下文信息是已知的，所以我使用**OPT置换策略**，每次优先替换当前`MidCode`处不活跃变量和常数，如果均活跃，则替换最晚使用的变量对应的寄存器。

寄存器的申请方面，我设计了上文所展示的5个接口，申请读写寄存器时，如果是变量已经拥有全局寄存器，则直接返回其拥有的全局寄存器，否则检查是否已经拥有临时寄存器，有则返回，否则检查是否有空闲寄存器，有则返回，否则检查是否有不活跃变量或者常数，有则返回，否则根据OPT策略找到一个寄存器。申请读寄存器中如果变量或者常数没有寄存器则需要将变量值加载进寄存器或将常数值写进寄存器，申请写寄存器时要对寄存器置脏位。当替换了一个活跃变量的寄存器时，如果是脏的，要写回内存。在寄存器申请方面，**我的一个设计亮点是提供了寄存器借用接口。**并不是所有的中间代码的变量数与寄存器数都是相等的，例如在除法优化、访存数组时，需要额外多使用1个寄存器，最简单的策略是专门为其预留一个寄存器如$t0$，但是这样的中间代码的数量是比较少的，这样做相当于牺牲掉了一个寄存器。而对于这种生命周期局限在一条中间代码的范围内的寄存器，我设置了借用的接口，使用前借用，使用后归还，借用寄存器的策略与为变量申请寄存器类似，也是优先借出空闲寄存器或者不活跃变量对应的寄存器。在寄存器申请方面，还需要注意的是，**读的寄存器之间不能冲突，写的寄存器不能与读的寄存器冲突，借出的寄存器不能与读、写和已借出的寄存器冲突。**

在函数调用处需要保存现场，我的策略是全局寄存器均保存，保存在函数调用栈的寄存器保存区。临时寄存器只保存活跃变量，如果脏，保存回变量所在位置。函数调用结束后恢复现场，从调用栈恢复全局寄存器，从变量所在位置恢复临时寄存器。

在基本块的结尾，对临时寄存器池进行清空，如果变量活跃且寄存器脏，则写回内存。

（四）其他优化

1、乘法优化

当乘法指令中一个操作数是常数且是2的幂次时，可以将`mult`指令转换成`sll`.

2、除法/求模优化

除法实现了**魔数优化**，优化与论文《Division by invariant integers using multiplication》[1]一致，取整方式为**向零取整。**值得注意的是，实现论文中计算m_{low}与m_{high}时如果使用`long`数据类型，计算的中间过程可能会出现`long`数据类型溢出从而导致一些边界数据的魔数时出现错误，解决这个问题的办法是使用`Java`中的`BigInteger`类型。

求模运算可以如下转换：

```
a % b = a - a / b * b
```

其中乘法再可以使用乘法优化。

3、全局变量转为局部变量

全局变量在数据流分析中都是作保守估计的，由于`main`函数只执行一次，所以只在`main`函数中定义和使用的全局变量可以转化为`main`中的局部变量，从而使得数据流分析的结果更加精确。

4、窥孔优化

在我的中间代码中，在表达式计算中会将右边的表达式计算出来并赋给一个临时变量，再将临时变量赋给目标变量，所以常常出现这样的中间代码：

```
#T23 = #T21 + #T22
@a = #T23
```

扫描中间代码时通过向后多看一条将其优化为

```
@a = #T21 + #T22
```

四、写在后面

一学期的编译课程设计让我在磨砺与挫折中顽强成长。最直观的便是锻炼了抗压能力，培养了时间规划能力。编译课程不断推进，而且本学期还有数据库、算法等其余的重要的核心专业课和许多一般专业课，加起来每周都有许多的工作量，所以如何分配好时间实现效率最优是非常重要的，而且编译器实现是有相当难度的，在实现的过程中对自己需要花费的时间是很难有把握的，所以这就迫使我努力加快效率，遇到困难不要慌，再多的事情也只能一件一件地做，努力做好当下即可，同时在实现过程中一边推进也在一边反思，不断调整自己的时间安排，最终完成好编译器的全部工作。

实现一个编译器也提升了架构能力。编译器是一个非常综合与复杂的系统，我使用Java作为编译器作为开发语言，将面向对象的思想使用到了极致，在迭代开发地过程中能够较好地控制复杂度，优良的架构也在代码优化阶段给予了我十足的回报。同时这个不断思考的过程也让我对架构设计有了更加深入的思考，也让我直观地看到架构好坏带来的影响，动手实现与能力提升是双向促进的。

通过完整地完成一个编译器，我也复习了理论知识，明白了一种高级语言是如何编译成汇编语言并在机器上执行的，对于“C语言为什么要这么设计”这个问题也有了自己的答案。在实现的过程中，我对理论课上讲的各种算法实现了一遍，记忆更加深刻，同时也让我切切实实体会到理论课上讲的算法是做什么用的，也纠正了自己对知识的一些错误理解。动手实现一个编译器让我将理论投入实践，同时又不拘泥于理论，对理论的知识加入自己的思考，对理论知识的应用加以拓展，将知识真正地化为己用。

本学期选择了自己熟悉的Java作为开发语言，比较遗憾地错过了C++的一个很好的学习机会。从编译器的诞生过程看，本来就是一个“鸡生蛋、蛋生鸡”的一个不断自展的过程，逐层编译直到得到一种高级语言，所以使用C++书写类C文法的编译器才更符合编译器发展的自然过程。同时C++在当今也有着广泛的应用范围，无论如何也是非常有必要学习的。在书写编译器的过程中，我也踩了许多坑，也有许多次因为思考不周导致的重构，也带来了许多错误的隐患，开发复杂系统的能力是非常重要的，设计实现编译器是一个很好的锻炼过程，在这方面我仍需要不断提升，向优秀看齐。

最后要感谢一路上帮助过我的老师、助教和同学们！他们或是启发我新的思路，或是帮助我找到新的错误，总之能够完成一个编译器，他们的功劳是不可磨灭的。

五、参考文献

[1] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication[J]. ACM SIGPLAN Notices, 1994, 29(6) : 61-72.