《编译技术》课程设计文档

姓名：黄雨石

学号：20376156

词法分析

编码前设计

首先定义`Token`类、`Type`枚举类、`Lexer`类，`Token`用于记录每个单词的类型、值、所在行数，`Type`则是由类别码构成的枚举类、`Lexer`提供词法分析以及存储结果的功能

在`Lexer`类中按字符读取从文件中读入的源码，由`if-else`分支语句进行类别判断，然后进入对应分支读取完整单词，得到一个`Token`实例，不断重复上述操作直到读到源码末尾

- 对于保留字采用`HashMap`进行查寻，保留字为键，类别码为值，当得到一个保留字单词就查询`HashMap`获得类别码
- 对于注释空白符跳过不识别，即直接读到注释之后的第一个字符
- 对于行号记录，读到`\n`字符行号就加一

本次不需要考虑错误处理的问题

编码后修改

与编码前基本一致，但是考虑到评测系统为`Linux`系统于是对于要跳过的空白符需要注意判断`\r`的情况

语法分析

编码前设计

封装`Parser`类，将`Lexer`类解析得到的所有`Token`以数组形式一次性传入`Parser`进行递归下降解析同时维护一个指向`Token`数组中元素的指针，然后使用自己封装的`peek()`、`retract(int step)`函数在递归下降过程中取出当前指针指向的元素或者将指针回溯。对于每一个非终结符建立相应的类作为递归下降树的非叶子结点，每一个叶子节点对应的为一个代表终结符的`Token`

其中我对于文法的具体分析处理主要如下：

- **表达式**：对于部分与`*Exp`相关的推导规则均存在左递归现象，需要改写文法如下：

```
改写前
-------
改写后

    MulExp -> UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
    ------------------------------------------------------
    MulExp -> UnaryExp { ('*' | '/' | '%') UnaryExp }

    AddExp -> MulExp | AddExp ('+' | '-') MulExp
    --------------------------------------------
    AddExp -> MulExp { ('+' | '-') MulExp }

    RelExp -> AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
    ----------------------------------------------------------
    RelExp -> AddExp { ('<' | '>' | '<=' | '>=') AddExp }

    EqExp -> RelExp | EqExp ('==' | '!=') RelExp
    --------------------------------------------
    EqExp -> RelExp { ('==' | '!=') RelExp }

    LAndExp -> EqExp | LAndExp '&&' EqExp
    -------------------------------------
    LAndExp -> EqExp { '&&' EqExp }

    LOrExp -> LAndExp | LOrExp '||' LAndExp
    ---------------------------------------
    LOrExp -> LAndExp { '||' LAndExp }
```

同时注意到以上列出的`*Exp`中均为`*Exp -> T {operator T}`（其中`T`可以认为是低一级的`*Exp`）的形式，于是可以建立一个基类`MultiExp`去统一这种形式，再使`*Exp`去继承它，`MultiExp`接口如下

```java
public class MultiExp<T> {
    private final String name;
    private final T first;
    private final ArrayList<Token> operators = new ArrayList<>();
    private final ArrayList<T> Ts = new ArrayList<>();
    // ...
}
```

于是可以通过上述方式将需要改写文法的`*Exp`放置于`Multi`软件包内，而其它诸如`UnaryExp`、`PrimaryExp`、`Number`、`LVal`、`FuncRParams`这种更低一级的表达式放置于`Unary`软件包中

- **语句**：由于推导规则中左侧为`Stmt`的规则非常复杂于是我们对于该规则右侧的每一个分支建立`*Stmt`类，即改写文法规则新增加`*Stmt`非终结符，具体如下：

```
    <AssignStmt> -> LVal '=' Exp
    <ExpStmt> -> Exp
    <LoopStmt> -> 'break' | 'continue'
    <ReturnStmt> -> 'return' [Exp]
    <InputStmt> -> <LVal> '=' 'getint' '(' ')'
    <OutputStmt> ->  'printf''('FormatString{','Exp}')'
    <SimpleStmt> -> [ <AssignStmt> | <ExpStmt> | <LoopStmt> |
<ReturnStmt> |
                     <InputStmt> | <OutputStmt> ] ';'
    <IfStmt> -> 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
    <WhileStmt> -> 'while' '(' Cond ')' Stmt
    <BlockStmt> -> Block
    <Stmt> -> <SimpleStmt> | <IfStmt> | <WhileStmt> | <BlockStmt>
```

可以按照带分号与不带分号分为`SimpleStmt`以及`IfStmt`、`WhileStmt`、`BlockStmt`，由于`SimpleStmt`推导规则右侧涉及到较多的新建非终结符，因此可以建立`Simple`软件包统一存储。同时为了方便在`SimpleStmt`中保存，可以创建一个`Simple`接口，由`AssignStmt`、`ExpStmt`...共同实现

- **函数**：注意到其实`MainFuncDef`与`FuncDef`非常相似，于是可以将`MainFuncDef`视作`FuncDef`的特例去继承`FuncDef`类
- **变量声明**：由于不需要输出`Decl`可以直接将`ConstDecl`、`VarDecl`都视为`Decl`用一个布尔变量标识即可

对于上述分析已经我们已经可以消除所有的左递归，但是实际过程中考虑到有部分非终结符的`FIRST`是相同的，于是采取超前扫描的方法，同时定义了`pre`、`nxt`分别表示当前遍历的的数组中的`Token`的前一个和后一个以减少指针回溯，除了在处理`SimpleStmt`的过程中指针可能需要回溯一整个`LVal`的大小，其它时候至多回溯一步即可

对于`SimpleStmt`中的`AssignStmt`、`InputStmt`、`ExpStmt`可能它们的`FIRST`均为`LVal`这时侯需要先记录指针的值然后超前解析一个`LVal`判断其后的`Token`类型即可判断需要进入的分支，然后回溯到之前记录的位置进入对应分支的解析函数再次开始解析

编码后修改

- 编码前并未考虑如何输出的问题，编码时采取对于每个节点重写它的`toString`方法，最后直接调用顶层`CompUnit`的`toString`方法即可
- 由于文法中很多地方都会使用`'[' [*Exp] ']'`这种方式来表示数组变量的维度以及下标，因此，我单独建立了一个`Index`类存储这样的结构，同时完成该类的`toString`方法，具体接口如下：

```java
public class Index {
    private Token lBTk;
    private Token rBTk;
    private Exp exp;
    // ...
}
```

- 编码前并未对于之后的错误处理预留接口，但是在递归下降的编码过程中可以通过判断当前`Token`非常自然的找到相关的错误并预留错误处理的空间

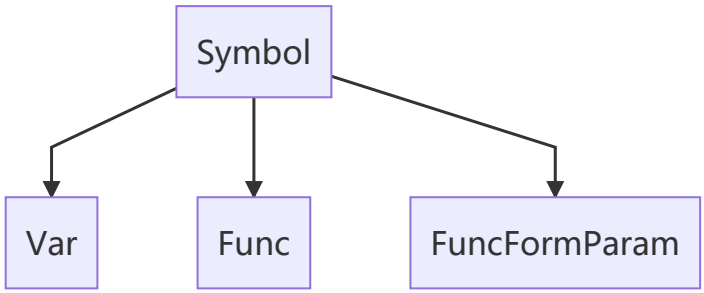
错误处理与符号表建立

编码前设计

- **符号表**：采用树形符号表，遵循一块一表，每次进入一个块时就建立一张符号表，表中存储新建的变量符号、函数符号、函数形参符号（当前为函数块），同时函数符号中存储该函数符号对应的函数块符号表以便查询函数中的变量以及形参

  其中符号一共分为三类，均实现`Symbol`接口：

  - `Var`：所有`int`和`const int`变量
  - `Func`：所有函数定义
  - `FuncFormParam`：函数形参



- **错误处理表**：建立一个`Error`类用于标识特定错误，将错误类型作为`Error`中的一个枚举类，再封装`ErrorTable`类，提供对于`Error`根据错误位置排序、打印、保存的功能
- **遍历语法树**：
  - **符号表建立**：首先维护一个当前符号表`curTable`，初始时即为全局符号表，之后对已经建立好的语法树进行自顶向下遍历，期间算法（不考虑错误处理）
    - 若遍历到`Block`或者`Func`，那么新建一个符号表，将`curTable`设置为新建的符号表，注意如果是`Func`需要将`Func`对应符号加入上一级符号表同时在其中记录新建的符号表引用`curTable`
    - 遍历到`Decl`项或者`FuncFormParm`，则将其加入当前符号表
    - 当从`Block`或者`Func`的遍历返回时将当前符号表设置为其父节点
  - **错误处理**：按错误类型分别讨论：
    - `a`：直接在词法分析中对解析到的`FormatString`判断即可
    - `i`、`j`、`k`：均属于同类错误，在语法分析递归下降的过程中判别，同时注意递归下降过程中不要偷懒，识别每个语法单元时要根据其完整的`First`来，否则可能出错
    - `b`、`c`：对于这两类错误可以通过在遍历语法树时根据`curTable`在该节点中查询是否存在名字或者在该节点以及其祖先节点中查询是否存在该名字，注意对于重复定义的函数其内部也要解析，对于函数形参其作用域为函数体本身，对于局部变量名可以和函数名相同

- `d`、`e`：在解析函数实参时通过`curTable`找到该函数的定义，然后根据其中保存的函数的符号表中存储的函数的形参定义判断
- `f`、`g`：除了维护一个`curTable`再维护一个当前的`curFunc`（`Symbol`），解析到`return`语句时就可以根据`curFunc`中存储的返回值类型对于`f`判断，`g`则是在解析`funcDef`的最后取出其`Block`中的最后一条语句判断是否为`return`
- `h`：直接查找当前节点以及其祖先节点判断变量是否为`Const`类型
- `l`：解析到`print`语句时直接遍历查找`%d`个数判断即可
- `m`：维护一个全局变量`cycLevel`，进入循环块加1，出循环块减1，当遇到`break`或者`continue`语句时判断`cycLevel`是否大于0即可

编码后修改

并未将左值与其定义链接起来同时也并未做常量传播，因此对于错误`e`的判断比预想要难，对于函数调用的实参需要单独重新遍历每一个实参即`Exp`的第一个`PrimaryExp`（不需要检查`Exp`内中参与表达式计算的每一个算子`LVal`、`Number`的类型间是否匹配，所以相当于只用看`Exp`的第一个`PrimaryExp`的维数即可并不需要遍历完），然后查询并计算其维数再在符号表看实参与形参维数是否可以匹配，最复杂的还要在遍历的过程中判断除了维数问题还有没有未定义的问题，此时需要遍历所有的`PrimaryExp`（按我的编码结构是先判断未定义问题再检测类型是否匹配，并且无法在检查类型是否匹配时判断是否已经检测出过未定义的问题）若有就返回，以免对于一行重复报错（`c`、`e`）。预计后续中间代码生成时会继续修改符号表定义

中间代码生成

编码前设计

在错误处理以及建立符号表的同时生成四元式的中间代码，以下为中间代码设计，基本格式为`op operand1 pperand2 res`，由于是类`mips`代码的设计，所以由中间代码的`op`很容易知道中间代码的用途：

```
// ALU
ASSIGN("="), // ASSIGN VAR (EMPTY) VAR
ADD("+"), // ADD VAR VAR VAR
SUB("-"), // SUB VAR VAR VAR
MUL("*"), // MUL VAR VAR VAR
DIV("/"), // DIV VAR VAR VAR
MOD("%"), // MOD VAR VAR VAR
NOT("!"), // NOT VAR (EMPTY) VAR
EQ("=="), // EQ VAR VAR VAR
NE("!="), // NE VAR VAR VAR
GT(">"), // GT VAR VAR VAR
GE(">="), // GE VAR VAR VAR
LT("<"), // LT VAR VAR VAR
LE("<="), // LE VAR VAR VAR

// IO
GET_INT("get_int"), // GET_INT (EMPTY) (EMPTY) [VAR or (EMPTY)]
PRINT_STR("print_str"), // PRINT_STR STR (EMPTY) (EMPTY)
PRINT_INT("print_int"), // PRINT_INT VAR (EMPTY) (EMPTY)

// FUNC
FUNC("func"), // FUNC VAR (EMPTY) (EMPTY)
FUNC_END("func_end"), // FUNC_END VAR (EMPTY) (EMPTY)
PREPARE_CALL("prepare_call"), // PREPARE_CALL VAR (EMPTY) (EMPTY)
CALL("call"), // FUNC VAR (EMPTY) (EMPTY)
```

```
PUSH_PAR_INT("push_para_int"), // PUSH_PAR_INT VAR (EMPTY) (EMPTY)
PUSH_PAR_ADDR("push_para_addr"), // PUSH_PAR_ADDR VAR (EMPTY) (EMPTY)
RETURN("return"), // RETURN [VAR or (EMPTY)] (EMPTY) (EMPTY)

// JUMP
JUMP("jump"), // JUMP (LABEL) (EMPTY) (EMPTY)
NEZ_JUMP("nez_jump"), // NEZ_JUMP (VAR) (EMPTY) (LABEL)
EQZ_JUMP("eqz_jump"), // EQZ_JUMP (VAR) (EMPTY) (LABEL)
LABEL("label"), // LABEL [(AUTO) or (LABEL)] (EMPTY) (EMPTY)

// DEF
DEF_VAL("var_def"), // DEF_VAL [VAR or (EMPTY)] (EMPTY) VAR
DEF_ARR("arr_def"), // DEF_ARR (EMPTY) (EMPTY) VAR
END_DEF_ARR("end_arr_def"), // END_DEF_ARR (EMPTY) (EMPTY) VAR

// BLOCK
BLOCK_BEGIN("block_begin"), // BLOCK_BEGIN VAR (EMPTY) (EMPTY)
BLOCK_END("block_end"), // BLOCK_END VAR (EMPTY) (EMPTY)

NOP("nop"); // NOP (EMPTY) (EMPTY) (EMPTY)
```

其中`(EMPTY)`表示占位符，同时可以令`VAR`为字符串`(AUTO)`表示自动生成一个新的临时变量或者一个新的标签，对于传入的`int`变量`VAR`定义为字符串`name(blockLevel,blockNum)`其中`blockLevel`代表变量在符号表树中的第几层，`blockNum`代表变量在符号表树的第`blockLevel`层的第`blockNum`个，对于数组变量`VAR`定义为字符串`name(blockLevel,blockNum)[index]`，其中`index`为数组展平为一维后的索引下标，最后令函数的返回值对应的`VAR`为`(RT)`

由此我们就可以直接通过字符串索引到变量在符号表中的具体位置，当然也会在中间代码中存储变量对应的`Symbol`，于是提供了两种方法对于中间代码中出现的变量进行查询，以下是一段符合`SYs`语言的代码及其对应生成的中间代码：

- `Sys`：

```
const int b = 9;

int fun(int a[][5]) {
    return a[3][6 * b];
}

int main() {
    int sum, a, b;
    if (a == 0) return 1;
    else return 2;
    sum = a + b
    return 0;
}
```

- 中间代码：

```
DEF_VAL 9 b(0,1)

========FUNC: fun(0,1)========
MUL 6 b(0,1) (T0)
ADD (T0) 15 (T1)
RETURN a(1,1)[(T1)]
========FUNC_END: fun(0,1)========
```

```
========FUNC: main(0,1)========
DEF_VAL sum(1,2)
DEF_VAL a(1,2)
DEF_VAL b(1,2)
DEF_VAL c(1,2)
DEF_VAL d(1,2)
DEF_VAL e(1,2)
SUB c(1,2) d(1,2) (T2)
MUL b(1,2) (T2) (T3)
SUB 0 e(1,2) (T4)
MOD (T3) (T4) (T5)
ADD a(1,2) (T5) (T6)
ASSIGN (T6) sum(1,2)
GE a(1,2) 8 (T7)
EQZ_JUMP (T7) (LABEL3)
LT b(1,2) 0 (T8)
NEZ_JUMP (T8) (LABEL2)
(LABEL3):
EQ c(1,2) 1 (T9)
EQZ_JUMP (T9) (LABEL0)
(LABEL2):
RETURN 1
JUMP (LABEL1)
(LABEL0):
EQ a(1,2) 3 (T10)
EQZ_JUMP (T10) (LABEL4)
RETURN 2
JUMP (LABEL5)
(LABEL4):
(LABEL5):
(LABEL1):
RETURN 0
========FUNC_END: main(0,1)========
```

对于控制流语句，翻译如下：

- `If-else`：

```
EQZ_JUMP Cond label_1
    ...(if的语句块)
JUMP label_2
label_1:
    ...(else的语句块)
label_2:
```

- `while-break-continue`：

```
label:
EQZ_JUMP Cond label__end
    ...(循环语句块)
JUMP label
label_end:
```

参考上述控制流语句，对于条件运算符`||`、`&&`同样采用上述方案，构造一系列跳转标签实现短路运算

中间代码的生成可以与错误处理和符号表建立同时在解析语法树的时候做，解析到对应的句子时候生成对应的中间代码

编码后修改

- 增加`END_ARR_DEF`便于区分赋值与初始化，同时增加`Calc`类可以直接在扫描语法树的过程中计算出`ConstExp`、和全局变量的初值
- 没有对于数组变量的`load`或者`save`，修改方法为遇到表达式右侧的数组变量生成一句`ASSIGN a(xx, xx)[index] (Txx)`中间代码，代表将内存中数组中的值加载到临时变量中，遇到左侧的数组变量那么一定会生成`ASSIGN T(xx) a(xx, xx)[index]`中间代码，代表直接将内存中的地址赋值，其它的中间代码中除了函数压栈可能会与数组有关外都不会出现与数组有关的操作数，于是将数组变量与其它变量区分开来，对于数组变量的操纵都是直接针对内存

目标代码生成

编码前设计

- **内存分配**：
  - 全局变量：全局数组变量全部在`.data`段初始化，全局`int`型变量从`$gp`开始处放置
  - 局部变量：如果寄存器有位置放置在寄存器中其余则均放置在栈中，且进入函数前，就会为每个函数在栈中分配空间，该空间大小为其中间代码所包含的所有临时变量和局部变量的大小且该大小以及它们相对于函数栈针的偏移量会在生成汇编代码前通过递归遍历符号表计算得出，即栈针只在函数调用前后上下移动一次，在函数执行过程中不会移动，只会通过偏移量计算出栈中变量的位置进行寻址
- **寄存器分配**：采取理论课中介绍的临时寄存器分配方法，由于没有做图着色等优化，并没有区分全局寄存器和临时寄存器，如果有空闲寄存器那么直接分配，如果没有那么就需要将当前寄存器的值写回再建立新的寄存器与变量之间的映射，选择换出的变量所在的寄存器采取`OS`课设中学过的页面置换的`CLOCK`策略，按寄存器序号循环扫描。但是有部分特定寄存器不参与分配：
  - `$zero`：0号寄存器，永远为0
  - `$at`：操作系统使用
  - `$v0`：专门用于存储函数返回值以及系统调用传参
  - `$a0`：专门用于`print`，`getInt`函数以及给立即数赋值
  - `$a1`：专门用于计算数组地址，处理移位用
  - `$gp`：全局`int`变量指针
  - `$sp`：栈指针
  - `$ra`：函数返回地址
- **寻址**：封装两个函数`save`、`load`处理所有的存取操作，对于`save`，如果寄存其中有该变量就存入其中否则存入内存中，对于`load`同理，注意对于地址形参，我们要将值存入该地址处而不是存入该地址在栈中的位置，而取出同样是去该地址处取而不是去栈中取出它
- **函数调用**：
  - **寄存器保护**：每次发生函数调用就将寄存器中的值调用`save`函数全部写回，不需要压栈，于是函数调用结束也不需要恢复，之后用到直接从内存取即可
  - **栈针变化**：再调用函数中将栈指针下移被调用函数所需空间大小，调用结束由调用函数恢复
  - **参数压栈**：此时栈指针还未移动但是由于我们已经在符号表中存储了参数的相对地址于是可以直接计算出参数地址然后压栈
  - **返回值与返回地址**：返回地址由被调用函数压栈，返回时取出，返回值不需要压栈直接存储在`$v0`中，函数返回后执行一条将返回值赋给临时变量的指令
- **四元式翻译**：对于四元式中的出现的不同的`op`，采用不同的翻译方法：
  - **基本运算**：

        - 将三个操作数中的非结果中间变量转化为变量所在的寄存器(若未分配寄存器则通过`lw`加载到临时寄存器中)，结果操作数则直接分配寄存器或找到它在的寄存器。
        - 将中间代码翻译为对应的mips指令。
    - `IO`：`getint`直接执行系统调用`li $v0, 5 - syscall`，结果赋值给变量。
        - `output`若为`STRCON`，先获取位于`.data`段中的地址，再进行系统调; 若为一个`int`，则将该变量存入`$a0`后执行系统调用。
    - **函数调用相关**：

      维护一个调用函数栈，栈顶存储当前调用函数及其实参所在变量
        - 在`PREPARE_CALL`时，记录当前调用函数，压入调用函数栈（为了处理多重调用，实际是编码过程中改的）。
        - 在`PUSH_*`时计算参数值或数组的首地，加入调用函数栈栈顶
        - `CALL`：将栈顶弹出，将其中参数压栈然后调用对应函数
    - **跳转相关**：直接翻译即可，中间代码这部分生成十分完善
    - **赋值以及初始化**：直接调用`save`函数即可，注意全局数组变量的初始化以及字符串的初始化单做如下：

```
// 数组
.data
arr_name: .space SPACE
 ...

// or

.data
arr_name: .word val1 val2 ... valn
...

// 字符串
.word
str_num: .asczii STRCON
```

编码后修改

编码前存在许多情况未考虑，具体如下：

- 函数参数压栈，如果实参是一个`int`变量，和函数调用的返回值，那么按照之前的设计，将会先将`int`变量压栈，然后去计算函数调用返回值，再将函数调用的返回值压栈，而实际这样会使做为实参的函数调用在发生调用时覆盖之前压栈的`int`变量，最后传入错误的参数，所以一定要将所有实参全部都计算出来再一起压栈，比如如下情况：

```
PREPARE_CALL fun0(0,1)
PUSH_PAR_INT a(1,2)
PREPARE_CALL fun1(0,2)
CALL fun1(0,2)
ASSIGN (RT) (T1)
PUSH_PAR_INT (T1)
CALL fun0(0,1)
```

  对应于`Sys`代码就是：

```
void fun0() {
    // ...
}

int fun1() {
    //...
    return 1;
}

// ...
int main() {
    int a = 9;
    // ...
    fun0(a, fun1());
    // ...
}
```

可以看到中间代码先将`a`压栈了，然后去计算`fun1()`的返回值，由于是函数调用嵌套，同时我对于函数调用的处理是先将参数按符号表计算出的位置压栈，但是栈针不移动，全部压完栈再移动栈针，故对于嵌套的函数调用`fun1()`的栈空间就会覆盖掉压入栈的`a`，所以需要先全部计算出函数的实参值后一并压栈

- 对于全局数组地址作为形参时，由于`save`函数对于形参地址是直接写入地址处，于是当该地址在寄存器中且需要回写时就会将地址写入地址处（赋值时不会有问题，因为会将值直接写入地址处），采取的方法是永远不回写作为形参的地址，毕竟地址不会改变，这样同时还会节省一些访存类指令
- 分配寄存器时要考虑寄存器的冲突，需要考虑需要两个寄存器作为操作数的`alu`类指令，当第一个寄存器刚分配，用可能该寄存器就又被分配给了第二个操作数，要确保这两个寄存器在不是同一个变量的时候不同
- 目前还没划分基本块，但是注意由于控制流语句以及函数调用会导致函数执行流混乱且无法预测，于是需要在进入一个基本块前将所有寄存器回写防止出现函数在运行过程中明明回写了某值但是由于是循环，于是接着再次执行的时候会再次执行`sw`指令回写一个错误的值（区分生成汇编代码过程和函数设计执行过程的关系），但是只要再进入基本块前就将所有寄存器回写就可以避免问题，注意由于做了短路求值，于是每一次条件跳转以及函数调用前都需要将寄存器的值回写，编码前没有考虑过这种情况，同时为了解决这种情况还生成了一些空跳指令，比如如下情况：

```
label:  // 1
EQZ_JUMP Cond label__end
    ...(循环语句块)
JUMP label
label_end:
```

注意上面中间代码在`1`这个位置，由于`1`之前的代码可以进入`1`，而第四行的跳转也可以进入`1`，因此需要保证从两个位置处进入`1`的寄存器分配状态是一致的，即必须在进入`1`前清空寄存器分配的映射同时回写，没有划分基本块，否则可以出基本块回写，进入基本块解除寄存器映射，所以一个折中的解决方法就是遇到跳转指令就回写同时解除映射，故需要在`1`前生成一个跳转`JUMP label`，保证寄存器状态在进入`1`前已经为空

- 使用`*u`指令，使用不带`u`的`alu`类计算指令当发生溢出时会直接进入异常处理程序发生异常

代码优化

运算强度削弱

乘除

- 首先对于乘法，可以将赋值操作转换为移位操作，例如将`mul $s1, $s2, num`可以翻译为：

```
sll $s1, $s2, x0
sll $a1, $s2, x1
addu $s1, $a1, $s1
sll $a1, $s2, x3
addu $s1, $a1, $s1
# ...
nge $s1, $s1 # if num < 0
```

注意$|num| = 2^{x_i} + 2^{x_{i-1}} + \ldots + 2^{x_0}$，其中$x_i > x_{i-1} > \ldots > x_0 \geq 0$，该拆分可以提前计算出来，同时需要特判$num = 0$的情况，我们可以计算出这种转换所用的指令的`cycle`数，如果`cycle`数大于等于4，那么应该直接翻译为一条乘法指令，不然就是负优化

- 对于除法，可以参考下面的算法：

procedure CHOOSE_MULTIPLIER(**uword** d, **int** $prec$);
Cmt. d – Constant divisor to invert. $1 \leq d < 2^N$.
Cmt. $prec$ – Number of bits of precision needed, $1 \leq prec \leq N$.
Cmt. Finds m, sh_{post}, ℓ such that:
Cmt. $2^{\ell-1} < d \leq 2^{\ell}$.
Cmt. $0 \leq sh_{\text{post}} \leq \ell$. If $sh_{\text{post}} > 0$, then $N + sh_{\text{post}} \leq \ell + prec$.
Cmt. $2^{N+sh_{\text{post}}} < m * d \leq 2^{N+sh_{\text{post}}} * (1 + 2^{-prec})$.
Cmt. Corollary. If $d \leq 2^{prec}$, then $m < 2^{N+sh_{\text{post}}} * (1 + 2^{1-\ell})/d \leq 2^{N+sh_{\text{post}}-\ell+1}$.
Cmt. Hence m fits in $\max(prec, N-\ell)+1$ bits (unsigned).
Cmt.
int $\ell = \lceil \log_2 d \rceil$, $sh_{\text{post}} = \ell$;
udword $m_{\text{low}} = \lfloor 2^{N+\ell}/d \rfloor$, $m_{\text{high}} = \lfloor (2^{N+\ell} + 2^{N+\ell-prec})/d \rfloor$;
Cmt. To avoid numerator overflow, compute m_{low} as $2^N + (m_{\text{low}} - 2^N)$.
Cmt. Likewise for m_{high}. Compare m' in Figure 4.1.
Invariant. $m_{\text{low}} = \lfloor 2^{N+sh_{\text{post}}}/d \rfloor < m_{\text{high}} = \lfloor 2^{N+sh_{\text{post}}} * (1 + 2^{-prec})/d \rfloor$.
while $\lfloor m_{\text{low}}/2 \rfloor < \lfloor m_{\text{high}}/2 \rfloor$ **and** $sh_{\text{post}} > 0$ **do**
    $m_{\text{low}} = \lfloor m_{\text{low}}/2 \rfloor$; $m_{\text{high}} = \lfloor m_{\text{high}}/2 \rfloor$; $sh_{\text{post}} = sh_{\text{post}} - 1$;
end while; /* Reduce to lowest terms. */
return (m_{high}, sh_{post}, ℓ); /* Three outputs. */
end CHOOSE_MULTIPLIER;

Inputs: **sword** d and n, with d constant and $d \neq 0$.
udword m;
int ℓ, sh_{post};
$(m, sh_{\text{post}}, \ell) = \text{CHOOSE\_MULTIPLIER}(|d|, N-1)$;
if $|d| = 1$ **then**
  Issue $q = d$;
else if $|d| = 2^{\ell}$ **then**
  Issue $q = \text{SRA}(n + \text{SRL}(\text{SRA}(n, \ell-1), N-\ell), \ell)$;
else if $m < 2^{N-1}$ **then**
  Issue $q = \text{SRA}(\text{MULSH}(m, n), sh_{\text{post}}) - \text{XSIGN}(n)$;
else
  Issue $q = \text{SRA}(n + \text{MULSH}(m - 2^N, n), sh_{\text{post}})$
    $- \text{XSIGN}(n)$;
  Cmt. Caution — $m - 2^N$ is negative.
end if

if $d < 0$ **then**
  Issue $q = -q$;
end if

当然如果对于立即数属于$\{1, -1\}$的情况特判会更快，上述算法的主要思想是将$\frac{n}{d}$转化为$\frac{n \times m}{2^{N+l}}$，即乘法加移位运算，当然还需要考虑$m$的溢出、$n$的正负性的问题

地址计算

对于访存一个数组的元素，比如`a[5]`，我们可以用移位运算代替乘法去计算偏移量`20`

常量合并

- 对于类型为`const`或者位于全局的变量的初始值都可以在中间代码生成的过程中计算出来，于是可以写一个计算类，通过`Java`的异常机制，计算不出来那么就生成正常的中间代码
- 对于`stmt`中的一些计算语句，或者比较语句我们也可以将其算出来，但是如果遇到左值，可以用初值代替其中为`const`的变量，而非`const`变量则不行，防止变量除了初始化之外还在其它地方被赋值

对于如下代码：

```
const int a[10] = {1,2,3,4,5,6,7,8,9,10};

int main() {
    int b = a[0] + 5;
    printf("%d", a[7] + 10);
    printf("%d", b);
    return 0;
}
```

可以翻译成如下中间代码：

```
// ...
======FUNC: main(0,1)======
DEF_VAL 6 b(1,1)
PRINT_INT 18
PRINT_INT b(1,1)
RETURN 0
======FUNC_END: main(0,1)======
```

do-while

对于`Sys`语言中的`while(cond) {...}`可以翻译成`if (cond) {do{...} while (cond)}`，即如下：

```
# while
begin:
beqz cond end
...
j begin
end:

# do - while
beqz cond end
begin:
...
bnez cond begin
end:
```

对于一开始不满足情况那么二者均为跳转一次，但是如果多次进入循环体的内部，易得上面的`while`会执行$2n-1$次跳转，而下面的`do-while`会执行n次跳转，且不会有其他副作用，但是注意的是要翻译两次`Cond`，一次是满足要跳转`begin`，一次是不满足要跳转到`end`，这里有个问题就是如果你是解析两次`Cond`，那么如果遇到错误，你会报两次错，所以建议用一个行号为键值的`HashMap`来存储错误，这样就不会报两次同一行的错误了

数据流分析

到达定义分析

活跃变量分析

死代码删除

常量折叠

复写传播

临时变量寄存器分配

窥孔优化