

# 编译器设计文档

## 参考编译器介绍

[编译器示例代码-sysy-compiler](#)

## 总体结构

整体分为前端（词法、语法、AST）、中端（中间代码表示 IR、符号表）、后端（目标代码生成三部分，另外把错误处理与工具模块拆分出来，主入口为 `Compiler.java`。

## 接口设计

### 前端

```
1 public class FrontEnd {
2     private static Lexer lexer;
3     private static Parser parser;
4
5     public static void SetInput() throws IOException {
6         lexer = new Lexer(IOhandler.GetInput());
7         parser = new Parser();
8     }
9
10    //词法分析器生成Token流
11    public static void GenerateTokenList() throws IOException {
12        lexer.GenerateTokenList();
13    }
14
15    //语法分析器生成语法树
16    public static void GenerateAstTree() {
17        parser.SetTokenStream(GetTokenStream());
18        parser.GenerateAstTree();
19    }
20
21    //取Tokens
22    public static ArrayList<Token> GetTokenList() {
23        return lexer.GetTokenList();
24    }
25
26    //获得Token流
27    private static TokenStream GetTokenStream() {
28        return new TokenStream(lexer.GetTokenList());
29    }
30
31    //取语法树
32    public static CompUnit GetAstTree() {
33        return parser.GetAstTree();
34    }
35 }
36
```

## 中端

```
1 public class MidEnd {
2     private static CompUnit rootNode;
3     private static IrModule irModule;
4
5     //创建符号表
6     public static void GeneratesymbolTable() {
7         SymbolManger.Init();
8         rootNode = FrontEnd.GetAstTree();
9         rootNode.Visit();
10        SymbolManger.GoBackToRootSymbolTable();
11    }
12
13    //创建中间代码
14    public static void GenerateIr() {
15        irModule = new IrModule();
16        IrBuilder.SetCurrentModule(irModule);
17        Visitor visitor = new Visitor(rootNode);
18        visitor.Visit();
19        IrBuilder.Check();
20    }
21
22    //获得符号表
23    public static SymbolTable GetSymbolTable() {
24        return SymbolManger.GetSymbolTable();
25    }
26
27    //获得中间表示
28    public static IrModule GetIrModule() {
29        return irModule;
30    }
31 }
32
```

## 后端

```
1 public class BackEnd {
2     private static IrModule midEndModule;
3     private static MipsModule backEndModule;
4
5     //生成Mips代码
6     public static void GenerateMips() {
7         backEndModule = new MipsModule();
8         MipsBuilder.SetBackEndModule(backEndModule);
9
10        midEndModule = MidEnd.GetIrModule();
11        midEndModule.toMips();
12        // 进行窥孔优化
13        if (Setting.FINE_TUNING) {
14            PeepHole peepHole = new PeepHole();
15            peepHole.Peep();
16        }
17    }
18 }
```

```

18
19     //获得Mips表示
20     public static MipsModule GetMipsModule() {
21         return backEndModule;
22     }
23 }
24

```

## 详细分析

### 前端

- `lexer`：词法分析，负责生成 `token` 流。
- `parser`：语法分析将 `tokens` 转为语法树。
- `ast`：完整的抽象语法树节点（`CompUnit`、`FuncDef`、`Stmt`、`Exp`、`Decl` 等），对应 `parser` 所使用的类。
- `FrontEnd.java`：前端的统一接口，输出 `AST`、`Tokens` 等供后续阶段使用。

### 中端

- `symbol`
  - `SymbolTable` / `SymbolManager` / 各类 `Symbol`：追踪作用域、类型与符号信息，建立符号表
- `visit`
  - 实现对 `AST` 的遍历与语义分析、类型检查、符号绑定等。
- `midend`
  - `MidEnd.java`：中端统一接口，负责把 `AST` 转换为中间表示（`IR`），并调用 `IR` 层优化。
  - `llvm` 包：`IrBuilder`、`IrModule`、`IrNode`、等，表示 `LLVM` 风格的中间表示与构建器。
- `optimize`
  - 多个优化和分析通道（`ActiveAnalysis`、`Lvn`、`MemToReg`、`InsertPhi`、`RemoveDeadCode/Block`、`RegisterAllocator` 等），可组合用于中端 `IR` 优化与寄存器分配。

### 后端

- `backend`
  - `BackEnd.java`、`PeepHole.java`：后端调度与局部 `peephole` 优化。
  - `mips`：目标为 `MIPS` 的代码生成器（`MipsBuilder`、`MipsModule`、`Register`）以及细分的指令/汇编构造（`assembly` 下的多个类），负责把 `IR` 翻译为 `MIPS` 汇编。
- `error`（错误处理）与 `utils`（工具）
  - `ErrorRecorder/Error/ErrorType`：集中管理编译过程中的错误与诊断信息。
  - `utils` 提供 `IO`、调试、配置、复杂度处理等功能。

# 自己编译器设计

大致分为前、中、后三部分，按照词法分析、语法分析、语义分析、中间代码生成、目标代码生成、代码优化的顺序来写。

词法分析器负责生成 `Token` 流，语法分析器负责生成语法树。语义分析器解析生成的语法树，此部分参考上述编译器设计 `Visitor` 类分别对各语法成分进行分析。中间代码生成可能需要再设计另外的 `Visitor` 类结合 `SymbolTable` 来进一步分析。

目标代码生成器根据中间代码来生成，之后考虑代码优化部分。

## 文件组织(暂定)

```
1  |─frontend
2  |   └─Parser
3  |       └─Decl
4  |       └─Exp
5  |       └─FuncDef
6  |       └─MainFuncDef
7  |       └─Stmt
8  |       └─Token
9  |       └─Tree
10 └─midend
    └─Symbol
    └─Visit
        └─Decl
        └─Exp
        └─Func
        └─MainFuncDef
        └─Stmt
```

## 词法分析设计

### 1. `token` 类设计包含

1. `token` 枚举类型如 `IDENFR`、`INTCON`、`CONSTTK` 等；`TokenType type`
2. 当前 `token` 对应字符串表示；`String lexeme`
3. 当前 `token` 所在行号；
4. 是否已经被输出；`boolean isPrinted`

### 2. 设计 `Lexer` 类包含

- 源码文件流 `FileInputStream file`
- 生成的tokens `ArrayList<Token> tokens`
- 产生的错误 `ArrayList<Error> errors`
- 当前读到的字符 `currentChar`，当前读到的行数 `currentLine`

3. 构造 `Lexer` 类时初始化源码文件流，并读取第一个字符。

4. 设计分析函数，可由 `Compiler` 调用。

5. 分析函数通过调用 `getToken()` 函数每处理一个 `token` 就将其加入 `tokens` 数组中。

6. `getToken()` 函数声明一个 `StringBuilder lexeme`，根据自动机判断当前字符应该什么归于什么类型的处理函数。

1. 首字符为英文字符或下划线 `_` , 进入处理标识符或关键字函数;
  2. 首字符为数字, 进入整数常量处理函数;
  3. 首字符为引号 `"` , 进入字符串常量处理函数;
  4. 首字符为 `/` , 进入注释或除号处理函数;
  5. 首字符为 `+` `-` `*` `%` `;` `,` `(` `)` `[` `]` `{` `}` , 进入单字符运算符或分隔符处理函数;
  6. 首字符为 `=` `>` `<` `!` , 进入双字符运算符处理函数;
  7. 首字符为 `&` , 进入与符号处理函数;
  8. 首字符为 `|` , 进入或符号处理函数;
  9. EOF 返回 `null` ;
7. 处理函数返回当前 `token` 。
8. 处理函数中出现的错误加入 `errors` 数组中, 方便后续处理。

## 语法分析

### 1. 设计语法树:

1. 一个语法成分对应一个结点, 他们有公共的父类 `Node` ;
  - `Node` 类包含枚举类 `GrammarType type` 表示当前语法成分的类型, 如 `Exp`、`IntConst` 等;
  - `Node` 类包含 `Token token` , 表示当前**终结符**对应的 `Token` ;
  - `Node` 类包含 `ArrayList<Token> tokens` , 表示 `Lexer` 分析得到的 `token` 流;
  - `Node` 类包含 `int tokenIndex` , 表示当前读入 `token` 流索引, 进入/结束某一成分分析时 `tokens[tokenIndex]` 都应处于未处理状态;
  - `Node` 类包含 `Node parent` 和 `ArrayList<Node> children` 分别表示父节点和子节点;
  - `Node` 类包含 `final String filename` 和 `final String ErrorFilename` 分别表示正确输出文件和错误输出文件;
  - `Node` 类含方法 `Token peekToken(int offset)` , 用于获取相对于索引 `tokenIndex` 偏移 `offset` 的值, 可以预读或回读;
  - `Node` 类包含方法 `void printToError(Error error)` , 输出对应错误时将其添加到 `GlobalError` 中;
2. `Node` 子类构造函数需 `GrammarType type`、`int tokenIndex`、`ArrayList<Token> tokens` , 终结符需设置 `Token token` ;
3. 子类含分析方法, 按照文法进行依次分析, 调用对应成分分析方法前需:
  1. 创建对象;
  2. 添加为当前成分的子节点;
  3. 调用 `parser()` ;
4. 分析方法末尾, 先输出当前成分类型 (终结符先输出当前 `token` ) , 再更新父节点的 `tokenIndex` (终结符保留当前`tokenIndex`指向对应 `token` , 直接更新父节点 `index+1` ) ;

### 2. 设计 `Parser` 类:

- 包含 `ArrayList<Token> tokens` 表示传入的 `token` 流;
- 包含 `ComUnit root` 表示初始根节点;

3. 对于类似 `AddExp → MulExp | AddExp ('+' | '-') MulExp` 的分析，由于存在左递归，将其转化成 `MulExp { ('+' | '-') MulExp }`，但是位于运算符左边的成分实际上是 `AddExp` 而不是 `MulExp`，所以如果存在运算符，应在左操作数调用 `MulExp` 的分析方法后输出成分 `<AddExp>`，（仅用于输出，语法树的构造无需额外子节点 `AddExp`）。

## 四、语义分析

### 1. 符号表设计：

#### 1. `SymbolType` 类设计：

```
1 public enum SymbolType {
2     CONST_INT("ConstInt"),
3     CONST_INT_ARRAY("ConstIntArray"),
4     STATIC_INT("StaticInt"),
5     INT("Int"),
6     INT_ARRAY("IntArray"),
7     STATIC_INT_ARRAY("StaticIntArray"),
8     VOID_FUNC("VoidFunc"),
9     INT_FUNC("IntFunc"),
10    ARRAY("Array"),
11    NOT_ARRAY("NotArray"),
12    NOT_EXIST("NotExist");
13    private final String typeName;
14
15    SymbolType(String typeName) {
16        this.typeName = typeName;
17    }
18
19    public String getTypeName() {
20        return typeName;
21    }
22 }
```

#### 2. `Symbol` 类设计：

1. `String symbolName` 表示符号名称；
2. `SymbolType symbolType` 表示符号类型；
3. `int lineNumber` 表示当前符号所在行；
4. `ArrayList<Symbol> params` 若当前符号为函数名，额外标记参数符号列表；
5. `ArrayList<Integer> initValues` 表示当前符号初始赋值；
6. `int size` 若当前符号为数组记录数组大小；
7. `IrValue irValue`，中间代码生成时使用，表示符号对应生成的 `IrValue`；

#### 3. `SymbolTable` 类设计：

1. `int depth` 表示当前作用域序号（不是深度，只表示创建时顺序）
2. `ArrayList<Symbol> symbolList` 表示当前作用域中符号；
3. `Hashtable<String, Symbol> symbolTable` 表示当前作用域中符号方便查找；
4. `SymbolTable fatherTable` 表示父作用域对应符号表；
5. `ArrayList<SymbolTable> sonTables` 表示子作用域所含符号表；

6. `int index` 遍历所有符号表时，表示遍历到子作用域的子符号表索引；
4. 设计一个 `GlobalSymbolTable` 类作为全局作用域的符号表；
  1. `SymbolTable globalSymbolTable` 表示全局作用域的符号表；
  2. 设计字段 `SymbolTable localSymbolTable` 表示当前处理的符号表；
  3. `int scopeDepth=1` 表示作用域序号（不代表深度）；
5. 额外设计 `OutSymbolTable` 类，表示全局作用域之外的符号表，方便添加库函数；

## 2. 主 `visitor` 类设计：

1. `ComUnit comUnit` 表示要分析的程序单元；
2. `visit()` 方法遍历语法树；
  1. 首先通过 `OutSymbolTable.addSymbol` 添加库函数定义；

```
1 OutSymbolTable.addSymbol(new Symbol("getint",
  SymbolType.VOID_FUNC,0,
2     new IrFunction(ValueType.FUNCTION, IrType.INT32,
    "@getint")));
```

2. 依次遍历所有声明，函数定义，最后处理主函数；

## 3. `VisitorDecl` 类设计：

1. `Decl → ConstDecl | VarDecl` 根据语法规则进行遍历；
2. 声明调用 `GlobalSymbolTable.addVarDef()` 或 `GlobalSymbolTable.addConstDef()` 添加进当前符号表；
  1. `GlobalSymbolTable.add...Def()` 方法
  2. 首先获取符号名，符号类型，当前行数创建 `Symbol` 对象；
  3. 然后依据符号类型判断是否设置数组长度；
  4. 为符号添加初始赋值（无初始值则保持空列表）；
  5. 添加符号进当前符号表；

```
1 public static void addVarDef(VarDef varDef, boolean isStatic) {
2     String Ident = varDef.GetIdent();
3     SymbolType symbolType = varDef.GetSymbolType();
4     ...//静态变量类型转换
5     int line = varDef.GetLineNumber();
6     Symbol symbol = new Symbol(Ident, symbolType, line);
7     if (symbolIsArray(symbolType)) {
8         symbol.setSize(varDef.GetArraySize());
9     }
10    symbol.setInitValues(varDef.GetInitValues());//为符号添加初始
    赋值（无初始值则保持空列表）；
11    localSymbolTable.AddSymbol(symbol);
12 }
```

## 3. `VisitorFuncDef` 类设计：

1. 获取返回值类型标记接下来要处理的块为最外层 `Block`；
2. 作用域加一，访问函数参数，回到上一级作用域，再添加符号到符号表

3. 进入下一级作用域，访问 Block，访问结束回到上一级作用域；
4. main 函数处理类似，但是不添加符号到符号表；
4. VisitorStmt 类设计：
  1. stmt.isBlock() || stmt.isIfStmt() || stmt.HasElseStmt() || stmt.isForStmt() 分别调用对应的 visitor 类；
  2. stmt.isLVal()，调用 visitLVal() 和 visitExp()；设计 VisitorLVal 类，可能出现的错误：
    1. LVal 为常量时，不能对其修改； ErrorType.h
    2. 变量未定义； ErrorType.c
  3. stmt.isExp() 调用 visitExp()；设计 visitExp()，可能出现的错误：
    1. 函数未定义； ErrorType.c；
    2. 函数调用时无实参，但函数定义有形参； ErrorType.d
  4. stmt.isReturn() 调用 visitReturn()；可能出现的错误：有返回值但不需要 ErrorType.f；
  5. stmt.isBreakContinue() 调用 visitJump()；
    1. 在语法分析阶段设置变量 boolean isForBody
    2. 可能出现的错误：非循环块使用 break 和 continue； ErrorType.m；
  6. stmt.isPrintf() 调用 visitPrint()；可能出现的错误：%d 数量不匹配； ErrorType.l

## 五、中间代码生成

采用 LLVM 作为中间代码

### IrType 设计

IrType 用来描述 IrValue 的类型；

与 enum 相似，通过 typeName 字段来标识不同类型；

```

1 public static final IrType MODULE = new IrType("module");
2 public static final IrType FUNCTION = new IrType("function");
3 public static final IrType POINTER = new IrType("pointer");
4 public static final IrType BASICBLOCK = new IrType("basicblock");
5 public static final IrType VOID = new IrType("void");
6 public static final IrType INT1 = new IrType("i1");
7 public static final IrType INT8 = new IrType("i8");
8 public static final IrType INT32 = new IrType("i32");
9 public static final IrType ARRAY = new IrType("array");//array元素默认int32
10 public static final IrType STRING = new IrType("string");//string元素默认int8
11 private final String typeName;
```

额外添加 int arraySize 来表示数组类型的数组长度；

对于指针类型 IrType("pointer")，设计子类 IrPointer，包含 public IrType targetType；表示指针指向的类型。如 IrGlobalValue 的类型为 new IrPointer(initial.irType)；

注意：对于 `array` 和 `string` 类型需要创建新的 `IrType` 类而不是直接使用 `IrType.ARRAY` 来赋值，因为不同的数组需要赋 `arraySize` 的值不同需要新的对象

为方便处理循环体，额外设计 `IrLoop` 类，包含四个基本块

```
1 private IrBasicBlock condBlock;
2 private IrBasicBlock bodyBlock;
3 private IrBasicBlock stepBlock;
4 private IrBasicBlock afterBlock;
```

## IrValue设计

我对 `IrValue` 的理解：对于源程序有语法树，各语法成分有公共父类 `Node`，而 `IrValue` 有点像是对于 LLVM 程序的 `Node` 类；

LLVM 由 `IrValue` 组成，生成 LLVM IR 的过程就是构造各种 `IrValue` 的过程；

```
1 public final ValueType valueType; // 枚举类（仅用来标识）
2 public final IrType irType; // 对于变量/常量表示其类型，对于函数/指令表示返回值类型
3 public final String irName; // LLVM中可能会使用的命名
4 public final ArrayList<IrUse> useList; // 记录当前IrValue所包含的使用关系（只有被使用者才有内容）
```

`IrUse` 类表示一种使用的关系，即记录了谁使用了谁；

`IrUser` 类继承 `IrValue` 类，表示使用者，包含 `ArrayList<IrValue> useValues` 表示其使用的值，这也是为什么使用者的 `useList` 没有添加内容。

## Instruction类

所有指令的父类，继承 `IrValue` 类

`InstructionType instrType` 表示指令的类型，`IrBasicBlock inBasicBlock` 表示当前指令所在的基本块；

```
1 // 添加指令到当前基本块
2 public static void addInstr(Instruction instr) {
3     currentBasicBlock.addInstruction(instr);
4     instr.setParentBasicBlock(currentBasicBlock);
5 }
```

指令构造函数中并没有设计自动添加基本块，所以需在 `IrBuilder` 中显式地为指令添加基本块。

## IrBasicBlock类

基本块类，同样继承 `IrValue` 类

`ArrayList<Instruction> instructions`；按顺序记录基本块所含指令；

`IrFunction function`；表示此基本块所属函数（此处函数有点类似于作用域的含义）；

## IrModule类

与 `comUnit` 类似，表示 LLVM 的整体。

包含：

```
1 ArrayList<String> declares; // 定义
2 ArrayList<IrFunction> functions; // 函数
3 ArrayList<IrGlobalValue> globals; // 全局变量/静态变量
4 HashMap<String, IrConstString> stringIrConstStringHashMap; // 字符串常量
```

## IrBuilder设计

`IrModule irModule` 表示输出的 LLVM IR 单元；

`IrFunction currentFunction` 表示当前正在处理的函数；

`IrBasicBlock currentBasicBlock` 表示当前正在处理的基本块；

`Stack<IrLoop> loopStack` 用来记录循环体的嵌套；

`int basicBlockNum`、`int globalVarNum`、`int stringConstNum` 分别记录基本块、全局/静态变量、字符串常量的数目方便命名；

`HashMap<IrFunction, Integer> localVarNum` 记录每个函数内的局部变量数目；

所有 `Instruction` 指令类型的对象都通过 `IrBuilder` 中 `GetNew...()` 方法来创建；

## 生成过程

与语义分析过程相似，依次遍历所有声明，函数定义，最后处理主函数。与语义分析添加符号到符号表对应的是，在中间代码生成阶段为 `Symbol` 类中 `IrValue IrValue` 字段赋予对应的对象。最后重写各 `IrValue` 的 `toString()` 方法并依次输出；给出 `IrModule` 的 `toString()` 方法：

```
1 public String toString() {
2     StringBuilder sb = new StringBuilder();
3     //输出声明
4     for (String declare : declares) {
5         sb.append(declare).append("\n");
6     }
7     //输出字符串常量
8     List<Map.Entry<String, IrConstString>> stringEntries
9         = new ArrayList<>(this.stringIrConstStringHashMap.entrySet());
10    for (Map.Entry<String, IrConstString> entry : stringEntries) {
11        sb.append(entry.getValue()).append("\n");
12    }
13    //输出全局变量
14    for (IrGlobalValue global : globals) {
15        sb.append(global).append("\n");
16    }
17    //输出函数
18    for (IrFunction function : functions) {
19        sb.append(function).append("\n");
20    }
21 }
```

```

20     }
21     return sb.toString();
22 }

```

## 目标代码生成 (MIPS)

参考中间代码生成的输出过程，将MIPS代码看作一个MipsModule类，生成目标代码的过程就是构建MipsModule类的过程；

### MipsModule类设计

MIPS分为.data和.text两段内容，分别用String列表存储；

```

1 // 存放 .data 段的内容 (全局变量)
2 private List<String> dataSection = new ArrayList<>();
3 // 存放 .text 段的内容 (指令)
4 private List<String> textSection = new ArrayList<>();

```

覆写toString()方法：先输出.data段内容，再输出.text段内容；

```

1 public String toString() {
2     StringBuilder sb = new StringBuilder();
3     sb.append(".data\n");
4     for (String s : dataSection) sb.append(s).append("\n");
5     sb.append("\n.text\n");
6     if (Backend.getOptimize()) {
7         List<String> optimizedText =
MipsOptimizer.optimize(this.textSection);
8         for (String s : optimizedText) sb.append(s).append("\n");
9     } else {
10        for (String s : textSection) sb.append(s).append("\n");
11    }
12    return sb.toString();
13 }

```

### MipsBuilder类

用来构建MipsModule类，整体架构为：

1. 处理字符串常量；
2. 处理全局变量；
3. 处理函数；
4. 手动生成库函数；

```

1 private static MipsModule mips = new MipsModule(); //存储生成的MipsModule类

```

记录为IrValue分配的栈空间

```

1 private static HashMap<IrValue, Integer> offsetMap = new HashMap<>(); // 记录为
  IrValue分配的栈空间
2 private static int currentFunctionStackSize = 0; // currentFunctionStackSize
  记录当前已用的字节数（正数）
3 // 给一个 value 分配栈空间
4 private static void allocateStack(IrValue value, int sizeBytes) {
5     // 栈向下增长，所以偏移量是负数
6     currentFunctionStackSize += sizeBytes; // 增加已用字节数
7     offsetMap.put(value, -currentFunctionStackSize);
8 }

```

记录数组实体相对于\$fp的偏移量

```

1 private static HashMap<AllocateInstruction, Integer> allocaArrayOffsets = new
  HashMap<>();

```

寄存器分配优化时使用

```

1 private static RegisterAllocator registerAllocator = new RegisterAllocator();

```

## 处理字符串常量

读取 irModule 中所有 IrConstString 类;

LLVM中字符串命名 @s\_number; number表示数字标号;

IrConstString类输出内容为 @s\_number = constant [number x i8] c"xxxxxxxx\00"

1. 将字符串开头的 @ 符号去除;

```

1 String label = irConstString.irName.substring(1);

```

2. 取 c" 和最后一个 " 之间的内容;

```

1 String rawFull = irConstString.toString();
2 int start = rawFull.indexOf("c\\") + 2;
3 int end = rawFull.lastIndexOf("\\");
4 String content = rawFull.substring(start, end);

```

3. 将LLVM IR的 Hex 转义符替换为 MIPS 的转义符

```

1 // 换行符: \0A -> \n
2 content = content.replace("\\0A", "\\n");
3
4 // 空字符: \00 -> 空字符串 (.asciiz 会自动补0, 所以这里删掉)
5 content = content.replace("\\00", "");
6
7 // 双引号: \22 -> \"
8 content = content.replace("\\22", "\\\"");
9
10 // 反斜杠: \5C -> \\
11 content = content.replace("\\5C", "\\");

```

4. 拼接汇编指令, 给content加上双引号加入 MipsModule;

```

1 String asm = String.format("%s: .asciiz \"%s\"", label, content);
2 mips.addGlobal(asm);

```

## 处理全局变量

读取 irModule 中所有 IrGlobalValue 类;

LLVM中全局变量命名 @g\_number; number表示数字标号;

IrGlobalValue 输出内容为:

@g\_number = dso\_local global 类型 initValue; 或静态变量 @g\_0 = internal global 类型 initValue

1. 获取标签名去除@

```

1 String label = global.irName.substring(1);

```

2. 获取初始值, 指定按4字节对齐;

```

1 mips.addGlobal(".align 2");

```

3. 处理单个整数的情况 @a = dso\_local global i32 10;

```

1 if (initVal instanceof IrConstInt) {
2     int val = ((IrConstInt) initVal).getValue();
3     mips.addGlobal(String.format("%s: .word %d", label, val));
4 }

```

4. 处理数组情况 @arr = dso\_local global [5 x i32] [i32 1, i32 2, ...];

1. 获取数组总长度, 如果数组全是 0 (null 或 显式空) -> 使用 .space 优化;

```

1 int totalSize = irArray.irType.arraySize;
2 if (elements == null) {
3     mips.addGlobal(String.format("%s: .space %d", label, totalSize *
4     4));
5     return;
6 }

```

2. 有具体数值, 生成 .word, 先单独输出标签 mips.addGlobal(label + ":");

3. 再输出已有初始值;

```
1  StringBuilder sb = new StringBuilder();
2  int count = 0; // 计数器
3  for (int i = 0; i < elements.size(); i++) {
4      if (count == 0) sb.append(".word "); // 每行开头加 .word
5      IrConstInt constInt = (IrConstInt) elements.get(i);
6      sb.append(constInt.getValue());
7      count++;
8      // 每 50 个元素, 或者到了最后一个元素, 就换行输出
9      if (count == 50 || (i == elements.size() - 1 && elements.size()
== totalSize)) {
10         mips.addGlobal(sb.toString());
11         sb = new StringBuilder(); // 清空 buffer
12         count = 0;
13     } else {
14         sb.append(", ");
15     }
16 }
```

4. 最后补零;

```
1  // 处理补零逻辑
2  for (int i = elements.size(); i < totalSize; i++) {
3      if (count == 0) sb.append(".word ");
4      sb.append("0");
5      count++;
6      if (count == 50 || i == totalSize - 1) {
7          mips.addGlobal(sb.toString());
8          sb = new StringBuilder();
9          count = 0;
10     } else {
11         sb.append(", ");
12     }
13 }
```

## 处理函数

首先手动添加跳转 `main` 函数指令;

```
1  mips.addInst("j main");
2  mips.addInst("nop");
```

读取 `irModule` 中所有 `IrFunction` 类;

1. 初始化栈;

```
1  offsetMap.clear();
2  currentFunctionStackSize = 0; // 重置栈计数
3  allocaArrayOffsets.clear(); // 重置 alloca 数组偏移记录
4  Map<IrValue, Integer> regAllocation = new HashMap<>(); // 记录
```

2. 预计算栈空间;

```

1. 1 // 为保存寄存器预留空间 ($ra, $fp)
   2 // $ra @ -4($fp), $fp @ -8($fp)
   3 currentFunctionStackSize += 8; // 目前只记录，具体指令在后面

```

```

2. 1 // 为参数分配空间
   2 for (IrValue arg : function.getParameters()) {
   3     allocateStack(arg, 4); // 每个参数分配 4 字节

```

3. 为函数体内所有有返回值的指令分配空间;

```

1 for (IrBasicBlock bb : function.getBasicBlocks()) {
2     for (Instruction instr : bb.getInstructions()) {
3         // 如果是 Alloc 指令，需要特殊处理：
4         // 1. 分配指针变量本身的空间 (4字节)
5         // 如果是 alloc 数组，还需要额外在栈上挖一块空地
6         // 比如: %arr = alloc [10 x i32]
7         // 2. 分配数组实体的空间 (N字节)
8         if (instr instanceof AllocateInstruction) {
9             // 1. 为指针变量 分配 4 字节
10            allocateStack(instr, 4);
11            // 2. 为数组实体分配 N 字节
12            int size = ((AllocateInstruction)
instr).getAllocatedSize();
13            if (size % 4 != 0) {
14                size += (4 - (size % 4));
15            }
16            currentFunctionStackSize += size;
17            // 3. 记录数组实体相对于 $fp 的偏移量
18            // 实体位于当前栈底 (也就是 -currentFunctionStackSize)
19            allocArrayOffsets.put((AllocateInstruction) instr, -
currentFunctionStackSize);
20        } else if (instr instanceof PhiInstr) {
21            allocateStack(instr, 4);
22        } else if (!instr.irType.isVoid()) {
23            allocateStack(instr, 4);
24        }
25    }
26 }

```

3. 生成函数序言;

1. 输出函数标签，去除@，替换\_

```

1 String label = function.irName.substring(1);
2 label = label.replace("@", "").replace(".", "_");
3 mips.addInst("\n" + label + ":");

```

2. 先建立栈帧指针，再开栈，最后保存寄存器

```

1 // 1. 保存旧的 $fp 到当前栈顶下方的预留位
2 mips.addInst("sw $fp, -8($sp)");
3
4 // 2. 设置新的 $fp (指向当前栈帧的基址，即 old SP)
5 mips.addInst("move $fp, $sp");

```

```

6
7 // 3. 分配栈空间 (更新 $sp)
8 if (currentFunctionStackSize > 32767) {
9     // 如果栈太大, 不能用立即数, 使用 $t0 中转
10    mips.addInst("li $t0, -" + currentFunctionStackSize);
11    mips.addInst("addu $sp, $sp, $t0");
12 } else {
13     // 栈较小, 直接用 addiu
14    mips.addInst("addiu $sp, $sp, -" +
currentFunctionStackSize);
15 }
16
17 // 4. 保存 $ra
18 // 注意: 此时已分配空间, 使用 $fp 寻址 (因为 $fp == Old SP)
19 if (!isLeaf) {
20    mips.addInst("sw $ra, -4($fp)");
21 }
22
23 // 5. 保存 Callee-Saved 寄存器 ($sx)
24 int extraSaveSize = 0;
25 // 建立一个 map 传给 EmitInstruction, 用于函数返回时恢复寄存器
26 Map<Integer, Integer> sRegStackOffsets = new HashMap<>();

```

### 3. 保存参数

```

1. 1 // 辅助方法: 处理大偏移量的 sw
2    private static void saveToStackOrReg(String srcReg, int
offset, boolean useFp) {
3        if (offset >= -32768 && offset <= 32767) {
4            mips.addInst("sw " + srcReg + ", " + offset + "
($fp)");
5        } else {
6            mips.addInst("li $at, " + offset);
7            mips.addInst("addu $at, $fp, $at");
8            mips.addInst("sw " + srcReg + ", 0($at)");
9        }
10    }

```

2. 四个参数以内直接存入栈;

3. 大于四个参数, 先计算参数位置并加载到临时寄存器, 之后移入栈;

```

1 // 检查参数是否被分配到了寄存器
2 Integer allocatedReg = regAllocation.get(arg);
3 if (i < 4) {
4     if (allocatedReg != null) {
5         // 优化: 直接移入分配的寄存器 $sx
6         mips.addInst("move " +
RegisterAllocator.REG_NAMES[allocatedReg] + ", $a" + i);
7     } else {
8         // 未分配寄存器: 存入栈
9         saveToStackOrReg("$a" + i, offset, true);
10    }
11 } else {
12    // 1. 计算该参数在 Caller 栈帧中的位置

```

```

13 // 第5个参数在 0(Old $sp), 即 0($fp) 第6个参数在
   4($fp)...
14 int callerOffset = (i - 4) * 4;
15
16 // 2. 从 caller 栈帧加载参数到临时寄存器 $t0
17 // 注意: 这里是正偏移, 访问的是 caller 放置参数的区域
18 mips.addInst("lw $t0, " + callerOffset + "($fp)");
19
20 // 3. 将参数保存到当前函数的局部栈帧
21 // offset 是 Step 1 分配的负偏移量
22 // 后续指令访问 %arg 时, 统一去 offset($fp) 读取
23 if (allocatedReg != null) {
24     mips.addInst("move " +
RegisterAllocator.REG_NAMES[allocatedReg] + ", $t0");
25 } else {
26     saveToStackOrReg("$t0", offset, true);
27 }
28 }
29 }

```

#### 4. 遍历基本块

1. 函数相关信息传入 `EmitInstruction` 类, 之后调用 `emit()` 方法处理基本块中指令;
2. 遍历函数内基本块, 再遍历基本块内指令;

```

1 for (IrBasicBlock bb : function.getBasicBlocks()) {
2     String bbLabel = bb.irName.replace("@", "").replace(".",
   "_");
3     // 生成块标签 (block_name:)
4     mips.addInst(label + "_" + bbLabel + ":");
5
6     for (Instruction instr : bb.getInstructions()) {
7         // 调用指令翻译器
8         emitter.emit(instr, optimize, isLeaf);
9     }
10 }

```

## EmitInstruction类

必要变量:

```

1 private final MipsModule mips;
2 private final HashMap<IrValue, Integer> offsetMap; // 记录为IrValue分配的栈空间
3 private final String currentFuncLabel; // 当前函数的名字 (用于拼接跳转标签)
4 private final HashMap<AllocateInstruction, Integer> allocaArrayOffsets; //
   alloc数组时记录为数组实体分配的空间
5 private boolean optimize = false; // 是否开启优化
6 private boolean isLeaf = false; // 是否为叶子函数
7 private final Map<IrValue, Integer> regAllocation; // 记录为IrValue分配的寄存器
8 private final Map<Integer, Integer> sRegStackOffsets; // 用于 epilogue 恢复

```

`emit()` 方法: 解析每个指令, 并调用解析对应指令的方法;

1. `loadToReg(IrValue val, String reg):`

1. 检查 `regAllocation`，若变量已在寄存器中，且目标寄存器与源寄存器不同，生成 `move`；若相同则跳过。
2. 常量/全局处理：若为立即数生成 `li`，若为全局变量/字符串生成 `la`。
3. 栈加载：若变量在栈上，根据 `offsetMap` 生成 `lw`。

#### 2. `saveReg(IrValue dest, String srcReg)`:

1. 若目标变量分配了寄存器并且目标寄存器与源寄存器不同，生成 `move`。
2. 若目标变量未分配寄存器，生成 `sw` 写入栈帧。

#### 3. `emitBinary(AluInstr instr)`: 根据运算符生成对应指令，之后保存结果

```
1 | mips.addInst(String.format("addu %s, %s, %s", destReg, leftReg,
  | rightReg));
2 | saveReg(instr, destReg);
```

#### 4. `emitLoad(LoadInstr instr)`: 调用 `loadToReg()`

```
1 | mips.addInst("lw $t0, 0($t1)"); // 从地址 $t1 处读取真实的值
2 | saveReg(instr, "$t0");          // 把值存入 %val 的栈槽
```

#### 5. `emitStore(StoreInstr instr)`:

```
1 | loadToReg(val, "$t0"); // 加载要存储的数据
2 | loadToReg(ptr, "$t1"); // 加载目标地址
3 | mips.addInst("sw $t0, 0($t1)"); // 写入内存
```

#### 6. `emitCmp(CmpInstr instr)`: 与 `emitBinary()` 同理，先加载操作数，然后根据运算符生成指令，最后保存结果；

#### 7. `emitBr(BranchInstr instr)`: 只处理条件跳转的情况，获取 `currentBlock`、`trueBlock`、`falseBlock` 三个基本块的名字，之后对不同情况生成跳转；

```
1 | // 如果 $t0 == 0 (即不满足条件)，跳过 j trueLabel指令
2 | mips.addInst("beqz $t0, " + skipLabel);
3 | mips.addInst("nop");
4 | // 处理 True 块的 Phi
5 | resolvePhiCopies(trueBlock, currentBlock);
6 | // 只有满足条件才执行这个长跳转
7 | mips.addInst("j " + trueLabel);
8 | mips.addInst("nop");
9 | mips.addInst(skipLabel + ":");
10 | // 处理 False 块的 Phi
11 | resolvePhiCopies(falseBlock, currentBlock);
12 | // 否则跳转 falseLabel
13 | mips.addInst("j " + falseLabel);
14 | mips.addInst("nop");
```

#### 8. `emitJump(JumpInstr instr)`: 获取基本块名，处理Phi指令，生成跳转；

#### 9. `emitCall(CallInstr instr)`: 首先获取形参，进行栈对齐；之后生成移动栈指针的指令

```
1 | if (stackArgs > 0) mips.addInst("addiu $sp, $sp, -" + (stackArgs * 4));
```

再将每一个参数存入寄存器，形参个数大于4时依次从栈顶开始存入；

```
1         if (i < 4) {
2             loadToReg(arg, "$a" + i);
3         } else {
4             loadToReg(arg, "$t0");
5             mips.addInst(String.format("sw $t0, %d($sp)", (i - 4) *
6                                     4));
7         }
```

生成 `jal` 指令调用函数，恢复栈指针，如果函数有返回值则保存；

```
1 mips.addInst("jal " + funcName);
2 mips.addInst("nop");
3 if (stackArgs > 0) mips.addInst("addiu $sp, $sp, " + (stackArgs * 4));
4 if (!instr.isType.isVoid()) saveReg(instr, "$v0");
```

#### 10. `emitRet(ReturnInstr instr)`:

1. 若返回值非 `void`，将返回值加载到 `$v0`；
2. 若是 `main` 函数，生成 `li $v0, 10` 和 `syscall` 结束程序；
3. 若非 `main` 函数，执行：
  - 根据 `sRegStackOffsets` 恢复被调用者保存的寄存器；
  - 恢复栈指针 `$sp` 和帧指针 `$fp`；
  - 若非叶子函数，恢复返回地址 `$ra`；
  - 生成 `jr $ra` 跳转指令。

#### 11. `emitAlloca(AllocatInstruction instr)`:

1. 从 `allocaArrayOffsets` 获取数组实体在栈上的偏移量；
2. 计算数组实体的绝对地址 (`$fp + offset`)，若偏移量过大则使用 `$at` 寄存器中转；
3. 将计算出的绝对地址存入该指针变量对应的栈槽中（即存储“指向数组的指针”）。

#### 12. `emitZext(ZextInstr instr)`: 将操作数加载到寄存器，生成 `andi $t0, $t0, 1` 确保高位清零，保存结果。

#### 13. `emitGEP(GepInstr instr)`:

1. 加载基地址 (`Base`) 和索引 (`Index`) 到寄存器；
2. 计算偏移量：将索引左移 2 位 (`sll $t1, $t1, 2`，即乘以 4)；
3. 地址相加：`addu $t2, $t0, $t1`；
4. 保存计算结果到目标寄存器/栈。

#### 14. **IO指令处理 (PrintInt/PrintStr)**:

1. **PrintInt**: 将待打印值加载到 `$a0`，设置 `$v0` 为 1，执行 `syscall`；
2. **PrintStr**: 将字符串地址加载到 `$a0`，设置 `$v0` 为 4，执行 `syscall`。

#### 15. `emitTrunc(TruncInstr instr)`:

1. **i32转换i1**: 生成 `andi $t0, $t0, 1` 保留最低位；
2. **i32转换成i8**: 生成 `andi $t0, $t0, 255` 保留低 8 位；
3. 保存截断后的结果。

16. **Phi指令消除 (resolvePhiCopies)**: 在跳转指令 (Branch/Jump) 生成前调用;

1. 遍历目标块中的所有 **Phi** 指令, 找到来自当前块的值;
2. 防止覆盖: 利用栈作为临时中转。先将所有需要传递的值压栈;

```
1 | loadToReg(values.get(i), "$t0");  
2 | mips.addInst("addiu $sp, $sp, -4");  
3 | mips.addInst("sw $t0, 0($sp)");
```

3. 再按顺序弹栈并保存到 **Phi** 指令对应的目的操作数位置。

```
1 | mips.addInst("lw $t0, 0($sp)");  
2 | mips.addInst("addiu $sp, $sp, 4");  
3 | saveReg(phi, "$t0");
```