实现文档

20373696 郭鸿宇

1 概述

本课程实验为实现一个将SysY语言翻译为MIPS汇编语言的编译器，编写编译器所用语言为C++11.

编译过程可以分为五个阶段：

- 词法分析
- 语法分析
- 语义分析和中间代码生成
- 代码优化
- 目标代码生成

此外，还有符号表管理和错误处理两部分穿插在五个阶段之中。接下来的架构设计章节将详细介绍每一部分。代码优化由于内容比较多，因此另起一章介绍。

该项目的文件组织形式如下：

```
├── Makefile    # 项目编译运行脚本
├── README.md   # 项目说明
├── SysY.c      # SysY文法
├── lib         # 存储文法定义输入输出函数实现
└── src
    ├── frontend
    │   ├── lexer   # 词法分析实现
    │   │   ├── lexer.cpp
    │   │   ├── lexer.hpp
    │   │   └── token.cpp
    │   └── parser  # 语法分析与符号表实现
    │       ├── parser.cpp
    │       ├── parser.hpp
    │       ├── symbol_table.cpp
    │       ├── symbol_table.hpp
    │       ├── syntax_tree.cpp
    │       ├── syntax_tree.hpp
    │       └── utils.cpp
    ├── include     # 调试辅助宏
    │   └── error.hpp
```

```
    ├── ir          # llvm ir 数据结构定义与实现
    │   ├── ir.cpp
    │   ├── ir.hpp
    │   ├── ir_build.cpp
    │   ├── ir_build.hpp
    │   ├── ir_utils.hpp
    │   └── opt     # 中间代码优化
    │       ├── dce.cpp
    │       ├── dull_muldiv.cpp
    │       ├── gcp.cpp
    │       ├── mem2reg.cpp
    │       ├── opt.hpp
    │       └── rmblk.cpp
    ├── machine     # 目标代码数据结构定义与实现
    │   ├── asm.cpp
    │   ├── asm.hpp
    │   ├── asm_build.cpp
    │   ├── asm_build.hpp
    │   ├── opt     # 目标代码优化
    │   │   ├── asm_opt.cpp
    │   │   └── asm_opt.hpp
    │   ├── reg_allocator.cpp
    │   └── reg_allocator.hpp
    ├── main.cpp
    └── utils       # 辅助工具
        ├── error_handle.cpp
        ├── error_handle.hpp
        ├── test.cpp
        └── test.hpp

11 directories, 40 files
```

2 架构设计

2.1 词法分析

词法分析部分代码实现位于 `src/frontend/lexer` 中。词法分析一般而言有两种实现方式，第一种是给语法分析提供一个返回 `token` 的函数接口，语法分析每调用一次该接口，就返回一个 `token` 。第二种是词法分析作为一个独立的过程，解析完所有输入符号并把 `token` 放在一个数组中，语法分析使用该数组进行递归下降分析。

第一种方法的问题在于语法分析时可能要前瞻多个 `token` 才能决定接下来使用哪一个产生式进行解析，而前瞻就代表词法分析器必须把这些 `token` 解析出来。但是词法分析器是无法倒退的，因此语法分析器必须将它已经拿到且之后可能再使用的 `token` 都存起来，非常麻烦，且词法分析和语法分析耦合度高。而在第二种方法中，语法分析器与词法分析器彻底解耦，语法分析器前瞻 `token` 也非常简单。因此我采用了第二种方法。

词法分析解析的结果是由 `TokenInfo` 结构体组成的 `vector` 。 `TokenInfo` 结构体定义如下：

```
struct TokenInfo {
  TokenType tokenType;  # token的类型
  std::string str;      # token本身
  int value;            # 如果token是整数，则存储整数的值
  int line;             # token所在行号，用于错误处理
};
```

识别单词 token 的过程如下，循环读取字符：

- 读取掉空白符，直到下一个非空白符或 EOF 为止。
- 如果当前字符为 EOF ，代表解析完毕，break
- 如果当前字符为 / ，则判断是否为单行或多行注释，如果是则读掉注释。
- 如果当前字符为数字，调用解析数字的函数
- 如果当前字符属于标识符的开始符号集合，则调用解析标识符函数
- 如果当前字符为双引号，调用解析字符串函数
- 如果以上条件均不满足，就调用解析其它符号的函数。注意对于 = 和 == 这样的关键字，不能只读一个就判断，要读完第二个字符后再判断。

2.2 语法分析

语法分析部分代码实现位于 src/frontend/parser 中。语法分析的过程即使用递归下降法解析词法分析生成的 token 序列，生成语法树。首先要改写文法，消除左递归，如将：

```
AddExp ::=  MulExp
          | AddExp ( '+' | '-' ) MulExp       // 左递归
```

改为

```
AddExp ::= MulExp{( '+' | '-' ) MulExp }
```

之后对于一个非终结符的不同产生式该选择哪一个的问题，通过前瞻多个 token 解决。我没有用回溯的方法，因为回溯的时间复杂度无法估计。

对于大部分非终结符，我都为其单独建立了一个类进行表示。这是因为不同非终结符所要表示的信息是不同的，比如 VarDecl 类需要包含的属性是 BType 和一系列 VarDef 。而 VarDef 类要包含的是 Ident 和 InitVal 。这种将类直接映射到非终结符的做法相比于只有一个语法树节点类，而用标识符或者枚举类型分辨到底是哪一种非终结符的方法而言，似乎有些过于具体了，毕竟光是类的定义就有四百多行。

但是在我看来，语法分析的过程就是要将源代码的结构梳理出来，而表示源代码结构的复杂度是固定的，对每一个非终结符建类，就是将这个复杂度表现在了语法树的类型多样性上，也可以说直接把语义信息编码在了类型定义中。这样一方面比较难以出现难以察觉的错误，因为举例而言，类型规定了 LVal 下只能有 Exp 类型的节点，如果不小心将 AddExp 插在 LVal 下，编译器会直接报类型不匹配错误。 另外在后续进行中间代码生成时，不用判断语法树节点到底是哪一个语法成分，因为类型已经代表了这个语法成分，大大简化了生成中间代码的过程。而如果使用标识符判断语法树节点到底是什么语法成分，虽然不用建这么多类了，但是很容易出错，而且将语义解析的复杂度转移到了代码生成中。

在递归下降法中，对每一个非终结符，都使用该非终结符特定的解析函数进行分析。解析函数会先实例化一个该非终结符类型对象，然后按照文法产生式调用对应非终结符的解析函数，最后返回指向该终结符对象的指针。

比如解析 Cond 的函数如下：

```
CondNode* Parser::condAnalyse()
{
  CondNode* node = new CondNode();
  node->lOrExpNode = lOrExpAnalyse();
  return node;
}
```

两个非常重要的工具函数是 popToken 和 peekToken 。 popToken 会弹出一个 token ， 将指针移向下一个 token ， 而 peekToken 会返回前瞻的 token ， 但不移动指针。这两个函数将对 token 序列的访问进行了封装。

```
TokenInfo* Parser::popToken()
{
  if (tokenInfoList[nowTokenListPtr]->tokenType == TokenType::END) {
    Log("error : pop END\n");
    return NULL;
  }
  TokenInfo* now = tokenInfoList[nowTokenListPtr];
  nowTokenListPtr++;  /* 将指针加1 */
  return now;
}

TokenInfo* Parser::peekToken(int num)
{
  if (nowTokenListPtr + num >= (int)tokenInfoList.size() ||
      nowTokenListPtr + num < 0) {
    Log("error : peek out of bounds\n");
    return NULL;
  }
  return tokenInfoList[nowTokenListPtr + num]; /* 返回当前指针加num个token，但不移动指针*/
}
```

有两个 Stmt 的产生式比较特殊：

```
Stmt -> LVal '=' Exp ';' |[Exp] ';'
```

在解析 Stmt 时，如果当前 token 是 IDENFR 并且下一个 token 不是 LPARENT ， 那么当前元素一定为 LVal ， 并且无法判断是 LVal '=' Exp ';' 还是 [Exp] ';' ， 此时先解析一个 LVal ， 然后判断接下来的 token 是否为等号，如果是等号就解析 LVal '=' Exp ';' ， 否则解析 [Exp] ';' 。

2.3 错误处理

我在词法分析与递归下降阶段完成所有的错误处理检查工作，为此需要先建立符号表。

2.3.1 符号表设计

符号表定义位于 src/frontend/parser/symbol_table.hpp 中，一些辅助函数实现位于 src/frontend/parser/symbol_table.cpp 中。我采用了双向树结构的符号表设计，每一个作用域都有一张独立的符号表，上下级表之间保存指向对方的指针，定义如下所示:

```
class SymbolTable {
  private:
  SymbolTable* parent; /* 指向父节点指针 */
  std::unordered_map<std::string, SymbolItem*> name2symbols; /* 符号名称到符号的映射 */
  std::vector<SymbolTable*> childs; /* 指向子节点指针 */

  public:
  SymbolTable(int id) : isTop(false), parent(NULL), tableId(id) {}
  /**
   * @brief 新建一个子表并返回
   * @return SymbolTable*
   */
  SymbolTable* newSon();
  /**
   * @brief 返回父表指针
   * @return SymbolTable*
   */
  SymbolTable* findParent();
  /**
   * @brief 在表中插入一条记录
   * @param symbolName 符号名
   * @param node 语法树节点
   * @param type 语法树节点类型
   */
  SymbolItem* insertNode(std::string* symbolName, SyntaxNode* node, SyntaxNodeType type);
```

在递归下降时，进入一个新的作用域时，调用 newSon 函数生成一个新的符号表。在退出作用域时，调用 findParent 函数将当前符号表更新为父节点。使用 insertNode 在当前符号表中插入一个符号。

这种树结构符号表可以保证每次在一张符号表上能且仅能看到该符号表本身以及所有它的祖先节点符号表，看不见与其没有直接隶属关系的符号表。在加入符号时，符号表仅扫描本表内是否有重名变量。在查找符号表时，从本级开始逐级向上查找，找到最近的一个即可，正好满足 SysY 语言关于变量作用域同名符号内层覆盖外层的要求。

在符号表中存储的符号分为两类：函数类型符号类型为 FuncSymbolItem ,其它符号类型为 ObjectSymbolItem 。它们都继承自基类 SymbolItem：

```
class SymbolItem {
  SymbolType symbolType;  /* 符号具体类型，标识是FuncSymbolItem还是ObjectSymbolItem */
  int line; /* 符号所在行号 */
};

class ObjectSymbolItem : public SymbolItem {
  bool isConst;     /* 是否为常量 */
  int dimension;    /* 变量的维度，如果是int，则维度为0 */
  std::vector<int> initValues; /* 数组或变量初始值 */
};

class FuncSymbolItem : public SymbolItem {
  SymbolType returnType; /* 函数返回类型 */
  std::vector<ObjectSymbolItem*> funcFParams; /* 函数参数类型 */
```

```
};
```

2.3.2 错误处理实现

ErrorInfo 类保存具体的错误信息与错误行号。 ErrorList 类保存所有的错误信息，这两个类都定义在 src/utils/error_handle.hpp 中。

```
class ErrorInfo {
    int line;
    char errorType;
};

class ErrorList {
  std::map<int, ErrorInfo*> errorInfos;
};
```

在 src/utils/error_handle.cpp 中声明了一个 ErrorList 类型的全局变量 errorList 。其它文件中函数如果要添加错误信息均 extern 这个 errorList 。

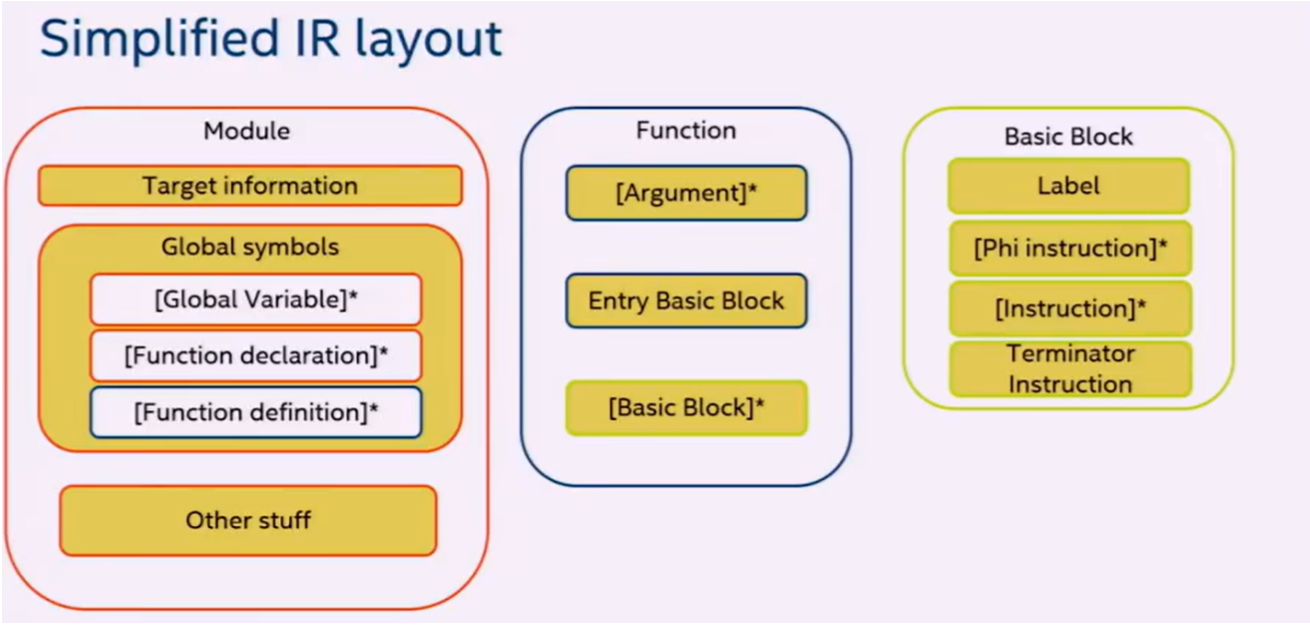
- FormatString 出现非法符号或者 printf格式字符与表达式个数不匹配：词法分析阶段不检测该错误，语法分析阶段解析到printf时查看。
- 名字重定义：在插入符号表时，符号表插入函数检测是否有同名符号，如果有就增加一个错误信息到 errorList 中。
- 未定义的名字：在分析到标识符时查符号表，如果没有这个符号就增加一个错误信息到 errorList 中。
- 函数参数个数或者类型不匹配：在分析到函数调用时查符号表，对比调用参数与符号表中函数定义信息，如果不匹配就增加一个错误信息到 errorList 中。
- 无返回值的函数存在不匹配的 return 语句：解析到 return 语句时，使用当前函数名查找符号表，查看是否有返回值，如果不匹配就增加一个错误信息到 errorList 中。
- 有返回值函数缺少 return ： 设置一个全局变量 hasRE 。若解析到 return 就将 hasRE 置为 true 。在解析完 block 时，如果函数有返回值并且 hasRE 为 false ， 就增加一个错误信息到 errorList 中。
- 不能改变常量的值：遇到等号左边的左值时直接查符号表即可解决。
- 缺少分号/ 右小括号/ 右中括号：递归下降解析到分号 / 右小括号 / 右中括号前时，直接查看下一个符号是不是分号 /右小括号 /右中括号，如果不是就报错。
- 在非循环块中使用 break和 continue：设置一个全局变量 cycleDepth表示当前所处循环深度。进入 while 时将 cycleDepth 加1，退出时将 cycleDepth 减一。如果识别到 break 或者 continue 时 cycleDepth 不为 0 ， 代表出错。

2.4 语义分析和中间代码生成

语义分析的最终目标是生成中间代码，我采用的中间代码是 llvm ir 。采用 llvm ir 的好处是生成中间代码后可以使用 llvm 编译运行，查看中间代码是否正确，如果出现语法错误， llvm 会给出比较详细的错误提示，非常容易 debug 。另外 llvm ir 具有非常合适的抽象层级，且必须是静态单赋值形式，非常适合做优化。

2.4.1 llvm ir结构

- LLVM IR 文件的基本单位称为 module
- 一个 module 中可以拥有多个顶层实体，比如 function 和 global variavle
- 一个 function define 中至少有一个 basicblock
- 每个 basicblock 中有若干 instruction，并且都以 terminator instruction 结尾



2.4.2 llvm ir 指令

以下列举了所有使用到的 llvm ir 指令。

instructions

llvm ir	usage	intro
add	<result> = add <ty> <op1>, <op2>	
sub	<result> = sub <ty> <op1>, <op2>	
mul	<result> = mul <ty> <op1>, <op2>	
sdiv	<result> = sdiv <ty> <op1>, <op2>	有符号除法
icmp	<result> = icmp <cond> <ty> <op1>, <op2>	比较指令,返回类型是i1
xor	<result> = xor <ty> <op1>, <op2>	按位异或
and	<result> = and <ty> <op1>, <op2>	与
or	<result> = or <ty> <op1>, <op2>	或
srem	<result> = srem <ty> <op1>, <op2>	取模
call	<result> = call [ret attrs] <ty> <fnptrval>(<function args>)	函数调用

llvm ir	usage	intro
alloca	`<result> = alloca <type>`	在当前函数运行栈上分配内存，函数运行结束后自动释放
load	`<result> = load <ty>, <ty>* <pointer>`	读取内存
store	`store <ty> <value>, <ty>* <pointer>`	写内存
getelementptr	`<result> = getelementptr <ty>, * {, [inrange] <ty> <idx>}*` `<result> = getelementptr inbounds <ty>, <ty>* <ptrval>{, [inrange] <ty> <idx>}*`	计算目标元素的位置（仅计算）
phi	`<result> = phi [fast-math-flags] <ty> [ <val0>, <label0>], ...`	
zext..to	`<result> = zext <ty> <value> to <ty2>`	类型转换，将 ty的value的type转换为ty2
shl	`<result> = shl <ty> <op1>, <op2>`	左移
lshr	`<result> = lshr <ty> <op1>, <op2>`	逻辑右移
ashr	`<result> = ashr <ty> <op1>, <op2>`	算术右移

terminator insts

llvm ir	usage	intro
br	`br i1 <cond>, label <iftrue>, label <iffalse>` `br label <dest>`	改变控制流
ret	`ret <type> <value>` ,`ret void`	退出当前函数，并返回值（可选）

2.4.3 llvm ir类型

在 `llvm ir` 中，每个 `value` 都有一个类型，以下列举了所有使用到的类型：

type	intro
LabelType	标签类型，基本块的标号属于这一类型
VoidType	空类型，函数没有返回值时的返回类型为void
FunctionType	函数类型
IntegerType	整数类型，有一个位宽属性

PointerType	指针类型，有一个指向的类型的属性
ArrayType	数组类型，包括数组元素类型与数组元素个数两个属性
OtherType	其它类型，如instruciton和module就属于其它类型

2.4.4 生成llvm ir

解析语法树并生成 `llvm ir` 的代码位于 `src/ir/ir_build.cpp` 中。为了使生成的 `llvm ir` 满足 `llvm` 规定的代码结构，有两个重要方面需要考虑：

- 保证生成的代码是静态单赋值形式
- 正确处理循环分支

2.4.1.1 保证生成的代码是静态单赋值形式

静态单赋值形式即一个变量只能被定义一次，但可以被使用多次。这里的变量就是 `llvm ir` 中的寄存器。由于循环分支的存在，一个变量只定义一次是无法实现程序的语义的，因此引入了 `phi` 函数。然而中间代码生成时直接生成具有 `phi` 函数的 `llvm ir` 过于复杂，因此可以先生成 `load/store` 形式的 `llvm ir` ，一个内存地址被 `store` 多次并不破坏静态单赋值形式。

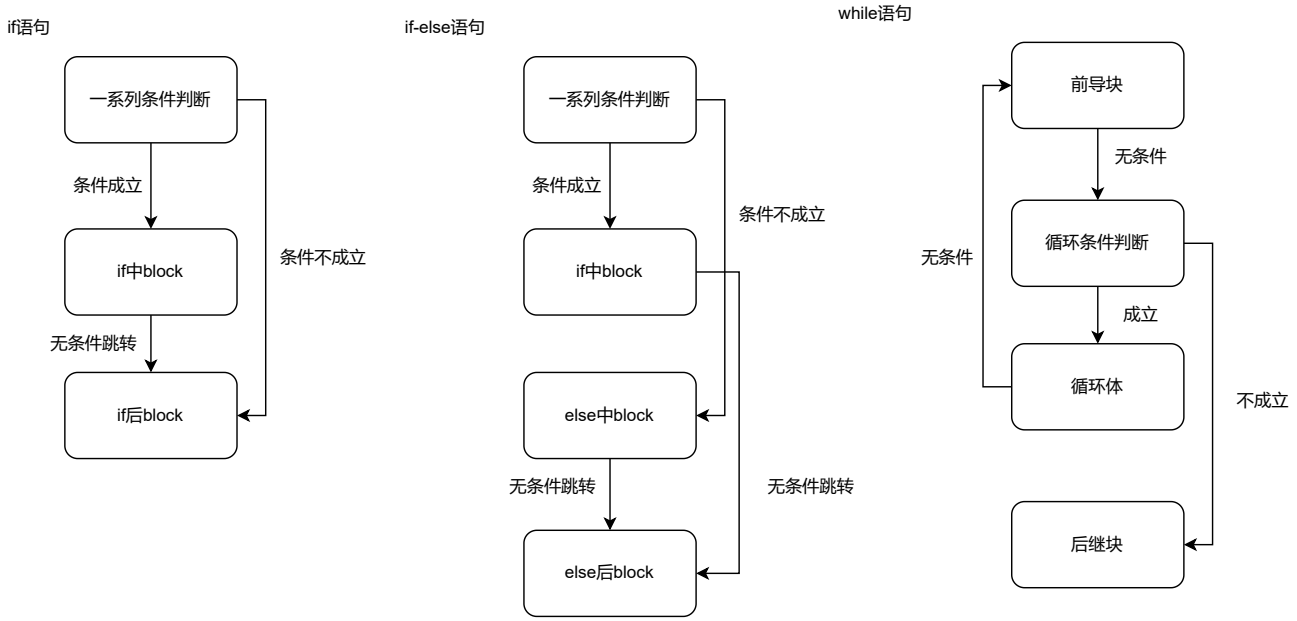
因此这里生成的代码是 `load/store` 形式的 `llvm ir` 。通过一个中端优化 `mem2reg` 可以将代码中多余的 `load/store` 消除并引入 `phi` 函数， `mem2reg` 具体将在代码优化一章介绍。

为了生成 `load/store` 形式的 `llvm ir` ，我们可以在遇到局部数组和变量的声明时生成 `alloca` 指令，并在符号表中记录 `alloca` 返回的地址寄存器号。在遇到对局部数组和变量的定义时，查符号表找到地址寄存器号，并生成 `store` 指令。在遇到对局部数组和变量的使用时，查符号表找到地址寄存器号，生成 `load` 指令。

同时，维护一个全局变量 `nowLlvmIrId` ，表示当前未分配的 `id` 号。新建虚拟寄存器时将虚拟寄存器号分配为 `nowLlvmIrId` ，并将 `nowLlvmIrId` 加 `1` ，如此就保证了没有两个被定义的虚拟寄存器是相同的，即维护了代码的静态单赋值形式。

2.4.1.2 正确处理循环分支

对于循环分支的处理如下图所示。

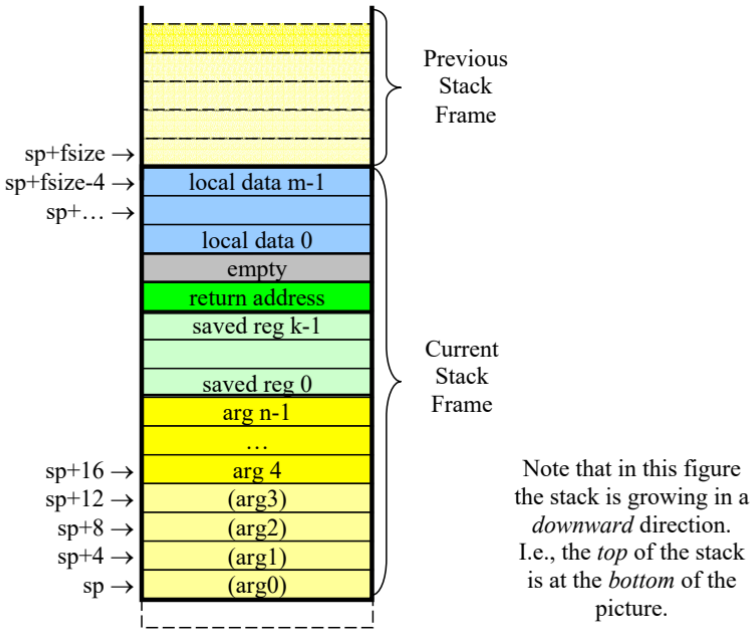


2.5 目标代码生成

目标代码生成阶段即为从将 `llvm ir` 翻译为可以在 `MARS` 中运行的 `MIPS` 汇编代码。实现位于 `src/machine/asmbuild.cpp` 。目标代码生成分为两部分：

- 指令选择：将 `llvm ir` 指令翻译为 `MIPS` 指令。此时指令定义与使用的寄存器仍然是虚拟寄存器。
- 寄存器分配：为虚拟寄存器分配物理寄存器，并在需要的地方插入 `lw/sw` 指令。

2.5.1 函数调用约定



栈向下增长，`sp` 指针指向栈顶部。栈一共分为 5 个部分：

- 最顶部的参数区用来存放要传递给改函数可能调用的任何一个函数的参数。这个空间**至少有4个字的大小**（让**被调用过程**存放 `$a0-$a3` ）,这部分空间的大小取决于可能调用的函数中所需的最大参数个数。注意这个空间**是调用者分配，但是由被调用者使用的。**
- 保存寄存器部分用来存放该过程需要使用的 `sp` 寄存器。需要在进入函数的时候保存，退出函数的时候恢复。
- 返回地址部分用来存放返回地址，在进入函数的时候保存。
- 局部数据部分用来存放函数执行过程中要用到的数据，以及存放在调用别的函数的时候需要保存的临时寄存器。

在函数的 `prologue` 部分，需要：

- 开辟栈空间（减 `sp` 寄存器）
- 保存返回地址
- 保存之后会用到的调用者保存寄存器

在函数的 `epilogue` 部分，需要：

- 恢复进入函数时保存的调用者保存寄存器
- 将栈中保存的返回地址 `load` 进 `ra` 寄存器
- 消除栈帧（加 `sp` 指针）
- 用 `jr $ra` 返回

调用函数时，需要：

- 生成传递的参数（向栈空间底部 store）
- jal到函数

调用函数返回时：

- 如果调用的函数有返回值，需要及时处理。

MARS中的特殊处理

MARS中用 .data声明数据段，内存中地址从 0 开始。用 .text声明代码段。

输入输出语句在生成目标代码时直接内联，不生成函数。

- 使用 .word声明全局变量：

```
int arr[3] = {1,2,3};
arr: .word 1,2,3

int a;
a: .word 0

int a = 5;
a: .word 5
```

- 使用 .asciiz 声明字符串，注意 .asciiz 自带终结符:

```
const_str: .asciiz "123"
```

- 使用 la 指令获取全局变量的地址。
- 输出 int：putint

```
add $a0, $0, $s0    # a0存放要输出的值
addi $v0, $0, 1     # v0值为1
syscall
```

- 输出字符串： putstr

```
la $a0 str_addr # a0存放要输出字符串的地址
addi $v0, $0, 4 # v0值为4
syscall
```

- 读入 int: getint

```
addi $v0, $0, 5 # v0值为5
syscall         # 输入的int值在v0中
```

2.5.2 指令选择

首先是 `llvm ir`指令到 `mips` 指令的映射，这个映射非常简单，具体见下表。需要注意的几个事情：

- `llvm ir` 中允许指令的两个操作数均为立即数，但是 `mips` 指令中最多有一个是立即数。因此如果出现两个立即数的情况，需要生成 `addiu` 指令将其中一个立即数变为寄存器。
- 函数参数全部通过内存传递
- `llvm ir`中的 `br` 指令在 `mips` 中没有直接对应，需要使用 `bne` 与 `j` 两条指令模拟。
- `llvm ir` 中的 `icmp` 指令在 `mips` 中没有直接对应，需要使用多条指令模拟效果。

llvm ir	**MIPS**	**说明**
`<result> = add <ty> <op1>, <op2>`	`add result, op1, op2`	有常数使用addiu，对于两个都是常数的指令需要两个addiu
`<result> = sub <ty> <op1>, <op2>`	`sub result, op1, op2`	同add
`<result> = mul <ty> <op1>, <op2>`	`mult op1, op2`+`mflo result`	如果有常数需要先addiu
`<result> = sdiv <ty> <op1>, <op2>`	`div op1, op2` + `mflo result`	如果有常数需要先addiu
`<result> = xor <ty> <op1>, <op2>`	`xor result, op1, op2`	如果有常数需要先addiu
`<result> = and <ty> <op1>, <op2>`	`and result, op1, op2`	如果有常数需要先addiu
`<result> = or <ty> <op1>, <op2>`	`or result, op1, op2`	如果有常数需要先addiu
`<result> = srem <ty> <op1>, <op2>`	`div op1, op2`+`mfhi result`	如果有常数需要先addiu
`<result> = call [ret attrs] <ty> <fnptrval>(<function args>)`		将参数压栈，并生成JAL指令
alloca		记录栈分配空间，然后直接删除该指令即可
`<result> = load <ty>, <ty>* <pointer>`	`lw result, 0(pointer)`	直接翻译即可
`store <ty> <value>, <ty>* <pointer>`	`sw value, 0(pointer)`	直接翻译即可
`<result> = getelementptr <ty>, <ty>* <ptrval>, {<ty> <index>}*`		直接翻译即可
`zext to`	move	
`br <label>`	`j label`	
`br i1 <cond> label <iftrue>, label <iffalse>`	`bne <cond>, $0, <iftrue>` + `j iffalse`	
icmp		具体见下表
`ret`	`jr $ra`	需要把返回地址load进ra，如果有返回值需要给v0寄存器赋值。

icmp指令映射

type	MIPS	说明
eq	xor result, op2, op1 + sltiu result, result, 1	条件成立result1，否则为0
ne	xor result, op2, op1	条件成立result为1，否则为0
ugt	sltu result, op2, op1	条件成立result为1，否则为0
uge	sltu result, op1, op2 + sltiu result, result, 1	条件成立result为1，否则为0,sltu result, result, 1可以翻转1，0
ult	sltu result, op1, op2	条件成立result为1，否则为0
ule	sltu result, op2, op1 + sltiu result, result, 1	条件成立result为1，否则为0
sgt	slt result, op2, op1	条件成立result为1，否则为0
sge	slt result, op1, op2 + sltiu result, result 1	条件成立result为1，否则为0
slt	slt result, op1, op2	条件成立result为1，否则为0
sle	slt result, op2, op1 + sltiu result, result 1	条件成立result为1，否则为0

注意在指令选择完成后，有些指令中的寄存器已经是物理寄存器了，比如开辟栈空间的指令，就是对 sp 寄存器进行操作。但是大部分寄存器仍然是虚拟寄存器。

寄存器分配只对虚拟寄存器进行分配，在寄存器分配器眼中，指令中已经分配的物理寄存器和立即数是没有区别的，因为指令选择阶段确定的物理寄存器之间一定没有冲突，而且也不会和寄存器分配器分配的物理寄存器产生冲突。

在我的寄存器类中，使用 isPhysReg 代表该寄存器是否是物理寄存器。

```
class AsmReg : public AsmOperand {
  public:
  bool isPhysReg; /* 为 true表示是物理寄存器，使用物理寄存器号，否则使用虚拟寄存器号 */
  int virtNumber; /* 虚拟寄存器号 */
  int physNumber; /* 物理寄存器号 */
};
```

使用简化方式对指令进行存储

一条 mips 指令需要保存的信息只有它的指令类型，操作数，以及结果。如果对每一种指令都建立一个类表示，并不方便。因此我只建立一个 AsmInst 类表示指令，而用 AsmInstIdtfr 表示指令类型。用一个 vector 保存指令的操作数。操作数在 vector 中的顺序与汇编代码中操作数的顺序相同。另外，操作数的读写属性也需要保存。初始化 AsmInst 时，使用 initializer_list 接收不定数量的操作数并存入 ops 。

```
class AsmInst {
  public:
  AsmInstIdtfr asmInstIdtfr;
  /* second是寄存器的读写属性，imm与label默认为读 */
  std::vector<std::pair<AsmOperand*, RWP>> ops;

  AsmInst(AsmInstIdtfr idtfr, std::initializer_list<std::pair<AsmOperand*, RWP>> ops) {
    this->asmInstIdtfr = idtfr;
    for (auto i : ops) {
      this->ops.push_back(i);
    }
  }
};
```

我用宏封装了给指令操作数指定读写属性的操作：

```
#define WRR(w, r1, r2) {(w), WRITE_RWP}, {(r1), READ_RWP}, {(r2), READ_RWP}
#define RRR(r1, r2, r3) {(r1), READ_RWP}, {(r2), READ_RWP}, {(r3), READ_RWP}
#define R(r) {(r), READ_RWP}
#define RR(r1, r2) {(r1), READ_RWP}, {(r2), READ_RWP}
#define WR(w, r) {(w), WRITE_RWP}, {(r), READ_RWP}
#define W(w) {(w), WRITE_RWP}
#define RRR(r1, r2, r3) {(r1), READ_RWP}, {(r2), READ_RWP}, {(r3), READ_RWP}
```

比如要生成一条 `addiu $4, $5, 32`，可以这样写：

```
new AsmInst(AsmInstIdtfr::ADDIU_AII, {WRR(new AsmReg(4), new AsmReg(5), new AsmImm(32))})
```

2.5.3 简单寄存器分配

简单寄存器分配的实现位于 `src/machine/reg_allocator.cpp` 。这里介绍的寄存器分配是一种简单的 `lw/sw` 形式寄存器分配方式，即给所有的虚拟寄存器在栈上分配空间，当需要用寄存器时，从栈中 `load` 出来。计算完毕后马上将结果 `store` 回内存。比较复杂的图着色寄存器分配将在代码优化一章中进行介绍。

- 首先声明一个空闲物理寄存器池，寄存器池中初始有3个空闲寄存器。
- 遍历每条指令，对指令中的每一个虚拟寄存器操作数，检查是否在栈上分配了空间，如果没有就分配栈空间。并记录虚拟寄存器号到栈偏移的映射。
- 遍历每条指令：
  - 对指令中的每一个是虚拟寄存器且读写属性为读的操作数，找到该虚拟寄存器的栈空间偏移，从空闲寄存器池中拿出一个物理寄存器，在指令前插入 `lw` 指令，将值从内存 `load` 到分配的物理寄存器中。并将操作数改为物理寄存器与物理寄存器号。
  - 如果指令中存在读写属性为写的虚拟寄存器，则先从空闲寄存器池中拿出一个物理寄存器，指令中被写的操作数改为该物理寄存器，然后在该指令后生成 `sw` 指令，将该物理寄存器的值 `store` 到栈上对应位置。
  - 在指令处理完后，释放所有用到的物理寄存器到寄存器池中。
- 将开辟栈空间与回收栈空间的指令的立即数加上寄存器分配过程中新分配的栈空间大小。

3 代码优化

优化代码分为中端优化和后端优化两部分，中端优化位于 src/ir/opt 文件夹中，opt.hpp 定义了一些优化的数据结构。管理中端优化函数为 opt.cpp/optir。优化包括：

- mem2reg.cpp
- dce.cpp（死代码删除）
- gcp.cpp（全局常量传播）
- dull_muldiv.cpp（运算强度削弱）
- rmblk.cpp （无用基本块删除）

后端优化位于 src/machine/opt 中， asm_opt.cpp 与 asm_opt.hpp 两个文件共同实现了图着色寄存器分配。

3.1 mem2reg

中间代码生成一章提到当时生成的 llvm ir 并不是真正的 ssa 形式，而是通过 memory 不是 SSA value 的特点开了一个后门，生成了 load store 形式的 llvm ir ，即所有局部 int 变量都使用 alloca 指令在栈上分配空间，计算前从内存 load 到寄存器，计算结束后 store 回内存。

频繁的 load/store 操作使得处理器效率十分低下，而且 load/store 破坏了变量的生存周期，非常不利于后续的代码优化。因此需要将局部 int 型变量通过 mem2reg 全部提升为虚拟寄存器，消除局部 int 型变量的 alloca/store/load ，并插入 phi 函数保持 ir 的 ssa 性质。mem2reg 一共包括 7 个步骤，分别为：

- 计算函数 CFG 图 （calFunctionCfg函数）
- 求每个基本块的严格支配 （calStrictlyDominated函数）
- 求每个基本块的直接支配 （calIdom函数）
- 求每个基本块的支配边界 （calDF函数）
- 计算变量的定义点 （calVarDefUse函数）
- 插入 phi 节点 （insertPhi函数）
- 变量重命名 （varRenaming函数）

3.1.1 求函数CFG图

控制流图（Control Flow Graph CFG）是指这样的有向图：每个基本块代表一个节点，如果基本块 i 可以跳转到基本块 j ，那么有向边 i->j 存在。

由于 llvm ir 已经划分了基本块，并且规定了基本块的最后一条指令一定是一条终结指令，因此 cfg 图很好建立，只需要遍历每个基本块，查看最后一条指令跳转到的基本块，并建立两个块之间的边。

函数cfg 图使用邻接表存储，定义在

```
opt.hpp
    class FunctionOptMsg
        std::unordered_map<int, std::vector<int>*> cfgGraph
```

3.1.2 求每个基本块的严格支配

首先给出支配与严格支配的定义：

- **支配**（dominate）：对于 CFG 中的节点 n_1 和 n_2，n_1 支配 n_2 当且仅当所有从入口节点到 n_2 的路径中都包含 n_1，即 n_1 是从入口节点到 n_2 的必经节点. 每个基本块都支配自身.
- **严格支配**（strictly dominate）：n_1 严格支配 n_2 当且仅当 n_1 支配 n_2 且 $n_1 \neq n_2$.

通俗的说，n_1 支配 n_2 即 n_1 是从入口节点到 n_2 的必经节点，因此为了求 n_1 支配的节点，我们可以将 n_1 从 CFG 图中删除，然后从入口节点开始 DFS 遍历。所有遍历到的节点都不是 n_1 的严格支配节点，因为 n_1 不是入口节点到这些节点的必经之路。而所有没有遍历到的节点除了 n_1 自身即为 n_1 严格支配的节点。

每个基本块所严格支配的节点存储在

```
opt.hpp
    class BasicBlockOptMsg
        std::unordered_map<int, BasicBlockOptMsg*> strictlyDominated
```

中，而严格支配该基本块的基本块存储在

```
opt.hpp
    class BasicBlockOptMsg
        std::unordered_map<int, BasicBlockOptMsg*> dominaterStrictly
```

3.1.3 求每个基本块的直接支配

- **直接支配者**（immediate dominator，idom）：节点 n 的直接支配者是离 n 最近的严格支配 n 的节点. 入口节点以外的节点都有直接支配者. 节点之间的直接支配关系可以形成一棵支配树（dominator tree）.

可以证明，对于一个节点 n，n 的直接支配者是所有支配 n 的节点中支配节点个数，即 dominate 集合最小的。因此我们只需要对每个节点 n，遍历严格支配 n 的所有节点 $m_1 \dots m_k$，找出 $m_1 \dots m_k$ 中支配集合最小的节点 m，即为 n 的直接支配者。

每个基本块所直接支配的节点存储在

```
opt.hpp
    class BasicBlockOptMsg
        std::vector<BasicBlockOptMsg*> idomds
```

3.1.4 求每个基本块的支配边界

- **支配边界**（dominance frontier）：节点 n 的支配边界 DF(n) = {x | n 支配 x 的前驱节点，n 不严格支配 x}

计算支配边界的伪代码如下：

Algorithm 3.2: Algorithm for computing the dominance frontier of each CFG node.

```
for (a, b) ∈ CFG edges do
    x ← a
    while x does not strictly dominate b do
        DF(x) ← DF(x) ∪ b
        x ← immediate dominator(x)
```

节点的支配边界存储在：

```
opt.hpp
    class BasicBlockOptMsg
            std::unordered_map<int, BasicBlockOptMsg*> DF;
```

3.1.5 计算变量的定义点

我们需要计算出对于一个局部 int 型变量，都在哪些基本块中对这个变量进行了定义。在 load/store 形式的 llvm ir 中，对局部 int 变量的定义通过 store 指令实现。问题在于如何识别某个 store 指令是一个对局部 int 变量的 store 指令，而不是对局部数组或者全局变量的。

在程序运行过程中对局部 int 型变量 store 前，一定会有 alloca 指令给该变量分配栈空间，因此我们可以按照前序遍历支配树的顺序遍历基本块，遇到 alloca 指令，并且该指令的类型是 i32 ，则一定是局部 int 型变量的 alloca ，于是记录 alloca 的地址寄存器到地址集合中，并用该地址代表该变量。遇到 store 指令，查看 store 的地址是否已经在地址集合中，若有则说明是对局部 int 变量的 store ，因此添加该基本块到定义该变量的基本块集合中。 前序遍历支配树的顺序代表了程序的某一种运行路径，因此不会出现对某一变量的 store 在 alloca 之前遍历到的情况发生。

变量定义点存储在：

```
opt.hpp
    class FunctionOptMsg
        /* key:局部int变量地址虚拟寄存器号, value:定义该变量的块列表 */
        std::unordered_map<int, std::set<BasicBlockOptMsg*>*> varAddrRegId2defBblk
```

3.1.6 插入phi节点

插入 phi 节点使用的是 ssa book 上的 ssa 构造标准算法，伪代码如下：

Algorithm 3.1: Standard algorithm for inserting ϕ-functions

```
1  for v: variable names in original program do
2      F ← {}                          ▷ set of basic blocks where φ is added
3      W ← {}                          ▷ set of basic blocks that contain definitions of v
4      for d ∈ Defs(v) do
5          let B be the basic block containing d
6          W ← W ∪ {B}
7      while W ≠ {} do
8          remove a basic block X from W
9          for Y: basic block ∈ DF(X) do
10             if Y ∉ F then
11                 add v ← φ(...) at entry of Y
12                 F ← F ∪ {Y}
13                 if Y ∉ Defs(v) then
14                     W ← W ∪ {Y}
```

之前已经计算出了伪代码中所描述的 `W` 集合，`DF` 集合。因此实现就是对该伪代码的非常简单的翻译。

注意到这里的 6 个步骤所做的事情仅仅是确定哪些基本块是必须要插入 `phi` 节点的，以及对什么变量插入 `phi` 节点。

3.1.7 变量重命名

遍历所有基本块，在每个基本块中遍历每条指令：

- 对于局部 `int` 型变量的 `alloca` 指令，直接删除。
- 对于局部 `int` 型变量的 `load` 指令，将所有其它指令中对该指令结果的使用替换为对该变量到达定义的使用。删除 `load` 指令。
- 对于局部 `int` 型变量的 `store` 指令，更新该变量到达定义为 `store` 的值，删除 `store` 指令。

遍历完一个基本块后，对于该基本块的所有后继基本块中的 `phi` 指令，将对应来自该基本块的值设为对应变量的到达定义。

3.1.8 解决mem2reg的丢失复制问题

在变量重命名过程中，对于局部 `int` 型变量的 `store` 指令，直接将该变量到达定义改为 `store` 的值，是会出一些问题的。对于源代码中 `a=b` 这样的拷贝语句，在 `mem2reg` 的变量重命名阶段会直接消除掉。而之后对于 `a` 的使用都会被替换为 `b` 的使用，因为 `ssa` 的性质，每个变量只定义一次，因此这样做没有问题。

但是在后端进行 `ssa` 形式下全局寄存器分配时，会将 `phi` 指令的参数和结果映射到同一个物理寄存器，以此来消除 `phi` 指令。此时一个变量只定义一次的性质就没有了。如果后续想使用 `a`，但 `b` 已经被改变了，那么 `a` 的值就不正确了。这就是丢失复制问题，即 `mem2reg` 的变量重命名和 `ssa` 形式下的寄存器分配共同导致了源代码中 `a=b` 这样的拷贝语句消失了。

对此，我的解决方法是在 mem2reg 变量重命名阶段保留拷贝语句。

3.2 死代码删除（DCE）

这个优化在 mem2reg 之后进行，即在标准的 ssa 形式 ir 上进行。

通俗的说，死代码删除即删除一个函数中不会对函数外界状态造成变化的语句。首先需要阐述关键操作的概念：

- 关键操作：如果函数中的一个操作会对函数外界产生影响，如产生函数返回值的操作，输入输出操作，或者更改函数外部可见的内存空间的操作，那么这个操作就是关键操作。由于死代码删除并不涉及函数间数据分析，因此所有调用函数的操作也被视为关键操作。
- 有用操作：对于关键操作使用的操作数，定义这些操作数的操作都是有用操作。对于有用操作使用的操作数，定义这些操作数的操作也是有用操作。

死代码删除的过程即为找到函数中所有的关键操作和有用操作，然后删除除这些外的所有操作。鉴于 ssa 一个变量只定义一次的性质，这个优化非常好做。需要注意的是数组需要视作一个整体。由于 llvm ir 使用 gep 指令获取数组地址的特点，比较难以判断一个 store 操作针对的到底是哪一个局部数组或者是全局数组。因此在实现中将所有 store 操作都视为了关键操作。这样的坏处是，如果一个局部数组没有进行 load 或者 load 出来的值没有被关键操作或有用操作使用，那么对这个数组的 store 操作都是无用操作，但是无法消除。

在 ir 的指令类中添加了一个是否为保留操作的标记：

```
class Instruction
    bool isCriticalInst;
```

优化流程如下：

- 首先遍历指令，建立虚拟寄存器号到定义指令的映射。
- 找到所有关键操作，将这些指令的 isCriticalInst 标记为 true 。然后将这些操作收录在操作集合中。
- 遍历操作集合，对某个操作，对所有该操作使用的值，将定义这些值的操作 isCriticalInst 标记为 true ，如果该操作没有被收录过，则收录在操作集合中。然后将已经查看过的操作从操作集合中删除，迭代直到操作集合为空。
- 遍历指令，删除所有 isCriticalInst 标记为 false 的操作。

3.3 全局常量传播（GCP）

这个优化在 mem2reg 之后进行，即在标准的 ssa 形式 ir 上进行。事实上我实现的全局常量传播可能更类似于复制传播+常数传播+常数合并。

通俗的说，全局常量传播即为如果我们可以确定一条指令的结果为编译期已知的常数，那么对于使用该指令结果的所有地方，将对虚拟寄存器的使用替换为对常数的使用，并删除该指令。这种替换可能会产生新的可以计算出常数结果的指令，因此这个操作会迭代进行直到不能更改任何操作为止。

该优化流程如下：

- 对变量定义点，如果定义点可以计算为已知常数，则删除定义指令，更新之后使用该定义的所有指令中使用值为该常数。
- 迭代直到不能更改任何指令为止。

事实上在实现中，除了会处理所有结果为已知常数的指令，如果某一条指令的结果可以确定为它的某个操作数，那么也会对该操作数进行传播。同时，对于全局int常量与全局数组常量，全局常量传播也对使用它们的地方进行了常量替换。

在全局常量传播完成后，对于条件为常量的 `br` 指令，会替换为无条件 `br` 指令。

对于可以计算出常量的指令与操作数组合，列表如下：

- 注：一条 `result = op op1 op2` 的指令进行的操作为 `result=op1 op op2`。

指令	操作数类型	替换为
add,sub,mul,sdiv,and,or,mod,icmp	op1和op2均为常数	返回常数经计算得到的对应结果
mul,sdiv,and,mod	只有op1是常数且op1等于0	返回常数0
add，or	只有op1是常数且op1等于0	返回op2
mul	只有op1是常数且op1等于1	返回op2
add,sub,or	只有op2是常数且op2等于0	返回op1
mul,and	只有op2是常数且op2等于0	返回常数0
mul,sdiv	只有op2是常数且op2等于1	返回op1
mod	只有op2是常数且op2等于1	返回常数0

3.4 运算强度削弱 （dull_muldiv）

运算强度削弱在全局常量传播之后进行，因此此时 `ir` 为标准的 `ssa` 形式，同时不会出现可以得出结果或结果就是某个操作数的乘除取模指令。

3.4.1 乘法优化

- 对一个常数乘以一个虚拟寄存器的乘法操作，如果常数是 2 的幂次，那么就把该指令替换为对虚拟寄存器的左移操作。
- 因为乘法的代价是 5 ，因此如果是 `x*3` ,可以将其替换为 `x+x+x` 。

3.4.2 除法优化

- 对于除数是 2 的幂次的除法指令，可以转换为对被除数的算术右移操作。
- 对于除数是任意常数，可以转化为乘法和右移，具体参考这篇文章[5] SuperSodaSea. 【编译笔记】变量除以常量的优化（一）——无符号除法. 知乎专栏. Published 2021. Accessed December 25, 2022. https://zhuanlan.zhihu.com/p/151038723

3.4.3 取模优化

- 对于模数是 2 的幂次的模运算，转换为先做除法再做减法，除法优化同上。

3.5 删除无用基本块 （rmblk）

`rmblk` 优化包括两个部分：

- 删除永远不会跳转到的基本块
- 优化掉只有一条无条件 `br` 指令的基本块

3.5.1 删除永远不会跳到的基本块

由于源代码自身的原因，可能会出现永远不会跳转到的基本块。全局常量传播之后，也可能会出现这样的基本块。对每个函数，初始化有效块集合为空，然后遍历所有的基本块的最后一条指令，即终结指令，如果该指令是 `br`，就将跳转目标块加入到有效块集合中。 遍历完之后删除所有不在有效块集合中的基本块。

这个过程是迭代进行的，直到不能再删除基本块为止。这是因为有可能出现只有基本块 `a` 指向基本块 `b`，而没有指向基本块 `a` 的块的情况，此时第一轮会删除基本块 `a`，第二轮会删除基本块 `b`。

3.5.2 优化掉只有一条无条件br指令的块

在生成中间代码的过程中，对于分支和循环的处理很容易出现如下这种基本块：

```
l1:
    br label l2
l2:
    br label l3
l3:
    br label l4
l4:
    ret void
```

这样在生成目标代码时会产生很多冗余的跳转指令，因此需要对其进行优化。优化的方式也很简单。对每个函数，遍历所有基本块，如果该基本块只有一条无条件 `br` 指令，那么就删除该块，并将所有其它指令中跳转到该块改为跳转到该块中无条件跳转指令跳转到的块。

3.6 图着色全局寄存器分配

图着色全局寄存器分配是我在后端做的唯一一个优化，但也是做的所有优化中效果最显著的，对于大部分辅助测试点，都消除了全部的局部 `int` 变量的 `lw/sw` 操作。

严谨起见，以下会以**虚拟寄存器**这一名词指代中间代码中使用的寄存器，而用**物理寄存器**指代 MIPS 体系结构中的一系列可用寄存器。**寄存器分配即为虚拟寄存器分配物理寄存器的过程。**

概括的说，我的图着色寄存器分配和一般的图着色寄存器分配有两个不同的地方：

- 1.我的图着色寄存器分配是在 ssa 形式上进行的，在对虚拟寄存器进行物理寄存器分配的同时也完成了消除 phi 指令。这样做的好处是规避了普通消除 phi 指令方法产生的多余复制指令与多余的基本块。
- 2.我没有按照通常的方法，按照虚拟寄存器的活动范围区分临时寄存器与全局寄存器，而是将所有的虚拟寄存器都拿来做全局分配。

寄存器分配需要如下 6 个步骤：

- 将 phi 节点中立即数参数全部改为虚拟寄存器 （phiImm2Reg函数）
- 求解各个基本块的定义和使用虚拟寄存器集合，然后求解各个基本块结束时仍然活跃的虚拟寄存器集合（calUseKill函数）（calLiveOut函数）
- 建立虚拟寄存器到活动范围的映射（initLR与calLR函数）
- 计算活动范围的冲突图 （calConflictGraph函数）
- 根据冲突图分配物理寄存器（allocReg函数）
- 插入保存恢复寄存器指令（insertSaveRecoverRegInst函数）

3.6.1 将phi节点中立即数参数全部改为虚拟寄存器

很容易出现这样的phi指令：

```
%reg21 = phi i32 [ 1, %l1 ], [ %reg18, %l10 ]
```

即如果从块 1 来，那么 reg21 需要取常数 1 ，如果从块 10 来，reg21 需要取 reg18 的值。这样是符合语义规范的，但是为了消除 phi 指令，必须将 phi 指令的参数和结果虚拟寄存器映射到同一个物理寄存器。而如果 phi 指令的参数中有常数，那么这个映射是无法完成的。

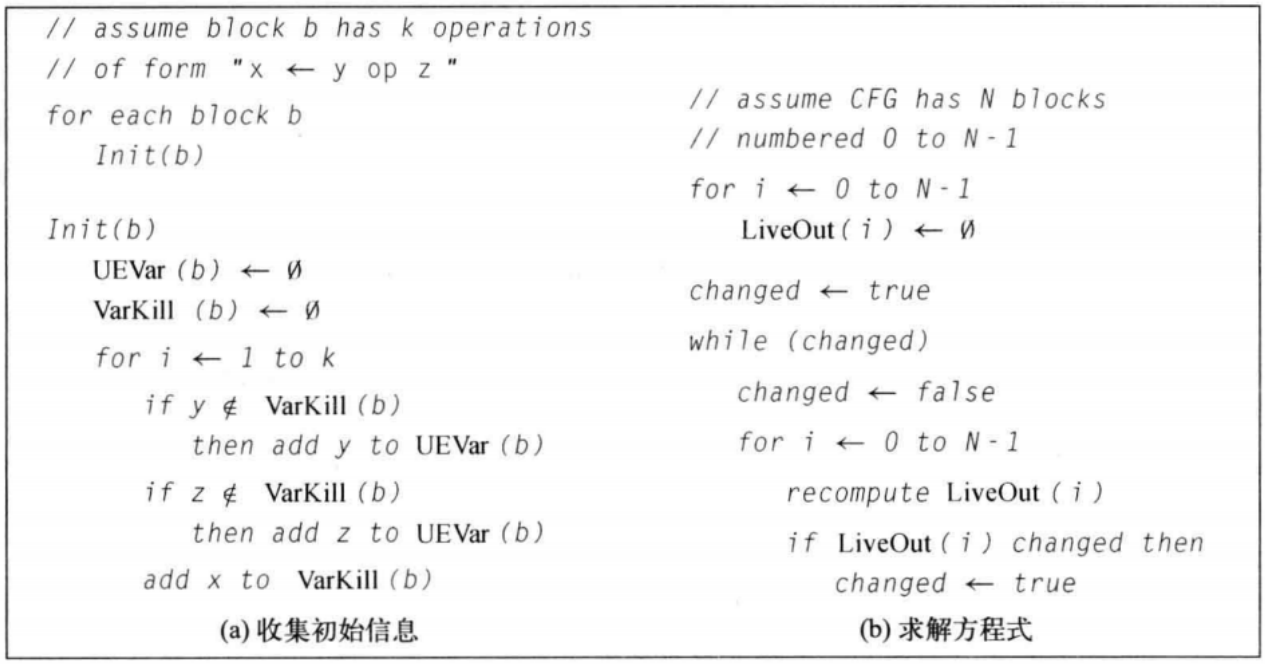
因此需要对 phi 指令参数中的常数做提升操作，提升为虚拟寄存器。对此我的解决办法是，在常数参数来自的基本块的终结指令前插入一条常数赋值操作，新建一个虚拟寄存器并将常数赋值给它，然后将 phi 指令中的常数替换为这个虚拟寄存器。

3.6.2 求解虚拟寄存器定义使用与 LiveOut

这一步即十分熟悉的活跃变量分析。只需要计算各个基本块结束时仍然活跃的虚拟寄存器集合即可，Liveout 的定义如下：

$$\mathrm{LiveOut}(n)=\bigcup_{m\in succ(n)}(\mathrm{UEVar}(m)\cup(\mathrm{LiveOut}(m)\cap\overline{\mathrm{VarKill}(m)}))$$

伪代码如下：



```
// assume block b has k operations
// of form "x ← y op z"
for each block b
   Init(b)

Init(b)
   UEVar(b) ← ∅
   VarKill(b) ← ∅
   for i ← 1 to k
      if y ∉ VarKill(b)
         then add y to UEVar(b)
      if z ∉ VarKill(b)
         then add z to UEVar(b)
      add x to VarKill(b)
```

(a) 收集初始信息

```
// assume CFG has N blocks
// numbered 0 to N-1
for i ← 0 to N-1
   LiveOut(i) ← ∅

changed ← true
while (changed)
   changed ← false
   for i ← 0 to N-1
      recompute LiveOut(i)
      if LiveOut(i) changed then
         changed ← true
```

(b) 求解方程式

3.6.3 建立虚拟寄存器到活动范围的映射

这一步建立虚拟寄存器到活动范围的映射，多个虚拟寄存器属于同一个活动范围代表这些虚拟寄存器**必须**分配同一个物理寄存器。在 ssa 形式中，每个虚拟寄存器只能定义一次，可以使用多次。而插入的 phi 函数记录了控制流图中不同代码路径上的不同定义到达同一个引用处的事实。如果一个操作引用了 phi 函数定义的虚拟寄存器，实际上会使用 phi 函数的某一个参数的值，具体是哪个参数则取决于控制流是如何到达 phi 函数的。所有这些定义应该驻留在同一个物理寄存器中，因而属于同一个活动范围。如此一来也就消除了 phi 函数。

这个映射使用并查集完成。初始时将所有虚拟寄存器都视为单独的一个个集合。然后遍历所有指令，遇到 phi 指令时，将 phi 指令的参数集合与结果集合取并集。

该并查集数据结构定义如下所示，使用 unordered_map 模拟一般并查集使用的数组。使用 findLR 获取虚拟寄存器对应的活动范围。使用 unionLR 合并活动范围。

```
asm_opt.hpp
    class AsmFuncOptMsg
        std::unordered_map<int,int> virtregId2LR;
        /* 传入虚拟寄存器号，返回对应的活动范围 */
        int findLR(int virtRegId);
        /* 传入两个虚拟寄存器号，将这两个虚拟寄存器对应的活动范围合并 */
        void unionLR(int i, int j);
```

3.6.4 计算活动范围冲突图

处于同一个活动范围的虚拟寄存器必须分配同一个物理寄存器，如果两个活动范围相交，那么它们对应的虚拟寄存器不能分配同一个物理寄存器。活动范围的冲突图反映了这些信息。

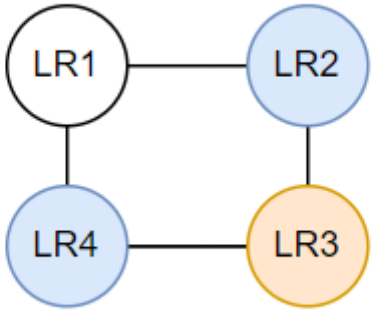
构建冲突图的算法**反向遍历**每个基本块，不断迭代 LiveNow 集合，对某一指令，将指令的计算结果的活动范围与 LiveNow 集合中的所有活动范围在冲突图中加边。然后更新 LiveNow 。伪代码如下：

```
for each LR_i
   create a node n_i ∈ N
for each basic block b
   LIVENOW ← LIVEOUT(b)
   for each operation op_n, op_n-1, op_n-2, ...op_1 in b
      with form op_i LR_a , LR_b ⇒ LR_c
      for each LR_i ∈ LIVENOW
         add (LR_c , LR_i ) to E
      remove LR_c from LIVENOW
      add LR_a and LR_b to LIVENOW
```

3.6.5 根据冲突图分配物理寄存器

分配算法使用的是编译理论所讲的方法，即先遍历冲突图，如果一个节点的度大于等于待分配物理寄存器个数，就将该节点标记为不分配物理寄存器。然后将节点从图中删除并加入一个队列。之后反向遍历队列，重建冲突图并着色。需要注意的是，按照该方法分配得到的结果有可能还能再分配物理寄存器。

举个例子，冲突图如下所示，如果一共有两个物理寄存器，那么分配方式可以如下图所示，`LR1` 被逐出了。然而此时 `LR1` 实际上可以变为橙色。



因此在一遍图着色分配完物理寄存器后，我会再遍历一遍冲突图，尝试对没有分配物理寄存器的节点进行寄存器分配。

3.6.6 插入保存恢复物理寄存器指令

这里我没有遵照 MIPS 调用规范，而是在进入函数时保存所有用到的物理寄存器，在退出函数时恢复所有用到的物理寄存器。

注意此时有可能还有一些指令中的寄存器是虚拟寄存器而非物理寄存器，这些寄存器会留待之后使用 简单寄存器分配方法解决。这一方法在第二章中有所描述。

3.7 面向调试的代码优化架构设计

在实现一系列优化的过程中，最让人头疼的应该是 debug 问题，或者说如何验证一个问题是某一个优化导致的，而不是另一个优化导致的。由于我采用的是 llvm ir，因此可以在做完一个优化后就导出文本形式的中间代码，然后使用 llvm 编译为目标代码，运行查验是否正确。下面是完整的用于中间代码优化的 optir 函数，使用宏定义控制是否打开中间代码输出：

```
inline void optir(Module* module)
{
  #ifdef ENABLE_Log
  std::string s;
  FILE* fp = fopen("base.ll", "w");
  s = module->toString();
  fprintf(fp, "%s", s.c_str());
  fclose(fp);
  #endif

  rmblk(module);
  Log("after rmblk1");
  #ifdef ENABLE_Log
  s = module->toString();
  fp = fopen("rmblk1.ll", "w");
  fprintf(fp, "%s", s.c_str());
  fclose(fp);
  #endif

  mem2reg(module);
  Log("after mem2reg");
  #ifdef ENABLE_Log
  s = module->toString();
  fp = fopen("mem2reg.ll", "w");
  fprintf(fp, "%s", s.c_str());
  fclose(fp);
  #endif

  globalConstantPropagation(module);
  Log("after globalConstantPropagation");
  #ifdef ENABLE_Log
  s = module->toString();
  fp = fopen("gcp.ll", "w");
  fprintf(fp, "%s", s.c_str());
  fclose(fp);
  #endif

  dullMulDiv(module);
```

```
  Log("after dullMulDiv");
  #ifdef ENABLE_Log
  s = module->toString();
  fp = fopen("dull.ll", "w");
  fprintf(fp, "%s", s.c_str());
  fclose(fp);
  #endif

  deadCodeElim(module);
  Log("after deadCodeElim");
  #ifdef ENABLE_Log
  s = module->toString();
  fp = fopen("dce.ll", "w");
  fprintf(fp, "%s", s.c_str());
  fclose(fp);
  #endif

}
```

4 调试工具

一个上万行代码的项目需要有比较方便的调试手段，更何况我使用的是 C With Class And STL 。代码中随处可见的裸指针使用使得段错误成为最常见的错误之一。我在 src/include/error.hpp 中定义了一系列的辅助工具用来帮助调试。

Log

log 是最方便最常用的调试手段。我对 fprintf 进行了一些封装，使得 log 可以自动在 stderr 输出 log 所在文件，行号，函数。

通过 ENABLE_Log 宏可以方便的打开或者关闭 Log 。

```
#define ANSI_BG_BLUE    "\33[1;44m"
#define ANSI_NONE       "\33[0m"
#define ANSI_FMT(str, fmt) fmt str ANSI_NONE

#define _Log(...) \
  do { \
    fprintf(stderr, __VA_ARGS__); \ /* 在标准错误流输出 */
  } while (0)

#ifdef ENABLE_Log       /* 如果定义了ENABLE_Log宏就输出log，否则不输出 */
#define Log(format, ...) \
    _Log(ANSI_FMT("[%s:%d %s] " format, ANSI_FG_BLUE) "\n", \
        __FILE__, __LINE__, __func__, ## __VA_ARGS__) /* 除了log输出的内容外，再输出log所在文件，行号，函数 */
#else
#define Log(format, ...)
#endif
```

panic

如果代码中的逻辑检测到出现了不可能的情况，说明一定出现了错误，此时可以调用 `panic` ，输出错误信息并直接终止程序。

```
#define Assert(cond, format, ...) \
  do { \
    if (!(cond)) { \
      fprintf(stderr, ANSI_FMT(format, ANSI_FG_RED) "\n", ##  __VA_ARGS__); \
      assert(cond); \
    } \
  } while (0)

#define panic(format, ...) Assert(0, format, ## __VA_ARGS__)
```

tprintf

该宏可以在输出内容前指定要输出的tab个数。非常适合打印符号表或者语法树。

```
#define tprintf(tabNum, ...) \
  do {  \
    printf("%*s", tabNum, ""); \
    printf(__VA_ARGS__); \
  } while (0)
```

4 总结

在写完这个编译器后，我深刻领会到了编译技术的博大精深。

对我而言最有难度是中间代码生成和代码优化两部分的实现。中间代码生成需要将语法树形式的源代码转变为线性的 `llvm ir` 。这是一个比较反直觉的过程，也是在我看来编译器的翻译功能真正体现的地方。生成中间代码这一部分我连续写了三天才最终搞定，可见难度之大。

另外一部分有难度的工作是代码优化。我做的最有难度的两个优化是 `mem2reg` 和图着色寄存器分配。这两个优化每一个都要分为差不多十个小的步骤，每一个步骤都很有可能出现难以察觉的错误。往往debug的时间会大于写代码实现的时间。但是通过这个过程，我也深刻理解了静态单赋值这一现代编译器中最重要的概念之一，并亲手实现了它。

在我实现编译器的过程中，我越来越深刻认识到编译器是一个软件工程问题。代码优化好不好做，并不取决于优化算法的难度或者工作量，而是取决于对中间代码数据结构的设计，整个优化代码架构的组织。好的代码结构可以避免很多的 `bug` ，并且易于修改，易于增加功能。

每一个阶段课程组都整理了大量的测试点供我们测试，大大缩短了 `debug` 的时间。评测机可以无限提交，也是非常好的体验。不过还有几点建议：

- 代码优化的测试部分建议多加一些正常的代码。正常的代码即真正的工程中产出的代码。现在的优化测试点，一方面比较少，可能不能对所作的优化有一个比较全面的评判。另一方面，往往做了一个优化，某个测试点的分数就提高了很多，显然这个测试点是特别构造出来用于测试某个优化的，但是实际工程中很少有人会写出这样的代

码。这样就变成了谁做的优化更多的比拼，往往不需要细致对优化进行调优。这感觉不是一个学习编译器优化的正确方式。

- 建议禁止使用 JAVA 实现编译器，而是使用 C/C++ 实现。一个不能与内存展开直接对话的语言是没有灵魂的。
- 建议删除竞速，而是设置 baseline 。只要优化达到一定分数就可以获得满分。事实上学习各种各样的优化算法是有趣的一件事，但当最终目标变成取得更高排名时，一切都变得索然无味了。

最后，当我写完这篇一万多字的设计文档，深感相关知识体系的庞大，以及编译相关理论的学无止境。感谢课程组对编译实验的建设，希望编译实验能越来越好！

参考文献

[1] Keith Cooper, Linda Torczon: Engineering a Compiler, 3rd Edition - August 20, 2022

[2] Lots of authors: Static Single Assignment Book, Wednesday 30^{th}May, 2018

[3] 实验简介 · GitBook. Github.io. Published 2022. Accessed December 25, 2022. https://buaa-se-compiling.github.io/miniSysY-tutorial/

[4] LLVM Language Reference Manual — LLVM 16.0.0git documentation. Llvm.org. Published 2022. Accessed December 25, 2022. https://llvm.org/docs/LangRef.html

[5] SuperSodaSea. 【编译笔记】变量除以常量的优化（一）——无符号除法. 知乎专栏. Published 2021. Accessed December 25, 2022. https://zhuanlan.zhihu.com/p/151038723

[6] 编译器是如何实现32位整型的常量整数除法优化的？[C/C++] - shines77 - 博客园. Cnblogs.com. Published 2015. Accessed December 25, 2022. https://www.cnblogs.com/shines77/p/4189074.html