南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理实验报告

了解编译器及 LLVM IR 编程

姓名：莫骐骥

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 10 月 2 日

摘要

在本次实验中，我使用现有的编译器简单的 C 程序上测试，粗浅地了解了程序从 c 代码到可执行文件都经历了什么。

关键字：编译器　LLVM

目录

一、 引言

这次预备实验旨在让我们对程序的编译、链接等流程有一个基本的认识，重点在于了解各个阶段的任务，以及 llvm ir 的简单编写，为后续开发打下基础。

二、 预处理器做了什么

(一) 代码样例的编写

一段 C 代码

```c
#include <stdio.h>
#ifndef a
#define a (x + 1)
#else
int is_exist = 101;
#endif

const int N = 100;
int x = 2;

void func_a(int n){//函数a: 输出局部变量x + 1 + n的结果
    int x = 1;
    printf("%d\n", a + n);
}

void func_b(){//函数b: 输出全局变量x + 1 + N 的结果
    printf("%d\n", a + N);
}

int main(){
    func_a(10);
    func_b();
    return 0;
}
```


上面这段代码，包含了标准输入输出库 *stdio.h*，定义宏 $a\ (x+1)$, 全局变量等 C 语言特性，具体由下表所列出：

表 1: 代码内容归纳

头文件包含	stdio.h
宏定义	a (x + 1)
全局常量	N(100)
全局变量	x
函数	$func_a, func_b$

(二) 结果分析

预处理阶段在正式的编译阶段之前进，根据已放置在文件中的预处理指令来修改源文件的内容。主要处理如 $\#include\ "filename"$,$\#include\ <stdlib.h>$ $\#define$ $\#ifndef$ 之类以 # 开头的命令，将对应的文件或后面定义的变量替换到目标文件中。

例如对于上一小节的 C 文件，其中第一行代码是包含 stdio.h 头文件，因此直接将 stdio.h 中的内容替换在 main.i 文件中。经过预处理，文件大小由 388 字节扩大到 16496 字节，主要原因就是包含了这个库。

除去头文件的部分，其余部分预处理后的代码如下所示:

预处理之后的 C 代码

```c
const int N = 100;
int x = 2;

void func_a(int n){
    int x = 1;
    printf("%d\n", (x + 1) + n);
}

void func_b(){
    printf("%d\n", (x + 1) + N);
}

int main(){
    func_a(10);
    func_b();
    return 0;
}
```

由于文件中 $\#ifndef\ a$ 之前没有对 a 进行定义，因此将 a 定义为（x + 1），并直接在 $func_a$ $func_b$ 函数中将 a 替换:

预处理变量 a 后的 $func_a$ $func_b$ 函数代码

```c
void func_a(int n){
    int x = 1;
    printf("%d\n", (x + 1) + n);
}

void func_b(){
    printf("%d\n", (x + 1) + N);
}
```

此外,$\#else$ 部分由于没有没有执行该分支,在预处理之后这部分代码就消失了:即 $intis_exist = 101$; 不会出现在预处理后文件的变量声明中。

三、 编译器做了什么

编译过程具体来说分六步，分别为：词法分析、语法分析、语义分析、中间代码生成、代码优化、代码生成。

(一) 词法分析

词法分析是编译器的第一个步骤，也称为扫描。顾名思义，词法分析器顺序地读入源程序的字符流，并将它们组织成有意义的词素序列。对于每一个词素。词法分析器产生如 $< token - name, attribute - value >$ 的词法单元。

写一个简单的 C 程序如下所示，不包含任何头文件：

$get_token.c$

```c
int main() {
    int a, b;
    a += b;
    return 0;
}
```

通过以下命令获得 token 序列

```
clang -E -Xclang -dump-tokens get_token.c
```

获得 token 序列如下：

int	'int'	[StartOfLine]	Loc=<token.c:1:1>	
identifier	'main'	[LeadingSpace]	Loc=<token.c:1:5>	
l_paren	'('	Loc=<token.c:1:9>		
r_paren	')'	Loc=<token.c:1:10>		
l_brace	'{'	[LeadingSpace]	Loc=<token.c:1:12>	
int	'int'	[StartOfLine]	[LeadingSpace]	Loc=<token.c:2:5>
identifier	'a'	[LeadingSpace]	Loc=<token.c:2:9>	
comma	','	Loc=<token.c:2:10>		
identifier	'b'	[LeadingSpace]	Loc=<token.c:2:12>	
semi	';'	Loc=<token.c:2:13>		
identifier	'a'	[StartOfLine]	[LeadingSpace]	Loc=<token.c:3:5>
plusequal	'+='	[LeadingSpace]	Loc=<token.c:3:7>	
identifier	'b'	[LeadingSpace]	Loc=<token.c:3:10>	
semi	';'	Loc=<token.c:3:11>		
return	'return'	[StartOfLine]	[LeadingSpace]	Loc=<token.c:4:5>
numeric_constant	'0'	[LeadingSpace]	Loc=<token.c:4:12>	
semi	';'	Loc=<token.c:4:13>		
r_brace	'}'	[StartOfLine]	Loc=<token.c:5:1>	
eof	''	Loc=<token.c:5:2>		

表 2: *get_token.c* 词法分析得到的 token 序列

可以看出，通过词法分析我们可以得到每一个词素的内容、类型、在程序中的位置的信息。例如: 程序中的第一个词素 int，被识别为 C 语言中的保留字 int，同时它也是一行的开始，因此用 $[StartOfLine]$ 标注出，它的首字符是第一行第一个字符，因此用 $Loc = < token.c : 1 : 1 >$ 标注出。词素 a 被识别为一种标识符，因此用 $identifier$ 标注出，它的前面有前导空格，因此用 $[LeadingSpace]$ 标注出，它是第二行的第九个字符，因此用 $< Loc = token.c : 2 : 9 >$ 标注出。类似地，上面的表格也体现了在词法分析后，其他保留字 $return$、运算符 $+=$、标识符 b、常数 0、界符 { } 等词法单元。

(二) 语法分析

将词法分析生成的词法单元来构建抽象语法树。对于 LLVM，可通过如下指令获得相应的抽象语法树：$clang - E - Xclang - ast - dump main.c$。

我直接用上一小节的 get_token.c 文件来观察输出的抽象语法树:

```
TranslationUnitDecl 0x1a5c5e8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x1a5ce80 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x1a5cb80 '__int128'
|-TypedefDecl 0x1a5cef0 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x1a5cba0 'unsigned __int128'
|-TypedefDecl 0x1a5d1f8 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
| `-RecordType 0x1a5cfd0 'struct __NSConstantString_tag'
|   `-Record 0x1a5cf48 '__NSConstantString_tag'
|-TypedefDecl 0x1a5d290 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x1a5d250 'char *'
|   `-BuiltinType 0x1a5c680 'char'
|-TypedefDecl 0x1a5d588 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag [1]'
| `-ConstantArrayType 0x1a5d530 'struct __va_list_tag [1]' 1
|   `-RecordType 0x1a5d370 'struct __va_list_tag'
|     `-Record 0x1a5d2e8 '__va_list_tag'
`-FunctionDecl 0x1abc090 <token.c:1:1, line:5:1> line:1:5 main 'int ()'
  `-CompoundStmt 0x1abc360 <col:12, line:5:1>
    |-DeclStmt 0x1abc290 <line:2:5, col:13>
    | |-VarDecl 0x1abc190 <col:5, col:9> col:9 used a 'int'
    | `-VarDecl 0x1abc210 <col:5, col:12> col:12 used b 'int'
    |-CompoundAssignOperator 0x1abc300 <line:3:5, col:10> 'int' '+=' ComputeLHSTy='int' ComputeResultTy='int'
    | |-DeclRefExpr 0x1abc2a8 <col:5> 'int' lvalue Var 0x1abc190 'a' 'int'
    | `-ImplicitCastExpr 0x1abc2e8 <col:10> 'int' <LValueToRValue>
    |   `-DeclRefExpr 0x1abc2c8 <col:10> 'int' lvalue Var 0x1abc210 'b' 'int'
    `-ReturnStmt 0x1abc350 <line:4:5, col:12>
      `-IntegerLiteral 0x1abc330 <col:12> 'int' 0
```

图 1: get_token.c 语法分析抽象语法树

图中的抽象语法树包含了函数声明和变量声明两大部分。函数声明只有一个即主函数，变量声明有 $\_int128$、$char*$ 等。主函数声明子树部分又有三个分支：declare statement（声明语句）、Compound Assign Operator（复合赋值运算符）语句、return statement（返回语句）。声明语句子树包含变量 a、b，复合赋值语句包含变量 a、b 和运算符“+=”，返回语句只有常数 0。暂时没有找到好的办法对 ast 进行可视化处理。

(三) 语义分析

语义分析使用语法分析得到的抽象语法树和符号表中的信息检查源程序中是否和语言定义的语义一致。主要工作包括类型检查、运算符的运算分量匹配等等。

(四) 中间代码生成

在源程序的语法分析和语义分析之后，编译器会生成一个明确的低级的或类机器语言的中间表示，我们可以把这个表示看做是某个抽象机器的语言，也即中间代码生成。

写一个不包含任何头文件的 c 代码，否则生成的文件太长:

$main.c$

```
const int n = 20;
int main() {
    int a, b, i, t, n;
    a = 0;
    b = 1;
    i = 1;
    while(i < n) {
        t = b;
        b = a + b;
        a = t;
        i = i + 1;
    }
    return 0;
}
```

观察一下 gcc 编译器获得的中间代码生成的多个阶段的输出:

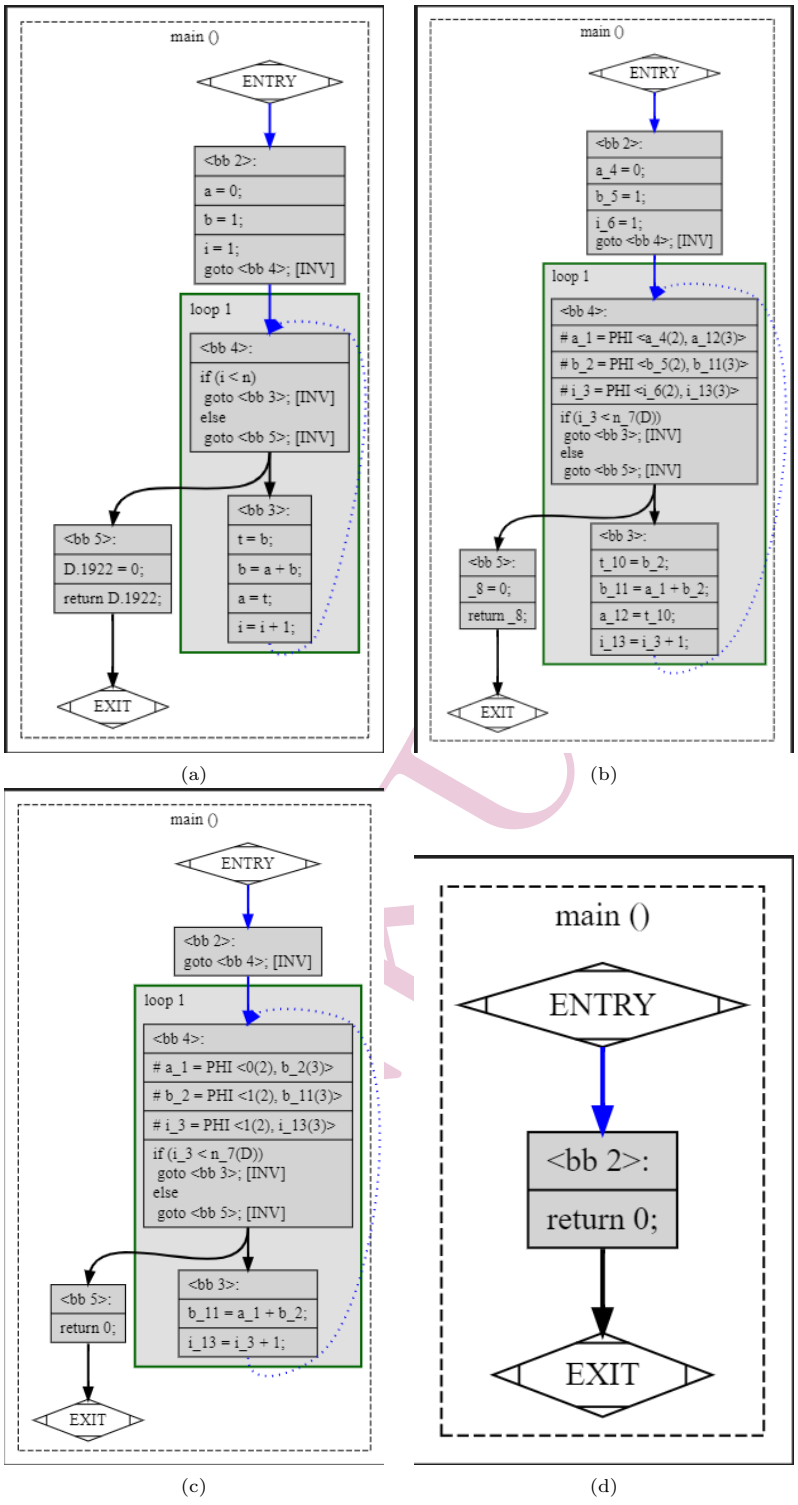

图 2: CFG

执行 $clang -Xclang -disable -O0 -optnone -S -emit -llvm main.c$, 生成未优化的 LLVM IR:

main.ll

```
; ModuleID = 'main.c'
source_filename = "main.c"
```

```
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
    :16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@n = dso_local constant i32 20, align 4

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  store i32 1, i32* %3, align 4
  store i32 1, i32* %4, align 4
  br label %7

7:                                                ; preds = %11, %0
  %8 = load i32, i32* %4, align 4
  %9 = load i32, i32* %6, align 4
  %10 = icmp slt i32 %8, %9
  br i1 %10, label %11, label %19

11:                                               ; preds = %7
  %12 = load i32, i32* %3, align 4
  store i32 %12, i32* %5, align 4
  %13 = load i32, i32* %2, align 4
  %14 = load i32, i32* %3, align 4
  %15 = add nsw i32 %13, %14
  store i32 %15, i32* %3, align 4
  %16 = load i32, i32* %5, align 4
  store i32 %16, i32* %2, align 4
  %17 = load i32, i32* %4, align 4
  %18 = add nsw i32 %17, 1
  store i32 %18, i32* %4, align 4
  br label %7

19:                                               ; preds = %7
  ret i32 0
}

attributes #0 = { noinline nounwind uwtable "correctly-rounded-divide-sqrt-fp
    -math"="false" "disable-tail-calls"="false" "frame-pointer"="all" "less-
    precise-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-math"="
    false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-
```

```
    zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-buffer
    -size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse
    ,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}
```

（五） 代码优化

LLVM 中的优化和分析被组织成 pass 结构。通过组合不同 pass 来完成不同的优化算法。LLVM 中有以下重要的优化 pass。adce: 激进的死代码删除。bb-vectorize: 基本块向量化。const-prop: 常量传播优化。dce: 死代码消除。globaldce: 全局死代码消除。deadargelim: 死变量消除。globalopt: 全局优化。gvn: 全局值命名。inline: 内联函数优化。instcombine: 冗余指令组合。licm: 循环不变代码提升。loweratomic: 把原子操作 Intrinsic 函数降低成非原子操作。lowerinvoke: 降低函数调用。lowerswitch: 降低 switch 语句，转化成分支语句。mem2reg: 把某块内存放在寄存器里。memcpyopt: 内存拷贝优化。simplifycfg: 控制流简化。tailcallelim: 尾调用消除。

我从中选取几个易实现的 pass 优化进行尝试，首先把 C 代码改一下：

```c
const int n = 20;
int global_no_use; //没有用到的全局变量

void no_use_func() {  //没有调用的函数
    int a = 1;
}

int func_a() {
    char local_no_use = 'a'; //没有用到的局部变量
    for(int i = 0; i < 10; i ++) continue;  //没有意义的for循环
    int x = 14;
    int y = 7 - x / 2;
    return y * (28 / x + 2);  //常量传播
}

int main() {
    int a, b, i, t, n;
    a = 0;
    b = 1;
    i = 1;
    while(i < n) {
        t = b;
        b = a + b;
        a = t;
        i = i + 1;
    }
    func_a();
```

```
    return 0;
}
```

使用指令 $clang - S - emit - llvm - Xclang - disable - O0 - optnonemain.cclang - S - emit - llvm - Xclang - disable - O0 - optnonemain.c$ 生成没有优化的中间代码。

main.ll

```
@n = dso_local constant i32 20, align 4
@global_no_use = common dso_local global i32 0, align 4

; Function Attrs: noinline nounwind uwtable
define dso_local void @no_use_func() #0 {
  %1 = alloca i32, align 4
  store i32 1, i32* %1, align 4
  ret void
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @func_a() #0 {
  %1 = alloca i8, align 1
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i8 97, i8* %1, align 1
  store i32 0, i32* %2, align 4
  br label %5

5:                                                ; preds = %9, %0
  %6 = load i32, i32* %2, align 4
  %7 = icmp slt i32 %6, 10
  br i1 %7, label %8, label %12

8:                                                ; preds = %5
  br label %9

9:                                                ; preds = %8
  %10 = load i32, i32* %2, align 4
  %11 = add nsw i32 %10, 1
  store i32 %11, i32* %2, align 4
  br label %5

12:                                               ; preds = %5
  store i32 14, i32* %3, align 4
  %13 = load i32, i32* %3, align 4
  %14 = sdiv i32 %13, 2
  %15 = sub nsw i32 7, %14
  store i32 %15, i32* %4, align 4
  %16 = load i32, i32* %4, align 4
  %17 = load i32, i32* %3, align 4
```

```
  %18 = sdiv i32 28, %17
  %19 = add nsw i32 %18, 2
  %20 = mul nsw i32 %16, %19
  ret i32 %20
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  store i32 1, i32* %3, align 4
  store i32 1, i32* %4, align 4
  br label %7

7:                                                ; preds = %11, %0
  %8 = load i32, i32* %4, align 4
  %9 = load i32, i32* %6, align 4
  %10 = icmp slt i32 %8, %9
  br i1 %10, label %11, label %19

11:                                               ; preds = %7
  %12 = load i32, i32* %3, align 4
  store i32 %12, i32* %5, align 4
  %13 = load i32, i32* %2, align 4
  %14 = load i32, i32* %3, align 4
  %15 = add nsw i32 %13, %14
  store i32 %15, i32* %3, align 4
  %16 = load i32, i32* %5, align 4
  store i32 %16, i32* %2, align 4
  %17 = load i32, i32* %4, align 4
  %18 = add nsw i32 %17, 1
  store i32 %18, i32* %4, align 4
  br label %7

19:                                               ; preds = %7
  %20 = call i32 @func_a()
  ret i32 0
}
```

先进行 mem2reg 优化：

mem2reg.ll

```
@n = dso_local constant i32 20, align 4
```

```
@global_no_use = common dso_local global i32 0, align 4

; Function Attrs: noinline nounwind uwtable
define dso_local void @no_use_func() #0 {
  ret void
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @func_a() #0 {
  br label %1

1:                                                ; preds = %4, %0
  %.0 = phi i32 [ 0, %0 ], [ %5, %4 ]
  %2 = icmp slt i32 %.0, 10
  br i1 %2, label %3, label %6

3:                                                ; preds = %1
  br label %4

4:                                                ; preds = %3
  %5 = add nsw i32 %.0, 1
  br label %1

6:                                                ; preds = %1
  %7 = sdiv i32 14, 2
  %8 = sub nsw i32 7, %7
  %9 = sdiv i32 28, 14
  %10 = add nsw i32 %9, 2
  %11 = mul nsw i32 %8, %10
  ret i32 %11
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  br label %1

1:                                                ; preds = %3, %0
  %.02 = phi i32 [ 1, %0 ], [ %4, %3 ]
  %.01 = phi i32 [ 0, %0 ], [ %.02, %3 ]
  %.0 = phi i32 [ 1, %0 ], [ %5, %3 ]
  %2 = icmp slt i32 %.0, undef
  br i1 %2, label %3, label %6

3:                                                ; preds = %1
  %4 = add nsw i32 %.01, %.02
  %5 = add nsw i32 %.0, 1
  br label %1

```

```
6:                                                ; preds = %1
  %7 = call i32 @func_a()
  ret i32 0
}
}
```

通过把内存放入寄存器中，中间代码减少了很多很多 load、store 指令，且把没有占据内存空间必要的局部变量消除了。

simplifycfg 控制流优化：

simplifycfg.ll

```
@n = dso_local constant i32 20, align 4
@global_no_use = common dso_local global i32 0, align 4

; Function Attrs: noinline nounwind uwtable
define dso_local void @no_use_func() #0 {
  ret void
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @func_a() #0 {
  br label %1

1:                                                ; preds = %3, %0
  %.0 = phi i32 [ 0, %0 ], [ %4, %3 ]
  %2 = icmp slt i32 %.0, 10
  br i1 %2, label %3, label %5

3:                                                ; preds = %1
  %4 = add nsw i32 %.0, 1
  br label %1

5:                                                ; preds = %1
  %6 = sdiv i32 14, 2
  %7 = sub nsw i32 7, %6
  %8 = sdiv i32 28, 14
  %9 = add nsw i32 %8, 2
  %10 = mul nsw i32 %7, %9
  ret i32 %10
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  br label %1

1:                                                ; preds = %3, %0
  %.02 = phi i32 [ 1, %0 ], [ %4, %3 ]
  %.01 = phi i32 [ 0, %0 ], [ %.02, %3 ]
```

```
  %.0 = phi i32 [ 1, %0 ], [ %5, %3 ]
  %2 = icmp slt i32 %.0, undef
  br i1 %2, label %3, label %6

3:                                                ; preds = %1
  %4 = add nsw i32 %.01, %.02
  %5 = add nsw i32 %.0, 1
  br label %1

6:                                                ; preds = %1
  %7 = call i32 @func_a()
  ret i32 0
}
```

通过控制流优化，将被调用的函数的块由 4 个减为 3 个。使用 instcombine 优化选项，将一组由数据流连接的指令重写为更高效的形式：

instcombine

```
@n = dso_local constant i32 20, align 4
@global_no_use = common dso_local global i32 0, align 4

; Function Attrs: noinline nounwind uwtable
define dso_local void @no_use_func() #0 {
  ret void
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @func_a() #0 {
  br label %1

1:                                                ; preds = %3, %0
  %.0 = phi i32 [ 0, %0 ], [ %4, %3 ]
  %2 = icmp ult i32 %.0, 10
  br i1 %2, label %3, label %5

3:                                                ; preds = %1
  %4 = add nuw nsw i32 %.0, 1
  br label %1

5:                                                ; preds = %1
  ret i32 0
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  br label %1

1:                                                ; preds = %2, %0
  br i1 false, label %2, label %3
```

```
32
33 2:                                                ; preds = %1
34   br label %1
35
36 3:                                                ; preds = %1
37   %4 = call i32 @func_a()
38   ret i32 0
39 }
```

通过优化，funca 中 x=14，y 经过计算后等于 0，返回值经计算后等于 0，因此在中间代码中直接消除了对 x、y 的声明和定义，直接返回 0，在 main 函数中，也将计算 fibonacci 数列的部分消除了（可能是因为只计算但没有调用输出），只留下对 funca 函数的调用。

还有很多 pass，仅仅是这三个优化步骤，就已经将中间代码文件缩短了一半左右。

(六) 代码生成

代码生成器以源程序的中间表示形式作为输入并将其映射到目标语言汇编代码。执行文档中对应的命令可以分别得到 X86、arm、LLVM 的汇编代码，其中生成的 LLVM 汇编代码规模最大，如下所示：

main.S

```
1         .text
2         .file   "main.c"
3         .globl  main                    # -- Begin function main
4         .p2align        4, 0x90
5         .type   main,@function
6 main:                                   # @main
7         .cfi_startproc
8 # %bb.0:
9         pushq   %rbp
10        .cfi_def_cfa_offset 16
11        .cfi_offset %rbp, -16
12        movq    %rsp, %rbp
13        .cfi_def_cfa_register %rbp
14        movl    $0, -24(%rbp)
15        movl    $0, -12(%rbp)
16        movl    $1, -8(%rbp)
17        movl    $1, -4(%rbp)
18 .LBB0_1:                                # =>This Inner Loop Header: Depth=1
19        movl    -4(%rbp), %eax
20        cmpl    -20(%rbp), %eax
21        jge     .LBB0_3
22 # %bb.2:                                #   in Loop: Header=BB0_1 Depth=1
23        movl    -8(%rbp), %eax
24        movl    %eax, -16(%rbp)
25        movl    -12(%rbp), %eax
26        addl    -8(%rbp), %eax
27        movl    %eax, -8(%rbp)
```

```
28          movl    -16(%rbp), %eax
29          movl    %eax, -12(%rbp)
30          movl    -4(%rbp), %eax
31          addl    $1, %eax
32          movl    %eax, -4(%rbp)
33          jmp     .LBB0_1
34  .LBB0_3:
35          xorl    %eax, %eax
36          popq    %rbp
37          .cfi_def_cfa %rsp, 8
38          retq
39  .Lfunc_end0:
40          .size   main, .Lfunc_end0-main
41          .cfi_endproc
42                                          # -- End function
43          .type   n,@object               # @n
44          .section        .rodata,"a",@progbits
45          .globl  n
46          .p2align        2
47  n:
48          .long   20                      # 0x14
49          .size   n, 4
50
51          .ident  "clang version 10.0.0-4ubuntu1 "
52          .section        ".note.GNU-stack","",@progbits
```

四、 汇编器做了什么

汇编器是将汇编语言翻译为机器语言的程序，以汇编语言作为输入，生成可重定位的机器码。一个汇编器的执行流程如下：

- 读下一行汇编语言指令
- 把这条汇编语言指令拆开成不同小段
- 查汇编代码和机器代码的对照表，将每小段的汇编代码转成 0101 码
- 组合每个 0101 码成一行完整的机器语言
- 输出这一行机器语言

通常汇编器从汇编程序文件，过滤掉注释行，读出每一行汇编命令。然后放入一个类似字符串中，感觉有点像词法分析时候输入的字符流。通过空格、逗号等，将字符串分成三部分。通过查找汇编语言和机器语言的对应表，完成每一段的翻译。将每一段机器码组合，有时候根据汇编语言的细节，也会加入一些额外的码。最后将它们输出为可重定位目标文件和共享目标文件。可重定位目标文件包含未被链接的符号代码、数据区域，可以被链接器链接成可执行目标文件；共享目标文件是一类特殊的可重定位目标文件，可以被程序在运行时动态的加载进内存。在运行时确定内存符号位置的文件。

上面的流程是分两趟来进行的，因为为了翻译汇编语言程序，汇编器需要知道两件事情：每个助记符的操作码和每个标号的地址。操作码信息以表形式内置在汇编器中，这个表给出对应每个助记符的操作码。编器第一遍扫描源程序只是为了确定每个标号的地址。只要在某行的开始发现标号，汇编器就将该标号及其地址输入到符号表中。在第一遍扫描结束时，符号表将包含程序中使用的所有标号及其地址。汇编器然后执行第二遍扫描，使用操作码表和符号表中的信息汇编每条指令。

生成的可重定位目标文件以 16 字节的序列称为 ELF 头的区域开始，这个区域记录了该文件使用的字长和端序信息。剩余的空间记录了能够被链接器识别的信息，包括 ELF 头的大小，目标文件的类型，程序段的数量和段头表的偏移地址和目标机器信息等等。ELF 文件以段头表作为结束，描述了文件中每个段的大小和偏移。ELF 头和段头表中间夹的部分由多个程序段组成。

汇编器生成的部分结果

```
main.o:     文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <func_a>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 20             sub    $0x20,%rsp
   c:   89 7d ec                mov    %edi,-0x14(%rbp)
   f:   c7 45 fc 01 00 00 00    movl   $0x1,-0x4(%rbp)
  16:   8b 45 fc                mov    -0x4(%rbp),%eax
  19:   8d 50 01                lea    0x1(%rax),%edx
  1c:   8b 45 ec                mov    -0x14(%rbp),%eax
  1f:   01 d0                   add    %edx,%eax
  21:   89 c6                   mov    %eax,%esi
  23:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 2a <func_a+0
      x2a>
  2a:   b8 00 00 00 00          mov    $0x0,%eax
  2f:   e8 00 00 00 00          callq  34 <func_a+0x34>
  34:   90                      nop
  35:   c9                      leaveq
  36:   c3                      retq

00000000000000dd <main>:
  dd:   f3 0f 1e fa             endbr64
  e1:   55                      push   %rbp
  e2:   48 89 e5                mov    %rsp,%rbp
  e5:   bf 0a 00 00 00          mov    $0xa,%edi
  ea:   e8 00 00 00 00          callq  ef <main+0x12>
  ef:   b8 00 00 00 00          mov    $0x0,%eax
  f4:   e8 00 00 00 00          callq  f9 <main+0x1c>
  f9:   bf 14 00 00 00          mov    $0x14,%edi
  fe:   e8 00 00 00 00          callq  103 <main+0x26>
 103:   89 c6                   mov    %eax,%esi
```

```
105:   48 8d 3d 00 00 00 00     lea    0x0(%rip),%rdi        # 10c <main+0x2f
    >
10c:   b8 00 00 00 00           mov    $0x0,%eax
111:   e8 00 00 00 00           callq  116 <main+0x39>
116:   b8 00 00 00 00           mov    $0x0,%eax
11b:   5d                       pop    %rbp
11c:   c3                       retq
```

五、 链接器做了什么

由汇编程序生成的目标文件不能够直接执行。大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真正在机器上运行的代码。在上一小节中，汇编器生成的可重定位机器码包含未被链接的符号代码、数据区域等.

汇编后符号表

```
0000000000000000 T func_a
0000000000000037 T func_b
0000000000000065 T func_c
                 U _GLOBAL_OFFSET_TABLE_
00000000000000dd T main
0000000000000000 R N
                 U printf
0000000000000000 D x
```

文件中包含了 abc 三个函数、printf 函数、主函数、全局常量 N、全局变量 x 的地址。使用链接器进行链接之后，文件中的符号如下所示：

（动态）链接后符号表

```
0000000000004014 B __bss_start
0000000000004014 b completed.8061
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000004000 D __data_start
0000000000004000 W data_start
0000000000001090 t deregister_tm_clones
0000000000001100 t __do_global_dtors_aux
0000000000003dc0 d __do_global_dtors_aux_fini_array_entry
0000000000004008 D __dso_handle
0000000000003dc8 d _DYNAMIC
0000000000004014 D _edata
0000000000004018 B _end
00000000000012e8 T _fini
0000000000001140 t frame_dummy
0000000000003db8 d __frame_dummy_init_array_entry
00000000000021d4 r __FRAME_END__
0000000000001149 T func_a
0000000000001180 T func_b
```

```
00000000000011ae T func_c
0000000000003fb8 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000002014 r __GNU_EH_FRAME_HDR
0000000000001000 t _init
0000000000003dc0 d __init_array_end
0000000000003db8 d __init_array_start
......
```

在链接时将链接方式修改为静态链接，重新生成可执行文件的反汇编文件，可以看到仅仅是符号表已经有上千行。不同于动态链接使动态库中的函数在程序运行后才被用到，静态链接则在运行前就将所需函数合并。静态链接生成一个完全链接的、可以加载和运行的可执行目标文件作为输出，因此在执行的时候速度快，但麻烦在于一方面目标文件很大，而且当链接文件被修改时，目标文件也需要修改。根据查询资料，一般来说，动态链接比静态链接损失了百分之五的执行速度，这是在可接受范围内的，因此往往选择动态链接。

六、 LLVM IR 编程

熟悉 LLVM IR 中间语言，对要实现的 SysY 编译器各语言特性，编写 LLVM 程序小例子，用 LLVM/Clang 编译城目标程序、执行验证。

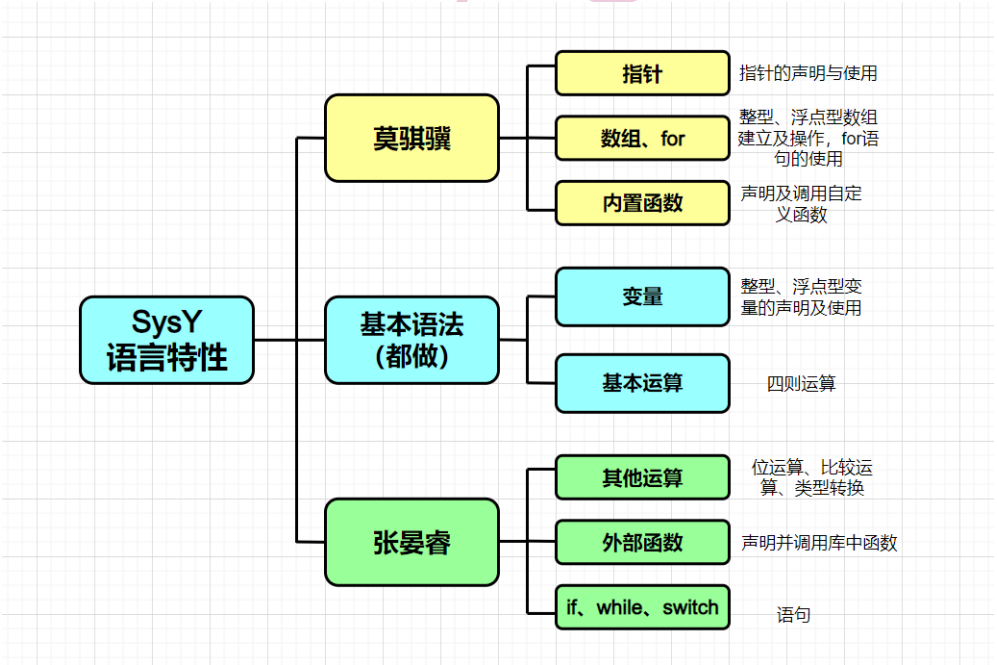

图 3: 小组分工图

C 代码

```c
#include <stdio.h>

const int N = 20;

```

```c
void func_a() { // 二元运算符
    int a = 10, b = 2;
    int sdiv_res = a / b;
    double c = 10.2, d = 9.8;
    double add_res = c + d;
    printf("%d\n", sdiv_res);
    printf("%f\n", add_res);
}

void func_b(int* arr) { //fibonacci数组
    int tmp1 = 0, tmp2 = 1;
    int* ptr = arr;
    for (int i = 0; ; i ++) {
        tmp2 = tmp2 + tmp1;
        tmp1 = tmp2 - tmp1;
        *ptr = tmp2;
        ptr = ptr + 1;
        if(ptr - arr >= N) break;
    }
}

int main() {
    func_a();
    int arr[N];
    func_b(arr);
    for(int i = 0; i < N; i ++) printf("%d\n", arr[i]);
    return 0;
}
```

上面的代码主要覆盖的 sysY 特性入下:

- int、float 两种变量类型的四则运算
- 全局常量的声明
- 指针的使用
- 数组的声明及操作
- for,if,break 语句
- 内部函数的声明和调用

llvm ir

```
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
    :16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

;声明全局变量N
@N = dso_local constant i32 20, align 4
```

```
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1 ;printf int
@.str.1 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1 ;printf double

;func_a
define dso_local void @func_a() #0 {
    %1 = alloca i32, align 4 ;a
    store i32 10, i32* %1, align 4 ;a = 10
    %2 = alloca i32, align 4 ;b
    store i32 2, i32* %2,align 4 ;b = 2
    %3 = alloca double, align 8 ;c
    store double 1.02e1, double* %3, align 8 ;c = 10.2
    %4 = alloca double, align 8 ;d
    store double 0.98e1, double* %4, align 8 ;d = 9.8
    %5 = alloca i32, align 4 ; sdiv_res
    %6 = load i32, i32* %1, align 4
    %7 = load i32, i32* %2, align 4
    %8 = sdiv i32 %6, %7 ;sdiv_res = a / b
    store i32 %8, i32* %5, align 4
    %9 = alloca double, align 8 ; sdiv_res
    %10 = load double, double* %3, align 8
    %11 = load double, double* %4, align 8
    %12 = fadd double  %10, %11
    store double %12, double* %9, align 8
    %13 = load i32, i32* %5, align 4
    %14 = call i32(i8*, ...)@printf(i8* getelementptr inbounds ([4 x i8],[4 x i8]* @.str, i64 0, i64 0), i32 %13)
    %15 = load double, double* %9, align 8
    %16 = call i32(i8*, ...)@printf(i8* getelementptr inbounds ([4 x i8],[4 x i8]* @.str.1, i64 0, i64 0), double %15)
    ret void
}

declare dso_local i32 @printf(i8*, ...) #1

;func_b
define dso_local void @func_b(i32* %0) #0 {
    %2 = alloca i32, align 4 ;tmp1
    %3 = alloca i32, align 4 ;tmp2
    %4 = alloca i32*, align 8 ;copy arr
    %5 = alloca i32*, align 8 ;ptr
    %6 = alloca i32, align 4 ; i
    store i32 0, i32* %2, align 4
    store i32 1, i32* %3, align 4
    store i32* %0, i32** %4, align 8
    %7 = load i32*, i32** %4, align 8
    store i32* %7,i32** %5, align 8
```

```
    store i32 0, i32* %6, align 4
    br label %8

8:
    %9 = load i32, i32* %2, align 4 ;9 = tmp1
    %10 = load i32, i32* %3, align 4 ;10 = tmp2
    %11 = add nsw i32 %9, %10 ; 11 = 9 + 10
    store i32 %11, i32* %3, align 4 ;store tmp2
    %12 = load i32, i32* %3, align 4 ; 12 = tmp2
    %13 = sub nsw i32 %12, %9 ; 13 = tmp2 - tmp1
    store i32 %13, i32* %2, align 4 ; store tmp1
    %14 = load i32, i32* %3, align 4 ; 从tmp2中取数
    %15 = load i32*, i32** %5, align 8 ;从ptr中取地址
    store i32 %14, i32* %15, align 4 ;*ptr = tmp2
    %16 = load i32*, i32** %5, align 8 ;16 = ptr
    %17 = load i32, i32* %16, align 4 ;17 = *ptr
    %18 = load i32, i32* %16, align 4 ;17 = *ptr,重复一个
    %19 = getelementptr inbounds i32, i32* %16, i64 1;ptr ++
    store i32* %19, i32** %5, align 8 ; ptr = 19
    %20 = load i32*, i32** %5, align 8 ;ptr
    %21 = load i32*, i32** %4, align 8 ;arr
    %22 = ptrtoint i32* %20 to i32
    %23 = ptrtoint i32* %21 to i32
    %24 = sub nsw i32 %22, %23
    %25 = sdiv exact i32 %24, 4
    %26 = icmp sge i32 %25, 20
    br i1 %26, label %30, label %27

27:
    %28 = load i32, i32* %6, align 4
    %29 = add nsw i32 %28, 1 ; i ++
    store i32 %29, i32* %6, align 4
    br label %8

30:
    ret void
}

;main
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca [20 x i32], align 16
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  call void @func_a()
  %4 = getelementptr inbounds [20 x i32], [20 x i32]* %2, i64 0, i64 0
  call void @func_b(i32* %4)
  store i32 0, i32* %3, align 4
```

```
98    br label %5
99
100   5:
101     %6 = load i32, i32* %3, align 4
102     %7 = icmp slt i32 %6, 20
103     br i1 %7, label %8, label %17
104
105   8:
106     %9 = load i32, i32* %3, align 4
107     %10 = sext i32 %9 to i64
108     %11 = getelementptr inbounds [20 x i32], [20 x i32]* %2, i64 0, i64 %10
109     %12 = load i32, i32* %11, align 4
110     %13 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4
            x i8]* @.str, i64 0, i64 0), i32 %12)
111     br label %14
112
113   14:
114     %15 = load i32, i32* %3, align 4
115     %16 = add nsw i32 %15, 1
116     store i32 %16, i32* %3, align 4
117     br label %5
118
119   17:
120       ret i32 0
121   }
122
123   attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide
        -sqrt-fp-math"="false" "disable-tail-calls"="false" "frame-pointer"="all"
         "less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-
        math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-
        signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector
        -buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+
        mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
124   attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-
        tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "
        no-infs-fp-math"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-
        math"="false" "no-trapping-math"="false" "stack-protector-buffer-size"="8
        " "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+
        x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
125
126   !llvm.module.flags = !{!0}
127   !llvm.ident = !{!1}
128
129   !0 = !{i32 1, !"wchar_size", i32 4}
130   !1 = !{!"clang version 10.0.0-4ubuntu1 "}
131
132   ;llvm-as llvm.ll -o llvm.bc
133
```

```
;llc llvm.bc -filetype=obj -o llvm.o

;gcc llvm.o -o llvm
```

上面代码的执行结果为：两个整数相除得到结果 5，两个浮点数相加得到结果 20.000，构造一个 fibonacci 数组：1,2,3,5,8，13,21，34,55，89..... 与 C 代码执行结果一致。

我们需要编写的是与 C 代码对应的 llvm ir 代码，.ll 文件还包含一些其他内容，例如：target datalayout,target triple,attribute #0,!llvm.module.flags 等内容具有普遍性，由其他 C 代码生成的中间代码复制而来。

全局声明

```
;声明全局变量N
@N = dso_local constant i32 20, align 4
```

全局变量用 @ 标识，i32 可认为是数据类型，即 32 位整数，align 4 按四字节对齐。

内置函数

```
define dso_local void @func_a() #0 {
    %1 = alloca i32, align 4 ;a
    store i32 10, i32* %1, align 4 ;a = 10
    %2 = alloca i32, align 4 ;b
    store i32 2, i32* %2,align 4 ;b = 2
    %3 = alloca double, align 8 ;c
    store double 1.02e1, double* %3, align 8 ;c = 10.2
    %4 = alloca double, align 8 ;d
    store double 0.98e1, double* %4, align 8 ;d = 9.8
    %5 = alloca i32, align 4 ; sdiv_res
    %6 = load i32, i32* %1, align 4
    %7 = load i32, i32* %2, align 4
    %8 = sdiv i32 %6, %7 ;sdiv_res = a / b
    store i32 %8, i32* %5, align 4
    %9 = alloca double, align 8 ; sdiv_res
    %10 = load double, double* %3, align 8
    %11 = load double, double* %4, align 8
    %12 = fadd double  %10, %11
    store double %12, double* %9, align 8
    %13 = load i32, i32* %5, align 4
    %14 = call i32(i8*, ...)@printf(i8* getelementptr inbounds ([4 x i8],[4 x
        i8]* @.str, i64 0, i64 0), i32 %13)
    %15 = load double, double* %9, align 8
    %16 = call i32(i8*, ...)@printf(i8* getelementptr inbounds ([4 x i8],[4 x
        i8]* @.str.1, i64 0, i64 0), double %15)
    ret void
}

declare dso_local i32 @printf(i8*, ...) #1
```

函数的定义用 define 开头，void 标明返回类型。对于局部变量的使用，要先对寄存器分配内存，例如对于局部变量 a，在函数体内用%1 寄存器存储，分配 1 个 32 位整数的存储空间，并按四

字节对齐；对于局部变量 c，分配一个 double 类型的存储空间，按八字节对齐。store 指令需指明数据类型和目的寄存器和对齐位数（浮点数的 store 用科学计数法），load 指令也是类似。此函数是计算两个整数的除法和两个浮点数的加法，对于浮点数加法，llvm 支持 fadd 指令，对于整数加法/减法，支持 add/sub 指令（有符号无符号均支持），对于整数除法，支持 udiv（无符号）、sdiv（有符号）。

外部函数

```
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1 ; printf int
declare dso_local i32 @printf(i8*, ...) #1 ;这是声明
```

对于外部函数 printf 的声明和调用，只会比葫芦画瓢地写，但是具体含义暂时还没搞清楚（尤其是调用语句）。

操作指针

```
%4 = alloca i32*, align 8 ;copy arr
%5 = alloca i32*, align 8 ;ptr
store i32* %0, i32** %4, align 8
%7 = load i32*, i32** %4, align 8
store i32* %7,i32** %5, align 8

%16 = load i32*, i32** %5, align 8 ;%16 = ptr
%17 = load i32, i32* %16, align 4 ;%17 = *ptr
%19 = getelementptr inbounds i32, i32* %16, i64 1;ptr ++
store i32* %19, i32** %5, align 8; ptr = %19
```

指针地址的数据类型为 int**，分配一个 int* 类型的空间，按八字节对齐，store 和 load 指令与前面整数、浮点数的存取指令写法类似。从指针中取数时需要进行两次取数指令，第一次是从指针寄存器中取出指针指向的地址，第二次是从该地址中取出数据，让指针指向 +1 的指令为 getelementptr 指令，声明索引类型为 i32 以及偏移量为 1 来计算偏移后的指针。

for、if、break 操作数组

```
...... ;声明变量并赋值
    br label %8

8:
    ..... ;对数组元素的操作，即执行for循环体
    ;if判断
    %20 = load i32*, i32** %5, align 8 ;ptr
    %21 = load i32*, i32** %4, align 8 ;arr
    %22 = ptrtoint i32* %20 to i32 ;ptr 扩展为 int32
    %23 = ptrtoint i32* %21 to i32
    %24 = sub nsw i32 %22, %23
    %25 = sdiv exact i32 %24, 4
    %26 = icmp sge i32 %25, 20
    br i1 %26, label %30, label %27

27:
```

```
    %28 = load i32, i32* %6, align 4
    %29 = add nsw i32 %28, 1 ; i ++
    store i32 %29, i32* %6, align 4
    br label %8

30:
    ret void
```

for 循环的边界控制在 llvm ir 中表现为 br 跳转指令，通过计算条件是否满足跳转到两个分支点，分别用于结束 for 循环和进行 for 语句末尾循环体。对于上面的代码，for 循环条件为计算当前指针所指向的地址和数组首地址的偏移量是否超出 20，将两寄存器中的地址值扩充为 32 位整数再相减之后除以 4（int32 为 4 字节），通过 icmp 对 26 号寄存器置位（i1），br 指令根据此寄存器的值选择跳转地址。

main 函数调用内部函数、分配数组内存

```
;main
define dso_local i32 @main() #0 {
  ......
  call void @func_a()
  %2 = alloca [20 x i32], align 16
  %4 = getelementptr inbounds [20 x i32], [20 x i32]* %2, i64 0, i64 0
  call void @func_b(i32* %4)
  ......
  ret i32 0
}
```

在主函数中通过 call 指令调用目标函数，用返回类型和 @ 函数名（形参）标识。

为数组 arr 分配 20 个 int32 的空间，按 16 字节对齐（发现当数组规模小于 4 时按 4 字节对齐，大于 4 时按 16 字节对齐，这样做可能是方便后续进行向量化打包处理）。%4 寄存器存放数组的首地址，使用 getelementptr 指令确定，这条语句中共有两个索引，第一个索引使%4 相对于 arr 首地址偏移了 0 x 20 x 4 字节，也就是没偏移，第二个索引使%4 在前一个索引偏移的基础上再偏移 0 x 4 个字节，因此计算之后%4 存储数组首地址。

这一段的理解参考博客LLVM IR(五)——IR 指令介绍. 另外，在编写代码时遇到了如下图所示的意外，在 funca 函数中给寄存器命名时 1 号寄存器不能使用，且寄存器名称必须升序命名，暂时没有找到原因，花了很久时间重新给所有寄存器重新命名。

```
mqq@mqq-virtual-machine:~/Desktop/complier/lab1/llvm_code$ llvm-as llvm.ll -o llvm.bc
llvm-as: llvm.ll:52:5: error: instruction expected to be numbered '%2'
    %1 = alloca i32, align 4 ;tmp1
    ^
```

图 4: 1 号寄存器不能使用

代码仓库.