



南開大學
Nankai University

南 開 大 學

計 算 機 學 院

编译原理实验报告

预备工作：了解编译器 & LLVM IR 编程 & 汇编编程

高连儒 杨涵

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2024 年 9 月 18 日

摘要

在本次实验之中，我们将主要对编译系统的各方面进行基础的了解，为之后的编译原理实验打下基础，在本实验中，首先我们通过一个简单的程序了解了编译器在各编译阶段所做的工作，同时，我们同时也了解了 LLVM IR 和汇编两种语言的简单编程方式，熟悉了两种语言，并编写简单小程序。

关键字：编译器、汇编、LLVM IR

目录

| | |
|-------------------------------|-----------|
| 一、 分工说明与实验平台 | 1 |
| (一) 分工说明 | 1 |
| (二) 实验平台 | 1 |
| 1. 杨涵 | 1 |
| 2. 高连儒 | 1 |
| 二、 了解编译器 | 2 |
| (一) 总览 | 2 |
| (二) 预处理器 | 3 |
| (三) 编译器 | 4 |
| 1. 词法分析 | 4 |
| 2. 语法分析 | 4 |
| 3. 语义分析 | 5 |
| 4. 中间代码生成 | 5 |
| 5. 代码优化 | 5 |
| 6. 代码生成 | 6 |
| (四) 汇编器 | 6 |
| (五) 链接器加载器 | 7 |
| (六) 额外工作: 程序微调 | 7 |
| 1. 局部变量与全局变量 | 7 |
| 2. while 与 for 循环语句 | 8 |
| 3. 代码优化: 减少变量数量 | 9 |
| 三、 llvm ir 程序设计——杨涵 | 10 |
| (一) 语言特性总结 | 10 |
| (二) llvm 小程序 | 11 |
| 四、 汇编程序设计——高连儒 | 14 |
| 五、 总结 | 16 |
| 六、 仓库链接 | 16 |

一、 分工说明与实验平台

(一) 分工说明

- 杨涵：了解编译器 +llvm ir 程序设计
- 高连儒：了解编译器 + 汇编程序设计
- 其中，“了解编译器”部分各自独立撰写。

(二) 实验平台

1. 杨涵

- 编译环境：
gcc (Ubuntu 13.2.0-4ubuntu3) 13.2.0
Ubuntu clang version 16.0.6 (15)
QEMU emulator version 6.2.0 (Debian 1:6.2+dfsg-2ubuntu6.22)
riscv64-unknown-linux-gnu-gcc () 12.2.0
- 硬件环境：
12th Gen Intel(R)Core(TM)i7-12700H (20 CPUs), 2.3GHZ
windows 11、ubuntu 23.10
NVIDIA GeForce RTX 3070 Laptop GPU

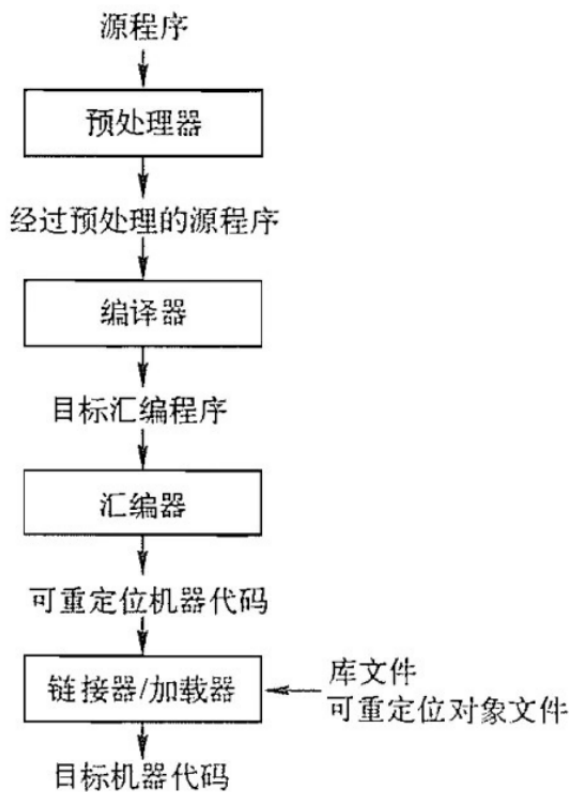
2. 高连儒

- 编译环境：
gcc (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0
QEMU emulator version 6.2.0 (Debian 1:6.2+dfsg-2ubuntu6.22)
riscv64-unknown-linux-gnu-gcc () 12.2.0
- 硬件环境：intel i9 12900H laptop
windows 11、ubuntu 22.04
NVIDA 3070Ti Laptop

二、 了解编译器

(一) 总览

我们编写的高级语言程序要成为系统可识别、运行的可执行文件，需要经过一系列过程。以一个 C 程序为例，整体的流程如图所示：



简单来说，不同阶段的作用如下：

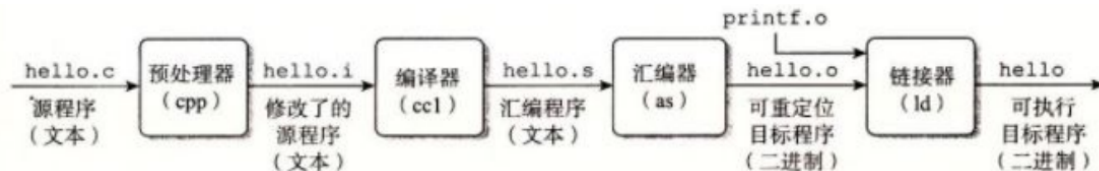
预处理器 处理源代码中以 # 开始的预编译指令，例如展开所有宏定义、插入 #include 指向的文件等，以获得经过预处理的源程序。

编译器 将预处理器处理过的源程序文件翻译成为标准的**汇编语言**以供计算机阅读。

汇编器 将汇编语言指令翻译成**机器语言指令**，并将汇编语言程序打包成可重定位目标程序。

链接器 将可重定位的机器代码和相应的一些目标文件以及库文件链接在一起，形成真正能在机器上运行的目标机器代码。

一个 C 程序 hello.c，经历上述 4 个编译阶段最终生成可执行程序：



下面将以“斐波那契数列”的简单 c 语言程序为例，分析各个阶段的具体工作。完整的参考代码如下。

实验程序：斐波那契数列

```

1 #include<stdio.h>
2 int main()
3 {
4     int a,b,i,t,n;
5     a=0;
6     b=1;
7     i=1;
8     scanf("%d",&n);
9     printf("%d\n",a);
10    printf("%d\n",b);
11    while(i<n)
12    {
13        t=b;
14        b=a+b;
15        printf("%d\n",b);
16        a=t;
17        i=i+1;
18    }
19    return 0;
20 }

```

(二) 预处理器

预处理阶段会处理预编译指令，包括绝大多数的 # 开头的指令，如 #include、#define、#if 等等，对 #include 指令会替换对应的头文件，对 #define 的宏命令会直接替换相应内容，同时会删除注释，添加行号和文件名标识。

对于 gcc，通过添加参数 -E 令 gcc 只进行预处理过程，参数 -o 改变 gcc 输出文件名，因此通过命令 gcc main.c -E -o main.i 得到预处理后文件。观察到生成的文件比原代码长了许多（具体来说，原代码 20 行，.i 文件 800 余行）。以下是对 .i 文件中的部分信息引用与说明。

```

1 # 21 "/usr/include/features-time64.h" 2 3 4
2 # 1 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 1 3 4

```

- 头文件包含：将 #include 指令指定的头文件内容插入到源文件中。
- 添加行号和文件名信息：以便追踪源代码位置。

```

1 typedef __uint64_t __uint_least64_t;这段代码干了什么？
2 typedef struct __G_fpos_t
3 {
4     __off_t __pos;
5     __mbstate_t __state;
6 } __fpos_t;

```

- 类型别名定义：提高可读性，或者为了与某些标准或库兼容：确保代码在不同平台上的一致性。

- `__fpos_t` 代表文件位置的结构体类型，通常用于标准库函数中。通过定义结构体、创建类型别名，可以提高代码的可读性和维护性，并且使得不同代码部分能够一致地引用这个结构体。

此外，`main` 函数中代码主体并没有变化。

(三) 编译器

本节将详细阐述编译器在各个环节进行的工作，即分析编译器是如何将预处理器处理后的代码变为汇编代码的。

1. 词法分析

词法分析将源程序转换为单词序列。通过 `clang -E -Xclang -dump-tokens main.c` 查看词法分析得到的 token 序列。

```

1 identifier 'b'      [StartOfLine] [LeadingSpace]    Loc=<main.c:14:9>
2 equal '='          Loc=<main.c:14:10>
3 identifier 'a'      Loc=<main.c:14:11>
4 plus '+'           Loc=<main.c:14:12>

```

上面给出的是对 `main` 函数中部分代码进行的词法分析。可以看到，除了对运算符、变量进行识别之外，还有类似“位于行最前端”“前有空格”等辅助信息。此外，词法分析的作用对象是预处理器处理之后的结果，因此同样会有长度的额外扩充。

2. 语法分析

将词法分析生成的词法单元来构建抽象语法树 (AbstractSyntaxTree, 即 AST)。LLVM 可以通过 `clang -E -Xclang -ast-dump main.c` 命令获得相应的 AST。

语法分析的作用对象同样是预处理器处理之后的代码，我们关注的主要是 `main` 函数对应的语法分析树，需要跳过前面大部分内容。由于并非可视化视图，需要看出树形结构相对困难，这里截取比较易于理解的部分：

```

1 -FunctionDecl 0x63f9794ff250 <main.c:2:1, line:20:1> line:2:5 main 'int
  ()'-CompoundStmt 0x63f9794fff70 <line:3:1, line:20:1>
2   |-DeclStmt 0x63f9794ff5a8 <line:4:5, col:18>
3     |-VarDecl 0x63f9794ff310 <col:5, col:9> col:9 used a 'int'
4     |-VarDecl 0x63f9794ff390 <col:5, col:11> col:11 used b 'int'
5     |-VarDecl 0x63f9794ff410 <col:5, col:13> col:13 used i 'int'
6     |-VarDecl 0x63f9794ff490 <col:5, col:15> col:15 used t 'int'
7     |-VarDecl 0x63f9794ff510 <col:5, col:17> col:17 used n 'int'|-BinaryOperator 0x63f9794ff600
      <line:5:5, col:7> 'int' '='|-DeclRefExpr 0x63f9794ff5c0 <col:5> 'int' lvalue Var
      0x63f9794ff310 'a' 'int'|-IntegerLiteral 0x63f9794ff5e0 <col:7> 'int' 0
8     |-BinaryOperator 0x63f9794ff660 <line:6:5, col:7> 'int' '='
9     |-DeclRefExpr 0x63f9794ff620 <col:5> 'int' lvalue Var 0x63f9794ff390 'b'
      'int'
10    |-IntegerLiteral 0x63f9794ff640 <col:7> 'int' 1|-BinaryOperator 0x63f9794ff6c0 <line:7:5, col:7>
      'int' '='|-DeclRefExpr 0x63f9794ff680 <col:5> 'int' lvalue Var 0x63f9794ff410 'i' 'int'|-
      IntegerLiteral 0x63f9794ff6a0 <col:7> 'int' 1

```

这部分描述的是 `main` 函数的声明（显然，在最底部对应有 `return` 的节点）、几个变量的声明与初始值信息。对于 `a=0` 这条语句，可以看到对应子树的根节点是运算符 `0`，两个节点分别是 `'a'`

与'0'，与我们学习的抽象语法树格式是一致的；比起我们上课画的简单抽象语法树，节点上还有一些额外信息，如变量类型、变量地址、语句对应位置等。

3. 语义分析

该步使用语法树和符号表中信息来检查源程序是否与语言定义语义一致，进行类型检查等。编译器主要负责验证代码的逻辑正确性，确保所有的变量和函数调用都有正确的类型和使用方式，检查表达式和语句的语义是否符合编程语言的规范，以及进行作用域解析和符号表的管理等工作。

4. 中间代码生成

使用命令 `clang-S-emit-llvm main.c` 获得 `llvmlir` 语言的中间代码。这一步生成的文件非常简洁，已经有些类似汇编代码。主要的特点是用 `%num` 的形式表示临时变量（临时寄存器），用 `alloca` 命令分配内存（本质是将寄存器的值写入内存地址，类似 `c` 语言中的指针变量），`load` 指令加载值、`store` 指令存入对应内存地址、`add` 指令进行运算操作等。与汇编指令比较类似的一点还在于使用 `label` 指令标注代码块，但跳转指令使用的是 `br`。

需要注意的是，`llvmlir` 语言的一大特点是几乎所有变量（甚至包括指令调用本身）都需要指明参数类型，如 `i32`（短整型）、`i1`（`bool` 类型）等。

```
1 12:                                     ; preds = %16, %0
2   %13 = load i32, ptr %4, align 4
3   %14 = load i32, ptr %6, align 4
4   %15 = icmp slt i32 %13, %14
5   br i1 %15, label %16, label %26
```

这里贴出的是比较典型的一个代码块，意思是将 13 号寄存器的值与 14 号寄存器的值比较，如果 `%13` 的值小于 `%14` 的值则跳转到 `%16` 代码块，否则到 `%26` 代码块。这个过程需要借助 `%15` 中间变量（存储比较结果，`i1` 布尔类型）来完成。这段代码对应的是 `while` 语句块的条件判断部分。

5. 代码优化

进行与机器无关的代码优化步骤、从而改进中间代码，生成更好的目标代码。由于输出文件太长（输出需要用到输入输出重定向），这里不作示例，给出 `llvm` 编译器代码优化几个具体步骤执行的操作说明：

- **Analysis Passes:** 分析 Pass 负责在编译过程中对代码进行各种分析。这些 Pass 不修改程序的结构或语义，而是收集程序的各种信息以供其他 Pass 使用。分析 Pass 提供的数据可以用于优化或验证代码。常见的分析 Pass 包括：
 - **Alias Analysis:** 用于分析指针别名关系，确定哪些指针可能指向同一内存位置。
 - **Loop Analysis:** 对循环结构进行分析，提取循环嵌套层次、循环依赖信息等。
 - **Control Flow Graph (CFG) Analysis:** 分析程序的控制流图，检测分支、循环、异常处理等结构。
- **Transform Passes:** 转换 Pass 会修改代码以优化或改变程序的结构。它们在编译过程中对中间表示（IR）进行转换，旨在提高程序性能或生成目标代码。这类 Pass 主要用于优化和代码生成。常见的转换 Pass 包括：

- Loop Unrolling: 对循环展开以减少循环控制开销。
- Inlining: 将被调用的函数内联到调用点, 减少函数调用开销。
- Dead Code Elimination (DCE): 消除程序中从未使用到的代码。
- **Utility Passes:** 实用 Pass 是一些辅助性 Pass, 用于支持调试、分析报告生成或其他非优化类的功能。这类 Pass 通常不影响生成的代码, 而是为开发人员或编译器提供额外的信息或工具。常见的实用 Pass 包括:
 - PrintModulePass: 用于将当前模块的中间表示输出到控制台, 帮助开发者调试。
 - VerifierPass: 检查代码的合法性和一致性, 确保程序不包含非法或未定义的 IR 结构。
 - StatisticsPass: 收集编译过程中的各种统计信息, 如优化次数、函数数量等。

6. 代码生成

这一步主要是将优化后的中间代码转变为目标汇编代码(可以由输入决定, 包括 llvm, arm, x86 等)。由于中间代码已经与最终的汇编代码非常相似, 所以这一步的转化并不需要太多工作。这里生成的代码使用的寄存器名称都具有实际含义, 不再像前文中的那样以数字指代临时寄存器。

这一步生成的汇编程序其实是大家相对熟悉的, 但需要注意的是, 编译器生成的汇编程序不仅包括有语义直接与机器代码对应的指令(如 x86 架构下的 jmp, add, sub 等指令), 也包括一些伪指令(即没有直接的机器代码与之对应, 但汇编器可以理解并根据指令作出目标行为), 如 align 对齐地址指令等。

```
1 .L2:  
2 movl    -28(%rbp), %eax  
3 cmpl    %eax, -16(%rbp)  
4 jl     .L3
```

这里给出的示例同样为循环判断退出条件对应的代码(需要用到的数据已经存入相关寄存器), 可以直观地体现与中间代码的差别。

(四) 汇编器

简单来说, 汇编器负责将上一阶段我们得到的汇编代码转化为机器能直接加载的机器代码(即二进制代码)。到了这一步骤, 人类已经很难去理解代码具体的含义(除非照着指令说明一一对应, 但难度很高且意义不大), 但这是计算机执行代码的必要步骤。

这里给出一些汇编器执行的具体操作, 就不再给出示例(输出出来为 16 进制机器码, 意义并不显然)。

- **符号解析:** 确定程序中的符号(如变量、标签)的地址或偏移量, 并将它们与具体内存位置关联。
- **伪指令处理:** 处理伪指令, 如 .data、.text 等, 帮助定义数据段、对齐等。
- **指令编码:** 将汇编语言指令转换为对应的机器指令, 生成处理器能执行的二进制代码。
- **段管理:** 将代码和数据分配到不同的段, 如代码段、数据段, 并确保段的组织符合目标文件格式。

- **重定位信息生成**：生成重定位信息，允许链接器在链接阶段调整符号地址，确保程序正确执行。
- **生成目标文件**：生成目标文件，包含机器码、符号表和重定位信息，供链接器使用。

(五) 链接器加载器

链接器

- **整合目标文件**：将多个目标文件（.o 或 .obj 文件）和库文件（.a 或 .lib 文件）合并成一个可执行文件或共享库。
- **符号解析**：解决不同文件中的函数和变量引用，确保它们正确地指向实际的定义。
- **重定位**：调整目标文件中的地址，以反映最终可执行文件中的实际内存布局。

加载器

- **加载文件**：将可执行文件加载到内存中，以便程序可以运行。
- **地址映射**：将程序的虚拟地址映射到实际的物理内存地址。
- **初始化执行**：设置程序的运行环境，并开始执行程序。

编译工具经过一系列处理后生成可执行文件，执行可执行文件时由**加载器**将二进制文件载入内存、开始执行。这部分工作更偏向体系结构课程的研究内容，与编译原理课程关系不大，因此不作详细展开。

(六) 额外工作: 程序微调

此节的研究方法主要为：对高级语言源程序进行修改，观察编译器生成的中间代码或汇编代码的变化。

1. 局部变量与全局变量

将 a,b,i,t,n 五个变量从局部变量改为全局变量（即在 main 函数外声明）后，观察中间代码程序变化。

局部变量下的声明语句

```
1 %l = alloca i32, align 4 ;该变量有些奇怪，见下文
2 %2 = alloca i32, align 4
3 %3 = alloca i32, align 4
4 %4 = alloca i32, align 4
5 %5 = alloca i32, align 4
6 %6 = alloca i32, align 4
7 store i32 0, ptr %1, align 4 ;%1除了在这里赋值之外没有见到再出现，作用存疑
8 store i32 0, ptr %2, align 4 ;以下几句为a,b,i赋值语句
9 store i32 1, ptr %3, align 4
10 store i32 1, ptr %4, align 4
```

全局变量下的声明语句

```

1  @a = dso_local global i32 0, align 4      ;变量声明在main函数体之外, 且直接生成
    了内存地址
2  @b = dso_local global i32 0, align 4
3  @i = dso_local global i32 0, align 4
4  @n = dso_local global i32 0, align 4
5  @t = dso_local global i32 0, align 4
6  .....
7  .... 下面为main函数体中的赋值语句 .....
8  %1 = alloca i32, align 4                  ;该变量与上文的%1相同
9  store i32 0, ptr %1, align 4              ;%1在这里赋值后不再出现
10 store i32 0, ptr @a, align 4              ;以下几句为a,b,i赋值语句
11 store i32 1, ptr @b, align 4
12 store i32 1, ptr @i, align 4

```

如注释所述, 可以很明显地看出全局变量与局部变量声明的区别, 即全局变量声明时就已分配了对应的内存地址, 可以直接调用它对应的指针 (且以 @ 开头); 局部变量如果想要使用内存相关功能 (包括 store 等函数), 必须先使用 alloca 分配内存地址到寄存器中。此外, %1 疑似是多余的。

2. while 与 for 循环语句

可以将 while 循环改写为 for 循环语句, 以此观察两种不同的循环结构对中间程序的影响。修改后的循环体如下:

实验程序: for 循环

```

1  for(i=1;i<n;i++)
2  {
3      t=b;
4      b=a+b;
5      printf("%d\n",b);
6      a=t;
7  }

```

这里我将贴出 for 循环下完整的中间程序代码:

for 循环下的 llvm 代码

```

1  12:                                     ; preds = %24, %0
2      %13 = load i32, ptr %4, align 4
3      %14 = load i32, ptr %6, align 4
4      %15 = icmp slt i32 %13, %14
5      br i1 %15, label %16, label %27
6
7  16:                                     ; preds = %12
8      %17 = load i32, ptr %3, align 4
9      store i32 %17, ptr %5, align 4
10     %18 = load i32, ptr %2, align 4
11     %19 = load i32, ptr %3, align 4
12     %20 = add nsw i32 %18, %19

```

```

13 store i32 %20, ptr %3, align 4
14 %21 = load i32, ptr %3, align 4
15 %22 = call i32 @printf(ptr noundef @.str.1, i32 noundef %21)
16 %23 = load i32, ptr %5, align 4
17 store i32 %23, ptr %2, align 4
18 br label %24 ; 注意这里!!
19
20 24: ; preds = %16
21 %25 = load i32, ptr %4, align 4
22 %26 = add nsw i32 %25, 1
23 store i32 %26, ptr %4, align 4
24 br label %12, !llvm.loop !6
25
26 27: ; preds = %12
27 ret i32 0

```

注意到，上面的程序有非常明显的冗余！在代码块 16 的最后一行额外创建了一个代码块（24）用于循环变量 i 的值迭代、重新进入循环判断条件（代码块 12），但实际上代码块 24 完全是没有必要创建的，直接添加在 16 块末端即可。推测这是因为 clang 编译器识别 for 语句的逻辑需要为变量迭代、条件判断分别创建块，但额外的跳转指令会带来不必要的开销。事实上，while 循环与 for 循环除了没有创建块 24 之外是完全一致的（再给出 while 语句对应会显得报告过于冗余，但这个结论是仔细比对后作出的判断，真实可靠）。当循环语句足够多时，仅从 llvm 中间程序的角度来看 while 语句可能带来比 for 语句更高的性能。

此外，使用 `llc main.ll -o main.S` 语句，即借助中间 llvm 程序生成汇编程序后，同样发现 for 循环下比起 while 循环多了一个代码块，结论是一致的。

3. 代码优化：减少变量数量

原代码中：`t=b, b=a+b, a=t` 的写法可以缩减为两个变量，运算逻辑变为：`b=a+b, a=b-a`，同样选择用 b 来表示当前斐波那契数列的最后一项。修改后的程序略。

做这步的目的是想研究编译器是否会对这种具有较强明确含义的运算式子进行逻辑上的优化和替换，但就 clang 而言似乎并没有额外工作。这个结论对于中间代码和生成的汇编代码是一致的。

这里给出比较便于理解的 llvm 中间代码（对应着循环体中变量迭代的部分）：

```

1 %21 = load i32, ptr %3, align 4
2 %22 = load i32, ptr %2, align 4
3 %23 = sub nsw i32 %21, %22
4 store i32 %23, ptr %2, align 4
5 %24 = load i32, ptr %4, align 4
6 %25 = add nsw i32 %24, 1
7 store i32 %25, ptr %4, align 4
8 br label %11, !llvm.loop !6

```

注意到，虽然我们少定义了一个变量 t ，但在执行减法操作时又额外定义了一个临时变量用于存储减法操作的结果，实际性能可能并不会得到提升（最终和原程序相比用到的临时寄存器数目恰好也是相等的），同时会使代码可读性变差，因此从我个人角度而言更推荐老老实实定义临时变量在中间作赋值操作。

三、 llvm ir 程序设计——杨涵

(一) 语言特性总结

此节并非正式的定义与总结，算是本人自学 llvm ir 语言之后发现的一些值得注意的点。

- 类型声明：llvm ir 语言中需要特别关注变量类型的声明，包括但不限于变量声明、指令调用时在寄存器前对寄存器类型进行解释等。
- 变量类型：包括 float, i32 (即 int 类型变量), i1 (即 bool 类型变量) 等。课件 pdf 中给出了对于存放内存地址 (该地址保存 i32 变量) 寄存器的变量声明为 i32*, 但通过中间代码输出发现通常直接使用 ptr 说明这是指针类型变量即可。
- 前缀区别：分为 % 和两种。%+ 数字的组合意味着这是一个局部变量、临时寄存器，也可能是作为代码块的标签存在 (标注代码块时不需要加 % 前缀，使用 br 指令跳转时需要添加)；+ 变量名称的组合意味着这是一个全局变量，也可能与函数名连用、或是外部引用。
- 指令总结：llvm ir 语言不同于汇编语言，汇编中多是三目运算，即指令中包括两个操作数、一个目的寄存器，llvm ir 语言的指令多是类似 c 程序的二目运算，需要用到赋值操作来实现值的传递。

下面给出一些常见的指令类型。

1. 算术指令：用于执行寄存器间 (或与常量) 的运算操作，包括 add、sub、mul 等。格式如下文所示：

```
1 %1 = add nsw i32 %2, %3
```

这里的 nsw 用于指示不会发生有符号整型溢出，删掉不影响代码语义，但可能影响后续优化决策；由于运算指令只能对两个相同类型变量操作，因此只需要声明一次操作数类型。

2. 比较指令与跳转指令：比较指令包括 slt、sgt 等，返回类型为 i1 (bool) 类型，通常与跳转指令连用。跳转指令可以单独使用，无条件跳转：br %1 即可。

```
1 %9 = icmp sle i32 %7, %8
2 ; 小于等于指令，前面以icmp声明这是对int类型进行比较，结果存储在%9中
3 br i1 %9, label %10, label %16
4 ;%9为真则跳转至代码块10，否则跳转至代码块16
```

3. 内存相关指令：分配内存地址使用 alloca 指令。本质是给寄存器写入一个内存地址，有些类似 c 中的指针声明，直接使用寄存器的值是对地址本身操作，前面添加 * 才是对该内存地址保存的值进行操作。临时变量只有分配了内存地址才能使用 * 进行地址指向，包括 store (存值)、load (读值) 指令等。

```
1 %1 = alloca i32, align 4
2 store i32 2, i32* %1, align 4 ;注意%1前的变量声明是i32*,意思是将常量2写入%1
   对应内存地址中
3 %2 = load i32* %1, allgn 4
4 ; align可以理解成额外的可选项，意思是内存地址以4字节对齐
```

4. 函数调用指令：使用形如`%1=call i32 @f(i32 %2)` 这样的格式。与 c 的函数调用几乎完全相同，区别在于：1. 使用 `call` 指令。2. 需要声明函数返回类型。3. 函数名前记得添加标识。4. 参数列表同样需要一一指明类型。
5. (函数中的) 返回指令：使用 `ret` 指令返回函数值，类似于 c 中的 `return` 操作。形如 `ret i32 0` 或 `ret void`, 比较容易理解，不再多做解释。

(二) llvm 小程序

有了上面总结的知识点, llvm 语言几乎算是入门了。下面是对上面的知识点进行应用, 编写 llvm 程序的过程。这里我们选择构筑一个“计算阶乘”的程序, 代码逻辑本身比较简单, 但为了熟悉 llvm 语言的编写, 我们选择使用数组实现, 并额外定义一个函数体用于解决问题。具体代码见下文, 同时写有详细的注释。

```

1 ; 注意: 每个函数体中的临时变量可以重新编号
2 ; 函数声明
3 declare i32 @printf(i8*, ...)
4 declare i32 @__isoc99_scanf(i8*, ...)
5
6 @fmt = private constant [3 x i8] c"%d\00" ; 定义格式化字符串
7 ; 格式化字符串包括字符串结束符, 故scanf和printf参数列表定义3 x i8
8
9 ; Solve函数定义
10 define i32 @Solve(i32 %0) {
11 1:
12     ; 定义变量
13     %2 = alloca [105 x i32], align 4 ; 分配数组a[105]
14     %3 = alloca i32, align 4 ; 分配i的空间
15     %4 = getelementptr [105 x i32], [105 x i32]* %2, i32 0, i32 1 ; 获取a[1]
        的指针
16
17     store i32 2, i32* %3, align 4 ; i = 2
18     store i32 1, i32* %4, align 4 ; a[1] = 1
19
20     ; 进入while循环
21     br label %5
22
23 5: ; while循环的条件检查
24     %6 = load i32, i32* %3, align 4 ; 读取i的值
25     %7 = icmp sle i32 %6, %0 ; 比较i <= n
26     br i1 %7, label %8, label %15 ; 如果条件成立跳转到
        while_body, 否则跳转到while_end
27
28 8: ; while循环体
29     ; 计算 a[i] = a[i-1] * i
30     %9 = sub i32 %6, 1 ; i - 1
31     %10 = getelementptr [105 x i32], [105 x i32]* %2, i32 0, i32 %9 ; 获取a[i
        -1]的指针
32     %11 = load i32, i32* %10, align 4 ; 读取a[i-1]的值

```

```

33
34     %12 = getelementptr [105 x i32], [105 x i32]* %2, i32 0, i32 %6 ; 获取a[i
        ]的指针
35     %13 = mul i32 %11, %6 ; a[i] = a[i-1] * i
36     store i32 %13, i32* %12, align 4 ; 存储a[i]
37
38     ; i = i + 1
39     %14 = add i32 %6, 1 ; i + 1
40     store i32 %14, i32* %3, align 4 ; 存储i的值
41
42     br label %5 ; 跳转回到while_cond检查条
        件
43
44 15: ; while循环结束
45     %16 = getelementptr [105 x i32], [105 x i32]* %2, i32 0, i32 %0 ; 获取a[n
        ]的指针
46     %17 = load i32, i32* %16, align 4 ; 读取a[n]的值
47     ret i32 %17 ; 返回a[n]
48 }
49
50 ; main函数定义
51 define i32 @main() {
52     0:
53     ; 定义变量
54     %1 = alloca i32, align 4 ; 分配n的空间
55     %2 = alloca i32, align 4 ; 分配ans的空间
56
57     ; 调用scanf("%d", &n) 并保存返回值
58     %3 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x
        i8]* @fmt, i64 0, i64 0), i32* %1) ; 执行scanf, 并将返回值存储在%3中
59
60     ; 调用Solve(n) 并保存返回值
61     %4 = load i32, i32* %1, align 4 ; 读取n的值
62     %5 = call i32 @Solve(i32 %4) ; 调用Solve(n)函数, 并将返
        回值存储在%5中
63     store i32 %5, i32* %2, align 4 ; 存储Solve的返回值
64
65     ; 调用printf("%d", ans) 并保存返回值
66     %6 = call i32 @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]*
        @fmt, i64 0, i64 0), i32 %5) ; 执行printf, 并将返回值存储在%6中
67
68     ret i32 0 ; main函数返回0
69 }

```

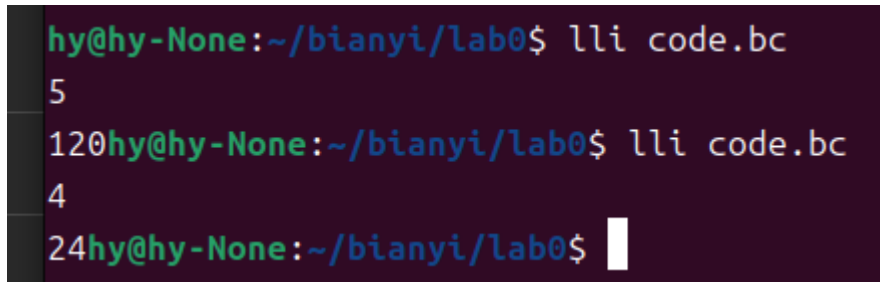
具体的设计思路与对应的高级语言语句已经在注释中标明。经过验证, 代码可以正常编译为正确的可执行文件。

使用流程 (llvm 程序保存在 code.ll 中): 先使用 `llvm-as code.ll code.bc` 转为二进制文件; 再使用 `lli code.bc` 运行生成的二进制文件即可。

设计代码过程中，额外发现的需要注意的点有：

- 每个函数体中的临时变量编号必须严格命名，例如不能在定义%1 的情况下定义%2。此外，寄存器编号是从%0 开始的。所谓“编号顺序”指的是从函数体起始到结束（语句顺序，并非逻辑顺序），新定义的变量（或 label）顺次编号！在 br 指令中新出现的 label 块不能算作是新定义！
- 函数体的寄存器编号要从函数列表算起，即函数列表中的参数也被视为函数体中的临时变量。
- 全局变量（为起始标识符）最好不使用数字、转而使用字符来命名，以增加代码可读性。
- 如 scanf 和 printf 这样要用到字符串的函数可以定义全局字符串，通过获取字符串地址（需要指定长度）来实现函数的调用。

下面是程序运行的截图。



```
hy@hy-None:~/bianyi/lab0$ lli code.bc
5
120hy@hy-None:~/bianyi/lab0$ lli code.bc
4
24hy@hy-None:~/bianyi/lab0$
```

四、 汇编程序设计——高连儒

我在这一部分，利用 SysY 写了两个简单的小程序，分别是求数组中的最大值和求斐波那契数列的值，具体代码如下：

求数组中的最大值

```

1 int main() {
2     int arr[5] = {3, 1, 4, 1, 5};
3     int result = arr[0];
4     for (int i = 1; i < 5; i++) {
5         if (arr[i] > result) {
6             result = arr[i];
7         }
8     }
9     return result;
10 }

```

求斐波那契

```

1 int main() {
2     int n = 6;
3     if (n == 0) return 0;
4     if (n == 1) return 1;
5     int a = 0, b = 1, fib;
6     for (int i = 2; i <= n; i++) {
7         fib = a + b;
8         a = b;
9         b = fib;
10    }
11    return fib;
12 }

```

然后我分别编写了其对应的汇编代码 (RISC-V 版), 具体如下,

求数组最大值

```

1     .section .text
2     .globl main
3
4 main:
5     addi sp, sp, -32      # 为栈帧分配32字节的空间
6     la a0, arr            # a0 = arr的地址
7     lw s0, 0(a0)          # s0 = arr[0], 初始化result
8     li s1, 1              # s1 = 1, 作为循环变量i
9
10 find_max_loop:
11     li t0, 5              # t0 = 数组长度5
12     bge s1, t0, end_main  # 如果i >= 5, 跳转到end_main
13     slli t1, s1, 2         # 计算arr[i]的地址偏移, t1 = i * 4
14     add t1, a0, t1         # t1 = &arr[i]

```



```

15     lw t2, 0(t1)           # t2 = arr[i]
16     blt s0, t2, update_max # 如果result < arr[i], 跳转到update_max
17
18 next_iteration:
19     addi s1, s1, 1         # i++
20     j find_max_loop        # 跳转到下一次循环
21
22 update_max:
23     mv s0, t2              # result = arr[i]
24     j next_iteration       # 继续循环
25
26 end_main:
27     mv a0, s0              # 返回result
28     addi sp, sp, 32        # 释放栈帧空间
29     jr ra                  # 返回
30
31     .section .data
32 arr:
33     .word 3, 1, 4, 1, 5    # 初始化数组

```

求斐波那契

```

1     .section .text
2     .globl main
3
4 main:
5     addi sp, sp, -32       # 为栈帧分配32字节的空间
6     li a0, 6               # a0 = 6, n = 6
7
8     li t0, 0               # t0 = 0, 表示斐波那契的初始值a
9     li t1, 1               # t1 = 1, 表示斐波那契的初始值b
10
11    li t2, 0                # t2 = 0, 临时变量存储斐波那契数列值fib
12    li t3, 2                # t3 = 2, 循环计数器i
13
14    beq a0, t0, return_zero # 如果n == 0, 返回0
15    beq a0, t1, return_one  # 如果n == 1, 返回1
16
17 fib_loop:
18    ble a0, t3, end_main    # 如果i > n, 跳转到end_main
19    add t2, t0, t1          # fib = a + b
20    mv t0, t1               # a = b
21    mv t1, t2               # b = fib
22    addi t3, t3, 1          # i++
23
24    j fib_loop              # 继续循环
25
26 return_zero:
27    li a0, 0                # 返回0

```

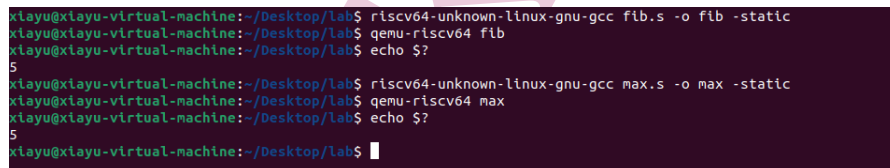
```
28     j end_main
29
30 return_one:
31     li a0, 1          # 返回1
32     j end_main
33
34 end_main:
35     mv a0, t2          # 返回fib的值
36     addi sp, sp, 32    # 释放栈帧空间
37     jr ra              # 返回
```

由于程序较为简单，所以转换思路也较为简洁，在第一个程序中，我们需要首先将数组保存在数据段。之后使用寄存器来存储数组元素、循环变量和当前最大值。在循环中，每次比较当前元素和已存储的最大值，如果新元素更大，则更新最大值。利用条件分支指令进行判断，当循环完成后，最终的最大值保存在寄存器中，并作为结果输出。

而在第二个斐波那契程序之中，我们需要通过两个寄存器初始化前两项值（ $a = 0$ 和 $b = 1$ ，这里是把 0 作为第一项来看），并使用一个循环逐步计算斐波那契数。每次循环时，将前两项的和存储在一个寄存器中，更新这两个寄存器的值，然后递增循环变量。在这之中条件分支指令用于判断循环终止条件。

另外，通过之前学过的集中汇编语言相比较，可以发现其较 x86 更为简洁，与 mips 等语言较为相似，也很容易混淆。

由于 SysY 中没有 printf 等函数，因此我们采用主函数返回值的方式返回最终结果，并通过 echo \$? 来查看最终结果，如图1所示：



```
xlayu@xlayu-virtual-machine:~/Desktop/lab$ riscv64-unknown-linux-gnu-gcc fib.s -o fib -static
xlayu@xlayu-virtual-machine:~/Desktop/lab$ qemu-riscv64 fib
xlayu@xlayu-virtual-machine:~/Desktop/lab$ echo $?
5
xlayu@xlayu-virtual-machine:~/Desktop/lab$ riscv64-unknown-linux-gnu-gcc max.s -o max -static
xlayu@xlayu-virtual-machine:~/Desktop/lab$ qemu-riscv64 max
xlayu@xlayu-virtual-machine:~/Desktop/lab$ echo $?
5
xlayu@xlayu-virtual-machine:~/Desktop/lab$
```

图 1: 汇编程序运行结果

五、 总结

本次实验中，我们进行的工作与得到的收获如下：

- 系统了解了编译器的工作流程与各个阶段的具体工作原理、对编译器有了更为细微的认识；
- 初步接触了 llvm ir 语言，对其高级语言与汇编程序之间的桥梁形象有了一定的认识；
- 对 RISC-V 汇编语言有了初步的了解，认识了其与 x86、mips 汇编语言的一些不同之处。

对编译器与各语言的理解是学好编译原理的基础，相信我们对它们的认识也会随着课程的推进越发深刻。

六、 仓库链接

- 杨涵：<https://github.com/Polariess/Compile.git>
- 高连儒：<https://github.com/Lie-cell/compile.git>