南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译系统原理实验报告

了解编译器，LLVM IR 编程及汇编编程

年级：2022 级

专业：计算机科学与技术

学号：2210983 2213917

学号：苑译元 林逸典

指导教师：王刚 李忠伟

2024 年 9 月 22 日

摘要

本文旨在通过一系列实验，系统地探索编译系统原理，为后续编写 SysY 编译器奠定坚实基础。实验内容涵盖编译过程分析、LLVM IR 编程以及汇编编程三个部分。通过详细分析 g++,LLVM 等几种编译工具，全面理解从源代码到可执行程序的完整编译流程。学习并实践中间语言编程，根据 SysY 编译器的语言特性，编写 LLVM IR 代码，并成功编译为可执行程序进行功能验证，以加深对编译器中间表示的理解。设计并实现 SysY 语言的程序，并编写与之等价的汇编程序，利用汇编器将汇编代码转换成可执行文件并执行，通过对比验证程序行为的正确性，进一步掌握低级编程及汇编语言与高级语言之间的关系。

关键字：语言处理系统；编译器；LLVM IR；汇编语言

目录

一、 实验概要

在本次实验中，我们将进行以下三个实验，完成编译系统原理实验的预备工作，为后续的实验奠定基础：

1. 编译过程分析：以熟悉的编译工具为研究对象探究语言处理系统的完整工作过程。
2. LLVM IR 编程：学习 LLVM IR 中间语言，根据 SysY 编译器的语言特性进行 LLVM IR 编程，并编译成可执行程序执行验证。
3. 汇编编程：设计 SysY 程序，编写等价汇编程序，并用汇编器生成可执行程序执行验证。

本文的编译过程分析和 LLVM IR 编程由林逸典完成，汇编编程由苑译元完成。

(一) LLVM IR

LLVM IR（LLVM Intermediate Representation，LLVM 中间代码）是 LLVM 编译器的关键组成部分，是一种与语言和机器无关的中间表示形式，它作为前端和后端之间的桥梁，使得 LLVM 能够解析多种源语言并为多种目标生成代码。通过 LLVM IR，LLVM 能够实现跨语言、跨平台的代码生成和优化。

(二) 汇编语言

汇编语言是一种靠近底层的编程语言，它直接与计算机硬件进行交互，是对机器语言的符号化表示。每一条汇编指令通常对应一条机器指令，因此汇编语言与计算机的硬件架构紧密相关。除此之外，汇编语言还扮演着编译过程的关键角色。

在本次实验中，我们将进行 RISC-V 汇编语言编程。RISC-V（Reduced Instruction Set Computer V，第五代精简指令集计算机）汇编语言是一种与 RISC-V 指令集架构紧密相关的编程语言，是一种基于精简指令集（RISC）原则的开源指令集架构。

二、 编译过程分析

(一) 编译过程

如图1所示，编译过程由四个阶段组成：预处理器将源程序处理成预处理后的源程序，编译器将预处理后的源程序处理成汇编程序，汇编器将汇编程序处理成可重定位机器码，链接器将可重定位机器码处理成二进制可执行文件。

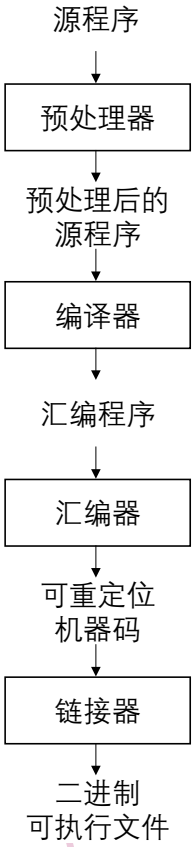

图 1: 编译过程

在本部分中，我将以一个简单的 C++ 源程序 factorial.cpp 为例，调整 g++ 和 LLVM 编译器的程序选项获得各阶段的输出，研究它们与源程序的关系。

(二) 预处理器

预处理器会处理预编译指令，主要包括将 #include 替换为对应的头文件，将 #define 替换为对应的内容，删除注释等功能。

factorial.cpp 中包含以下内容，用于测试预处理器功能。

factorial.cpp 部分代码

```
#include <iostream>
//测试预处理器对define和注释的处理
#define One 1
……
    f=One;
……
```

使用以下指令，通过 g++ 编译器将源程序编译成预处理后的源程序。

编译成预处理后的源程序

```
g++ -E factorial.cpp -o factorial.i
```

如以下代码所示，可以观察到程序将 iostream 对应的头文件替换了 #include <iostream> 由不足 100 行变为超过 28000 行, 将 #defined 定义的 One 替换为了 1，并删除了注释。

factorial.cpp 部分代码

```
……//#include <iostream>替换成对应的头文件
……
    f=1;//One被替换，注释被删除
……
```

(三) 编译器

编译器的工作原理是课程的重点内容，为了更深入地理解编译器的工作流程，我借助 LLVM 编译器，对 factorial.cpp 的编译过程进行分析，这一过程包括五个部分：词法分析、语法分析和语义分析、中间代码生成、代码优化以及代码生成。

1. 词法分析

使用以下指令，LLVM 编译器将输出词法分析结果。

输出词法分析结果的指令

```
clang -E -Xclang -dump-tokens factorial.cpp
```

如部分词法分析结果所示，在词法分析之后，程序被拆分成 token，并标明其类型和位置，以及包含 [StartOfLine]，[LeadingSpace] 等额外信息。

部分词法分析结果

```
//f=f*i;拆分成的token
identifier 'f'    [StartOfLine] [LeadingSpace]    Loc=<factorial.cpp:16:3>
equal '='                 Loc=<factorial.cpp:16:4>
identifier 'f'            Loc=<factorial.cpp:16:5>
star '*'                  Loc=<factorial.cpp:16:6>
identifier 'i'            Loc=<factorial.cpp:16:7>
semi ';'                  Loc=<factorial.cpp:16:8>
```

2. 语法分析和语义分析

语法分析使用词法分析获得的 token 来构建抽象语法树（AST 树），语法分析阶段使用语法树和符号表来检查源程序在语义上是否正确，主要包括类型检查等。

使用以下指令，LLVM 编译器将展示语法分析和语义分析的结果。

展示语法分析和语义分析结果的指令

```
clang -E -Xclang -ast-dump factorial.cpp
```

图2展示了”f=f*i;” 对应的语法分析和语义分析结果，其对应的 AST 树如图3所示。

```
|  `-CompoundStmt 0x2b1fb58 <line:15:2, line:18:2>
|    |-BinaryOperator 0x2b1fa80 <line:16:3, col:7> 'int' lvalue '='
|    | |-DeclRefExpr 0x2b1f9d0 <col:3> 'int' lvalue Var 0x2b1de28 'f' 'int'
|    | `-BinaryOperator 0x2b1fa60 <col:5, col:7> 'int' '*'
|    |   |-ImplicitCastExpr 0x2b1fa30 <col:5> 'int' <LValueToRValue>
|    |   | `-DeclRefExpr 0x2b1f9f0 <col:5> 'int' lvalue Var 0x2b1de28 'f' 'int'
|    |   `-ImplicitCastExpr 0x2b1fa48 <col:7> 'int' <LValueToRValue>
|    |     `-DeclRefExpr 0x2b1fa10 <col:7> 'int' lvalue Var 0x2b1dd28 'i' 'int'
```

图 2: 部分语法分析和语义分析结果

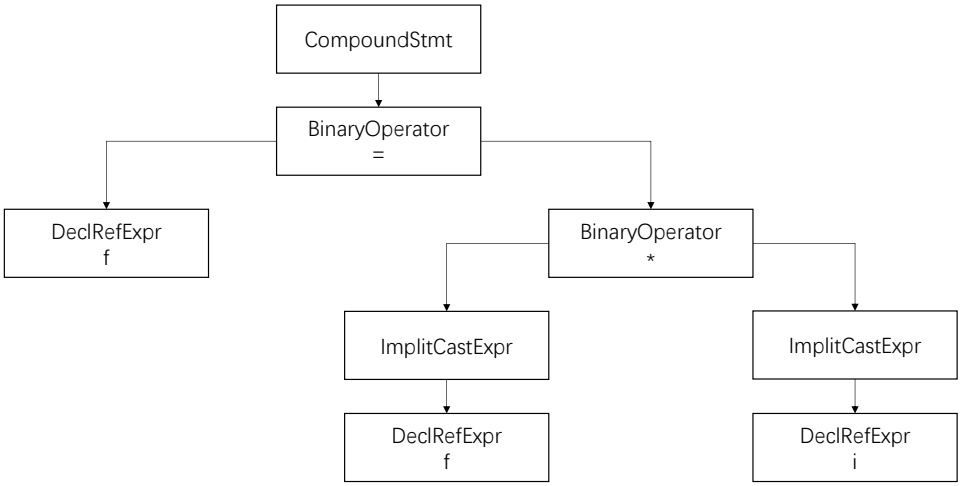

图 3: 部分语法分析和语义分析结果对应 AST 树

其中 ImplitCastExpr 节点是一个隐式类型转换表达式，它将左值转换为右值。这说明在语义分析进行了类型检查操作。

3. 中间代码生成

LLVM 可以使用以下指令生成 LLVM IR。

展示中间代码结果的指令

```
clang -S -emit-llvm factorial.cpp
```

以下是部分的 LLVM IR 结果:

部分中间代码结果

```
10:                                               ; preds = %6
  %11 = load i32, i32* %4, align 4   ; 从%4指向的地址加载当前乘积
  %12 = load i32, i32* %2, align 4   ; 从%2指向的地址加载循环变量
  %13 = mul nsw i32 %11, %12         ; 计算%11和%12的乘积
  store i32 %13, i32* %4, align 4    ; 将乘积存回%4指向的地址
  %14 = load i32, i32* %2, align 4   ; 再从%2指向的地址加载循环变量
  %15 = add nsw i32 %14, 1           ; 将循环变量增加1
  store i32 %15, i32* %2, align 4    ; 将新的循环变量存回%2指向的地址
  br label %6                        ; 跳转到循环的开始（标签%6）
```

4. 代码优化

代码优化是通过优化程序的中间代码，以减少资源消耗、提高执行效率或改善其他性能指标的优化过程。

我通过了解 LLVM 现有的三种优化 pass 进行代码优化探索。它们分别是 Analysis Passes（分析 pass），Transform Passes（转换 pass）和 Utility Passes（实用工具 pass）。

Analysis Passes 主要负责分析中间代码，获取优化信息。这些信息对于后续的 Transform Passes 来说非常有用，可以用于指导优化过程或用于调试和程序可视化。

Transform Passes 基于 Analysis Passes 提供的信息或其他启发式规则，以某种方式改变和优化中间代码。

Utility Passes 提供了一些不直接参与优化或分析的功能性实用程序。这些 pass 既不属于 Analysis Passes，也不属于 Transform Passes，但它们在编译过程中发挥着重要作用。

我还尝试阅读了 LLVM pass 源码进行代码优化探索。

5. 代码生成

使用以下指令，通过 g++ 完成编译器阶段的全部工作，实现汇编代码的生成。

编译成汇编代码

```
g++ -S factorial.cpp -o factorial.s
```

以下是部分的汇编代码结果：

部分汇编代码结果

```
.L3:
        movl    -20(%rbp), %eax
        cmpl    %eax, -16(%rbp)
        jg      .L2
        movl    -12(%rbp), %eax
        imull   -16(%rbp), %eax
        movl    %eax, -12(%rbp)
        addl    $1, -16(%rbp)
        jmp     .L3
```

（四） 汇编器

汇编器将汇编程序编译为可重定位机器码，即将辅助阅读的汇编指令转换为二进制的机器码，但这些机器码中的地址并不是最终的物理地址，而是相对于某个基准地址的偏移量。

使用以下指令，通过 g++ 编译器将源程序编译成可重定位机器码，并用 nm 辅助阅读。

编译成预处理后的源程序

```
g++ factorial.cpp -c -o factorial.o
nm factorial.o
```

可重定位机器码的部分内容如下：

可重定位机器码的部分内容

```
0000000000000000 T main
```

```
                 U __stack_chk_fail
0000000000000095 t _Z41__static_initialization_and_destruction_0ii
```

这些内容是链接器在处理可重定位机器码时需要解决的各种符号和依赖项，反映了程序在执行时所需的外部资源、库函数调用、全局变量等。

为了进一步探索汇编器的作用，由 objdump 对 factorial.o 进行反汇编，部分结果如下：

反汇编部分结果

```
3c:   8b 45 ec        mov   -0x14(%rbp),%eax   ; 将i的值加载到%eax
3f:   39 45 f0        cmp   %eax,-0x10(%rbp)   ; 比较i和n
42:   7f 10           jg    54 <main+0x54>     ; 如果i > n，跳转到循环结束
44:   8b 45 f4        mov   -0xc(%rbp),%eax    ; 将f的值加载到%eax
47:   0f af 45 f0     imul  -0x10(%rbp),%eax   ; %eax *= i（即f *= i）
4b:   89 45 f4        mov   %eax,-0xc(%rbp)    ; 将新的f值存回-0xc(%rbp)
4e:   83 45 f0 01     addl  $0x1,-0x10(%rbp)   ; i = i + 1
52:   eb e8           jmp   3c <main+0x3c>     ; 跳转到循环开始
```

可以观察到，反汇编结果与原汇编代码差别不大，最主要的差异就是原汇编代码中.L2 这类标签被 <main+0x54> 这类相对于基准地址的偏移量所取代，实现了生成可重定位机器码的功能。

（五） 链接器

链接器将多个编译或汇编生成的目标文件以及库文件链接成一个可执行文件。链接器执行验证结果如图4所示：

```
linyidian@linyidian:~/桌面/编译$ g++ factorial.cpp -o factorial
linyidian@linyidian:~/桌面/编译$ ./factorial
5
120
```

图 4: 链接器编译运行结果

由 objdump 对 factorial 进行反汇编，部分结果如下：

反汇编部分结果

```
1225:   8b 45 ec              mov   -0x14(%rbp),%eax
1228:   39 45 f0              cmp   %eax,-0x10(%rbp)
122b:   7f 10                 jg    123d <main+0x54>
122d:   8b 45 f4              mov   -0xc(%rbp),%eax
1230:   0f af 45 f0           imul  -0x10(%rbp),%eax
1234:   89 45 f4              mov   %eax,-0xc(%rbp)
1237:   83 45 f0 01           addl  $0x1,-0x10(%rbp)
123b:   eb e8                 jmp   1225 <main+0x3c>
```

可以观察到，链接器工作前后的反汇编代码发生一定变化。链接器工作后的反汇编代码更长，这是因为目标文件和库文件链接成一个可执行文件。

三、 LLVM IR 编程

在本部分中，我们将学习 LLVM IR 中间语言，根据 SysY 编译器的语言特性进行 LLVM IR 编程，并用 LLVM 的 llc 和 clang 工具以及 as 汇编器将 LLVM IR 编译成目标程序执行验证。

使用以下指令，编译并执行 LLVM IR 进行验证：

编译并执行 LLVM IR 的指令

```
llc -march=x86-64 <name>.ll -o <name>.s
as <name>.s -o <name>.o
clang <name>.o -o <name>
./<name>
```

(一) 斐波那契数列

在本节中，我们编写了斐波那契数列的 LLVM IR 形式，编译成目标程序并执行验证，以探索 SysY 编译器的整数基本运算和循环语句等语言特性。

斐波那契数列 LLVM IR 形式

```
define i32 @main() #0 {
        ; 声明变量:int a,b,i,t,n;
        %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
        ; 初始化:a=0;b=1;i=1;
        store i32 0, i32* %1, align 4
        store i32 1, i32* %2, align 4
        store i32 1, i32* %3, align 4
        ; 输入输出: cin>>n; cout<<a<<endl; cout<<b<<endl;
        %6 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds
            ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* %5)
        %7 = load i32, i32* %1, align 4
        %8 = load i32, i32* %2, align 4
        %9  = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
            i8], [3 x i8]* @.str.1, i64 0, i64 0), i32 %7)
        %10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8
            ], [3 x i8]* @.str.1, i64 0, i64 0), i32 %8)
        br label %11
11:                                                ;
    preds=%0,%15
        ;循环的跳转:while(i<n)
        %12 = load i32, i32* %3, align 4
        %13 = load i32, i32* %5, align 4
        %14 = icmp slt i32 %12, %13
        br i1 %14, label %15, label %22

```

```
15:                                                ;
    preds=%11
        ; a+b
        %16 = load i32, i32* %1, align 4
        %17 = load i32, i32* %2, align 4
        %18 = add i32 %16, %17
        ; cout<<a+b<<"␣";
        %19 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8
            ], [3 x i8]* @.str.1, i64 0, i64 0), i32 %18)
        ; 更新变量 t=b; b=a+b; a=t; i=i+1;
        store i32 %17, i32* %4, align 4
        store i32 %18, i32* %2, align 4
        store i32 %17, i32* %1, align 4
        %20 = load i32, i32* %3, align 4
        %21 = add nsw i32 %20, 1
        store i32 %21, i32* %3, align 4
        br label %11
22:                                                ;
    preds=%11
    ;return 0;
        ret i32 0
}
; 函数声明
declare dso_local i32 @__isoc99_scanf(i8*, ...)
declare dso_local i32 @printf(i8*, ...)
@.str = private constant [3 x i8] c"%d\00"
@.str.1 = private constant [3 x i8] c"%d␣"
```

测试样例:

输入

```
5
```

输出

```
0 1 1 2 3 5
```

执行验证结果如图5所示，与预期结果一致，LLVM IR 程序正确。

```
linyidian@linyidian:~/桌面/编译$ llc -march=x86-64 Fibonacci.ll -o Fibonacci.s
linyidian@linyidian:~/桌面/编译$ as Fibonacci.s -o Fibonacci.o
linyidian@linyidian:~/桌面/编译$ clang Fibonacci.o -o Fibonacci
linyidian@linyidian:~/桌面/编译$ ./Fibonacci
5
0 1 1 2 3 5 linyidian@linyidian:~/桌面/编译$
```

图 5: 斐波那契数列 LLVM IR 编译运行结果

(二) 浮点数基本运算

在本节中，我们设计了 SysY 语言浮点数基本运算的例子，并编写了相应的 LLVM IR 形式，编译成目标程序并执行验证，以探索 SysY 编译器的浮点数基本运算等语言特性。

SysY 语言浮点数基本运算的例子

```
#include <stdio.h>
int main()
{
    float a,b,add,sub,mul,div;
    scanf("%f%f",&a,&b);
    add=a+b;
    sub=a-b;
    mul=a*b;
    div=a/b;
    printf("%.2f %.2f %.2f %.2f",add,sub,mul,div);
    return 0;
}
```

浮点数基本运算 LLVM IR 形式

```
define i32 @main() #0 {
  ; 声明变量:float a,b,add,sub,mul,div;
  %1 = alloca float, align 4
  %2 = alloca float, align 4
  %3 = alloca float, align 4
  %4 = alloca float, align 4
  %5 = alloca float, align 4
  %6 = alloca float, align 4
  ; scanf("%f%f",&a,&b);
  %7 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([5 x
      i8], [5 x i8]* @.str, i64 0, i64 0), float* %1, float* %2)
  ; add=a+b;
  %8 = load float, float* %1, align 4
  %9 = load float, float* %2, align 4
  %10 = fadd float %8, %9
  store float %10, float* %3, align 4
  ; sub=a-b;
  %11 = load float, float* %1, align 4
  %12 = load float, float* %2, align 4
  %13 = fsub float %11, %12
  store float %13, float* %4, align 4
  ; mul=a*b;
  %14 = load float, float* %1, align 4
  %15 = load float, float* %2, align 4
  %16 = fmul float %14, %15
  store float %16, float* %5, align 4
  ; div=a/b;
  %17 = load float, float* %1, align 4
  %18 = load float, float* %2, align 4
  %19 = fdiv float %17, %18
  store float %19, float* %6, align 4
  ; printf("%.2f %.2f %.2f %.2f",add,sub,mul,div);
```

```
  ;在C语言中printf会自动将float转为double，在LLVM IR中要手动进行
  %20 = load float, float* %3, align 4
  %21 = fpext float %20 to double
  %22 = load float, float* %4, align 4
  %23 = fpext float %22 to double
  %24 = load float, float* %5, align 4
  %25 = fpext float %24 to double
  %26 = load float, float* %6, align 4
  %27 = fpext float %26 to double
  %28 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([20 x i8],
      [20 x i8]* @.str.1, i64 0, i64 0), double %21, double %23, double %25,
      double %27)
return 0;
  ret i32 0
}
; 函数声明
declare dso_local i32 @__isoc99_scanf(i8*, ...) #1
declare dso_local i32 @printf(i8*, ...) #1
@.str = private unnamed_addr constant [5 x i8] c"%f%f\00", align 1
@.str.1 = private unnamed_addr constant [20 x i8] c"%.2f␣%.2f␣%.2f␣%.2f\00",
    align 1
```

测试样例：

输入

```
2.5 0.5
```

输出

```
3.00 2.00 1.25 5.00
```

执行验证结果如图6所示，与预期结果一致，LLVM IR 程序正确。

```
linyidian@linyidian:~/桌面/编译$ llc -march=x86-64 float.ll -o float.s
linyidian@linyidian:~/桌面/编译$ as float.s -o float.o
linyidian@linyidian:~/桌面/编译$ clang float.o -o float
linyidian@linyidian:~/桌面/编译$ ./float
2.5 0.5
3.00 2.00 1.25 5.00linyidian@linyidian:~/桌面/编译$
```

图 6: 浮点数基本运算 LLVM IR 编译运行结果

(三) 数组与常值

在本节中，我们设计了 SysY 语言关于数组与常值的例子，并编写了相应的 LLVM IR 形式，编译成目标程序并执行验证，以探索 SysY 编译器的数组与常值等语言特性。

SysY 语言数组与常值的例子

```
#include <stdio.h>
int main()
{
```

```
    const int n=5;
    int A[5];
    for(int i=0;i<n;i++)
    {
        scanf("%d",&A[i]);
    }
    for(int i=n-1;i>=0;i--)
    {
        printf("%d␣",A[i]);
    }
    return 0;
}
```

数组与常值 LLVM IR 形式

```
define i32 @main() #0 {
  ;const int n=5;
  ;int A[5];
  ;循环的变量声明:for(int i=0;;)
  ;循环的变量声明:for(int i=4;;)
  %1 = alloca [5 x i32], align 16
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %2, align 4
  store i32 4, i32* %3, align 4
  br label %4
;循环的跳转:for(;i<n;)
4:                                                ; preds = %12, %0
  %5 = load i32, i32* %2, align 4
  %6 = icmp slt i32 %5, 5
  br i1 %6, label %7, label %15
;输入的循环体
7:                                                ; preds = %4
  %8 = load i32, i32* %2, align 4
  %9 = sext i32 %8 to i64
  %10 = getelementptr inbounds [5 x i32], [5 x i32]* %1, i64 0, i64 %9
  %11 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x
      i8], [3 x i8]* @.str, i64 0, i64 0), i32* %10)
  br label %12
;循环变量运算:for(;;i++)
12:                                               ; preds = %7
  %13 = load i32, i32* %2, align 4
  %14 = add nsw i32 %13, 1
  store i32 %14, i32* %2, align 4
  br label %4
;循环的跳转:for(;i>=0;)
15:                                               ; preds = %24, %4
  %16 = load i32, i32* %3, align 4
  %17 = icmp sge i32 %16, 0
```

```
  br i1 %17, label %18, label %27
;输出的循环体
18:                                               ; preds = %15
  %19 = load i32, i32* %3, align 4
  %20 = sext i32 %19 to i64
  %21 = getelementptr inbounds [5 x i32], [5 x i32]* %1, i64 0, i64 %20
  %22 = load i32, i32* %21, align 4
  %23 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4
      x i8]* @.str.1, i64 0, i64 0), i32 %22)
  br label %24
;循环变量运算:for(;;i--)
24:                                               ; preds = %18
  %25 = load i32, i32* %3, align 4
  %26 = add nsw i32 %25, -1
  store i32 %26, i32* %3, align 4
  br label %15
27:                                               ; preds = %15
  ;return 0;
  ret i32 0
}
; 函数声明
declare dso_local i32 @__isoc99_scanf(i8*, ...) #1
declare dso_local i32 @printf(i8*, ...) #1
@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@.str.1 = private unnamed_addr constant [4 x i8] c"%d␣\00", align 1
```

测试样例:

输入

```
1 2 3 4 5
```

输出

```
5 4 3 2 1
```

执行验证结果如图7所示，与预期结果一致，LLVM IR 程序正确。

```
linyidian@linyidian:~/桌面/编译$ llc -march=x86-64 array.ll -o array.s
linyidian@linyidian:~/桌面/编译$ as array.s -o array.o
linyidian@linyidian:~/桌面/编译$ clang array.o -o array
linyidian@linyidian:~/桌面/编译$ ./array
1 2 3 4 5
5 4 3 2 1 linyidian@linyidian:~/桌面/编译$
```

图 7: 数组与常值 LLVM IR 编译运行结果

(四) 隐式类型转换

在本节中，我们设计了 SysY 语言关于隐式类型转换的例子，并编写了相应的 LLVM IR 形式，编译成目标程序并执行验证，以探索 SysY 编译器的隐式类型转换等语言特性。

SysY 隐式类型转换的例子

```c
#include <stdio.h>
int main()
{
    float a;
    int b;
    scanf("%f",&a);
    b=a;
    printf("%d",b);
    return 0;
}
```

隐式类型转换 LLVM IR 形式

```
define i32 @main() #0 {
  ;声明变量float a;int b;
  %1 = alloca float, align 4
  %2 = alloca i32, align 4
  ;scanf("%f",&a);
  %3 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x
      i8], [3 x i8]* @.str, i64 0, i64 0), float* %1)
  %4 = load float, float* %1, align 4
  ;隐式类型转换
  %5 = fptosi float %4 to i32
  ;b=a;
  store i32 %5, i32* %2, align 4
  ;printf("%d",b);
  %6 = load i32, i32* %2, align 4
  %7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x
       i8]* @.str.1, i64 0, i64 0), i32 %6)
;return 0;
  ret i32 0
}
; 函数声明
declare dso_local i32 @__isoc99_scanf(i8*, ...) #1
declare dso_local i32 @printf(i8*, ...) #1
@.str = private unnamed_addr constant [3 x i8] c"%f\00", align 1
@.str.1 = private unnamed_addr constant [3 x i8] c"%d\00", align 1
```

测试样例:

输入

```
1.25
```

输出

```
1
```

执行验证结果如图8所示，与预期结果一致，LLVM IR 程序正确。

```
linyidian@linyidian:~/桌面/编译$ llc -march=x86-64 implicit.ll -o implicit.s
linyidian@linyidian:~/桌面/编译$ as implicit.s -o implicit.o
linyidian@linyidian:~/桌面/编译$ clang implicit.o -o implicit
linyidian@linyidian:~/桌面/编译$ ./implicit
1.25
1linyidian@linyidian:~/桌面/编译$
```

图 8: 隐式类型转换 LLVM IR 编译运行结果

(五) 总结

在本部分中，我们学习了 LLVM IR 中间语言，进行 LLVM IR 编程，探索了 SysY 编译器的语言特性在 LLVM IR 中间语言的表现。

对于 SysY 语言支持的 int 类型和 float 类型，在 LLVM IR 中表示为 i32 和 float，相应的运算类似于 add 和 fadd 等，如以下 LLVM 片段所示:

int 类型和 float 类型相关的 LLVM 片段

```
%1 = alloca int, align 4      ;声明int类型变量
%1 = alloca float, align 4    ;声明float类型变量
%18 = add i32 %16, %17        ;进行int类型运算
%10 = fadd float %8, %9       ;进行float类型运算
```

对于 SysY 语言支持的数组, 在 LLVM IR 中以 [n x i32] 形式声明, 运算时通过 getelementptr inbounds 获取指针，如以下 LLVM 片段所示:

数组相关的 LLVM 片段

```
;声明int类型数组变量（大小为5）
%1 = alloca [5 x i32], align 16
;获取对应的数组的元素的指针（%1[0][%9]）
%10 = getelementptr inbounds [5 x i32], [5 x i32]* %1, i64 0, i64 %9
```

对于 SysY 语言支持的常值，在 LLVM IR 中会被直接替换（根据之前的学习，可能在预处理阶段已经进行了替换）

对于 SysY 语言支持的隐式类型转换，在 LLVM IR 中依靠 fptosi 完成，如以下 LLVM 片段所示:

隐式类型转换相关的 LLVM 片段

```
%5 = fptosi float %4 to i32
```

综上所述，我们通过 LLVM IR 编程熟悉了 LLVM IR 中间语言，探索了 SysY 的多方面特性。

四、 汇编编程

在本部分中，我们设计 SysY 程序，编写等价 RISC-V 汇编程序，并用汇编器生成可执行程序执行验证。使用以下指令，编译并执行 RISC-V 汇编程序进行验证:

编译并测试 riscv 汇编的指令

```
riscv64-unknown-linux-gnu-gcc <name>.s -o <name> -static
qemu-riscv64 <name>
```

(一) 斐波那契数列

在本节中，我们仍然以斐波那契数列为例子，编写 RISC-V 汇编代码。

斐波那契数列的汇编代码

```
    .data
prompt:
    .asciz "Enter a number: "
format_in:  .asciz "%d"
format_out:  .asciz "%d\n"
    .text
    .globl main
main:
    # 函数前序，设置栈帧
    addi    sp, sp, -32         # 分配32字节的栈空间
    sw      ra, 28(sp)          # 保存返回地址在 sp + 28
    sw      s0, 24(sp)          # 保存帧指针在 sp + 24
    mv      s0, sp              # 设置帧指针 s0 = sp
    # 初始化变量
    li      t0, 0               # a = 0
    sw      t0, 16(s0)
    li      t0, 1               # b = 1
    sw      t0, 12(s0)
    li      t0, 1               # i = 1
    sw      t0, 8(s0)
    # 输出提示信息
    la      a0, prompt
    call    printf
    # 读取用户输入的 n
    la      a0, format_in
    addi    a1, s0, 0           # &n
    call    scanf
    # 打印初始的两个数字 a 和 b
    lw      a1, 16(s0)          # a1 = a
    la      a0, format_out
    call    printf
    lw      a1, 12(s0)          # a1 = b
    la      a0, format_out
    call    printf
    # 开始循环
loop_start:
    lw      t0, 8(s0)           # t0 = i
    lw      t1, 0(s0)           # t1 = n
    bge     t0, t1, loop_end    # 如果 i >= n, 跳出循环
    # t = b;
    lw      t2, 12(s0)          # t2 = b
    sw      t2, 4(s0)           # t = t2
    # b = a + b;
    lw      t3, 16(s0)          # t3 = a
```

```
    add     t2, t2, t3          # t2 = b + a
    sw      t2, 12(s0)          # b = t2
    # 打印新的 b 值
    lw      a1, 12(s0)          # a1 = b
    la      a0, format_out
    call    printf
    # a = t;
    lw      t2, 4(s0)           # t2 = t
    sw      t2, 16(s0)          # a = t2
    # i = i + 1;
    lw      t0, 8(s0)           # t0 = i
    addi    t0, t0, 1           # t0 = t0 + 1
    sw      t0, 8(s0)           # i = t0
    j       loop_start          # 回到循环开始
loop_end:
    # 函数后序，恢复栈帧
    lw      ra, 28(sp)          # 恢复返回地址
    lw      s0, 24(sp)          # 恢复帧指针
    addi    sp, sp, 32          # 释放栈空间
    ret                         # 返回
```

测试样例:

测试输入

```
5
```

目标输出

```
0
1
1
2
3
5
```

测试输入和结果输出如图9和图10所示，结果符合预期，汇编程序正确。

```
yuan@yuan-virtual-machine:~/lab1$ riscv64-unknown-linux-gnu-gcc testt.s -o test
-static
yuan@yuan-virtual-machine:~/lab1$ qemu-riscv64 test
Enter a number:
```

图 9: 测试输入

```
yuan@yuan-virtual-machine:~/lab1$ riscv64-unknown-linux-gnu-gcc testt.s -o test
-static
yuan@yuan-virtual-machine:~/lab1$ qemu-riscv64 test
Enter a number: 5
0
1
1
2
3
5
```

图 10: 结果输出

(二) 浮点数与数组

在本节的实验中，我们编写了在数组中存入五个浮点数并求和的 RISC-V 汇编代码。

SysY 浮点数与数组代码

```
#include <stdio.h>
int main() {
    float numbers[5];
    float sum = 0.0;
    printf("Please enter 5 floating-point numbers:\n");
    for (int i = 0; i < 5; i++) {
        printf("Enter number %d: ", i + 1);
        scanf("%f", &numbers[i]);
        sum += numbers[i];
    }
    printf("The sum of the 5 floating-point numbers is: %.2f\n", sum);
    return 0;
}
```

浮点数与数组汇编代码

```
    .data
    .align 3
    # 提示信息
prompt_msg:     .asciz "Please enter 5 floating-point numbers:"
    # 输入提示
input_msg:      .asciz "Enter number %d: "
    # scanf格式
format_float:   .asciz "%lf"
    # 输出提示
sum_msg:        .asciz "The sum of the 5 floating-point numbers is: %.2f\n"
    .text
    .align 1
    .globl main
    .type main, @function

main:
    # 保存返回地址和帧指针
    # 分配栈空间: 80字节（浮点数存储）+ 128字节（可变参数区域）
```

```
    addi sp, sp, -208
    sd ra, 200(sp)              # 保存返回地址
    sd s0, 192(sp)              # 保存帧指针
    addi s0, sp, 208            # 设置帧指针为栈顶

    # 保存需要在函数调用后保持的寄存器
    sd s1, 184(sp)              # 保存 s1（循环计数器）
    fsd fs0, 176(sp)            # 保存 fs0（浮点数总和）

    # 初始化总和为0.0，初始化循环计数器
    fmv.d.x fs0, zero           # 将 fs0 初始化为 0.0（总和，双精度）
    li s1, 0                    # 循环计数器 s1 = 0

    # 打印提示信息
    lui a5, %hi(prompt_msg)
    addi a0, a5, %lo(prompt_msg)
    call puts

loop_input:
    li t0, 5                    # t0 = 5，循环次数
    bge s1, t0, sum_output      # 如果 s1 >= 5，跳转到 sum_output

    # 提示用户输入第几个数
    lui a5, %hi(input_msg)
    addi a0, a5, %lo(input_msg)
    addi a1, s1, 1              # a1 = s1 + 1
    call printf

    # 为可变参数函数调整栈指针，使其16字节对齐
    addi sp, sp, -16            # 调整栈指针

    # 读取用户输入的浮点数
    lui a5, %hi(format_float)
    addi a0, a5, %lo(format_float)

    # 计算存储输入浮点数的地址
    addi t1, s0, -208           # t1 = s0 - 208（浮点数数组起始地址）
    slli t2, s1, 3              # t2 = s1 * 8（偏移量，双精度浮点数占8字节）
    add t3, t1, t2              # t3 = t1 + t2（当前浮点数的存储地址）
    mv a1, t3                   # a1 = t3

    call scanf

    addi sp, sp, 16             # 恢复栈指针

    # 计算存储输入浮点数的地址
    addi t1, s0, -208           # t1 = s0 - 208（浮点数数组起始地址）
    slli t2, s1, 3              # t2 = s1 * 8（偏移量，双精度浮点数占8字节）
```

```
    add t3, t1, t2              # t3 = t1 + t2 (当前浮点数的存储地址)
    mv a1, t3                   # a1 = t3

    # 从栈中加载浮点数并累加到总和
    fld ft0, 0(t3)              # 加载输入的浮点数到 ft0
    fadd.d fs0, fs0, ft0        # 累加到总和 fs0

    # 循环计数器加1
    addi s1, s1, 1
    j loop_input

sum_output:
    # 输出总和
    lui a5, %hi(sum_msg)
    addi a0, a5, %lo(sum_msg)

    # 为可变参数函数调整栈指针, 使其16字节对齐
    addi sp, sp, -16

    # 准备浮点数参数
    fmv.x.d a1,fs0              # 将总和 fs0 移动到 fa0, 供 printf 使用

    call printf

    addi sp, sp, 16             # 恢复栈指针

    # 恢复保存的寄存器和栈指针
    fld fs0, 176(sp)            # 恢复 fs0
    ld s1, 184(sp)              # 恢复 s1
    ld s0, 192(sp)              # 恢复 s0
    ld ra, 200(sp)              # 恢复 ra
    addi sp, sp, 208            # 恢复栈指针
    li a0, 0                    # 返回值 0
    ret
```

测试样例:

测试输入

```
2.0
2.0
2.0
2.0
2.0
```

目标输出

```
10.00
```

测试输入和结果输出如图11和图12所示，结果符合预期，汇编程序正确。

```
yuan@yuan-virtual-machine:~/lab1/better$ riscv64-unknown-linux-gnu-gcc testt.s -
o test -static
yuan@yuan-virtual-machine:~/lab1/better$ qemu-riscv64 test
Please enter 5 floating-point numbers:
Enter number 0:
```

图 11: 测试输入

```
yuan@yuan-virtual-machine:~/lab1/better$ riscv64-unknown-linux-gnu-gcc testt.s -
o test -static
yuan@yuan-virtual-machine:~/lab1/better$ qemu-riscv64 test
Please enter 5 floating-point numbers:
Enter number 1: 2.0
Enter number 2: 2.0
Enter number 3: 2.0
Enter number 4: 2.0
Enter number 5: 2.0
The sum of the 5 floating-point numbers is: 10.00
yuan@yuan-virtual-machine:~/lab1/better$
```

图 12: 结果输出

(三) 总结

在本部分中，我们设计了两段 SysY 程序，编写了等价 RISC-V 汇编程序，并通过编译和执行验证其正确性。

在进行汇编编程的过程中遇到了许多的困难，如数组越界，堆栈溢出，内存管理等问题。除此之外，还有类似数据竞争现象的罕见错误。

整个实验过程中，我们严格遵守 RISC-V 汇编语言的语法规范编写代码。我们还使用汇编器生成可执行程序，并通过 QEMU 进行模拟执行。测试结果与预期完全一致，证明了所编写的汇编程序在功能上的正确性。

通过本部分的实验，我们深入理解了 RISC-V 汇编语言的编程方法和调试技巧，为进一步的汇编编程打下了坚实的基础。

五、 总结

在本次实验中，我们探索了编译系统原理的多个关键方面。

在编译过程分析实验中，我们以熟悉的编译工具为研究对象探究语言处理系统的完整工作过程，详细分解了源代码从预处理、编译、汇编到链接的整个过程。

在 LLVM IR 编程实验中，我们学习了 LLVM IR 中间语言，根据 SysY 编译器的语言特性编写了四个 LLVM IR 程序，并编译成可执行程序执行验证。

在汇编编程实验中，我们深入学习了 RISC-V 汇编语言，并设计了 SysY 程序，编写等价汇编程序，并用汇编器生成可执行程序执行验证。

通过编译过程分析，LLVM IR 编程和汇编编程等实验环节，我们加深了对编译过程的理解，掌握了 LLVM IR 编程和汇编语言编程的基本技能，为后续编写 SysY 编译器奠定坚实基础。

本次实验的代码已上传至 GitHub。