南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译系统调研报告

了解编译器 & LLVM IR 编程 & 汇编编程

姓名：姚知言

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2025 年 1 月 26 日

摘要

本文以 GCC、LLVM/Clang 为研究对象，探究了语言处理系统的具体工作过程。以斐波那契数列、数组、循环等典型程序为例，根据编译器每个阶段的输出，分析预处理器、编译器、汇编器、连接器的作用，深入理解编译器各阶段的功能。探究了 SysY 编译器各语言的特性，通过 LLVM IR 编程探究了 LLVM IR 语言实现 SysY 的函数定义和语句的方式和 SysY 中常量与变量的声明、表达式和数组的特性。最后设计了循环求解圆周率 π 的近似值与冒泡排序两个 SysY 程序，编写了等价的 RISC-V 汇编程序，用汇编器生成可执行程序，测试通过。

在本次调研中，LLVM IR 编程由聂嘉欣同学负责，其余部分由本人完成。

关键词：GCC ；LLVM/Clang ；语言处理系统；LLVM IR 编程；RISC-V 汇编

目录

一、 语言处理系统的工作过程探究

(一) 总览

C 语言转换为可执行程序需要通过预处理器、编译器、汇编器、链接器四个部分，具体流程如图1所示。

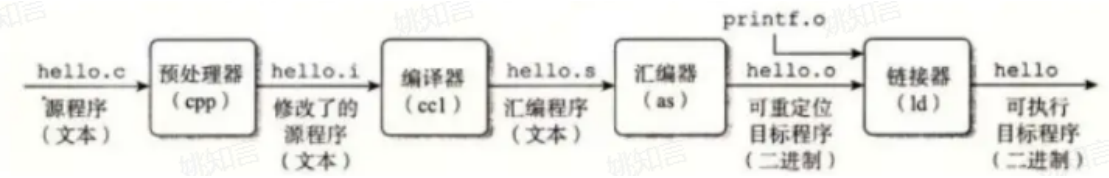


图 1: C 语言转换为可执行程序的全过程

(二) 演示程序说明

本文中所用到的演示程序说明如下。

1.c(斐波那契数列)

```c
#include<stdio.h>
int main() {
        int a, b, i, t, n;
         a = 0;
         b = 1;
         i = 1;
         scanf("%d",&n);
         printf("%d\n",a);
         printf("%d\n",b);
         while (i < n){
                  t = b;
                  b = a + b;
                  printf("%d\n",b);
                  a = t;
                  i = i + 1;
         }
         return 0;
}
```

2.c

```c
#include<stdio.h>
int main(){
         int a=1;
         return 0;
}
```

3.cpp

```cpp
#include<iostream>
```

```cpp
using namespace std;
int main(){
        int a=1;
        return 0;
}
```

4.c

```cpp
#include<stdio.h>
#define a 426
const int t=3;
#define b 999
int main(){
        int p=a; //111
        int q=a*b+t;
        return 0;
}
```

在 5.c&6.c 主要考虑上文中没有涉及的 for 循环、浮点数、数组等。

5.c & 6.c

```cpp
#include<stdio.h>
float c=3.88;
int main(){
        int a[3],b=2;
        a[2]=1;
        for(int i=0;i<15;i++){
                if(c==1)break;
                if(a[2]==10)continue;
                a[2]++;
                c+=c/b;
        }
        printf("a[2]=%d␣c=%f",a[2],c)
        //5.c此处的分号为故意缺省，在词法分析和语法分析中测试使用
        //6.c中补充此分号，有无此分号为两文件唯一差别
        return 0;
}
```

（三） 预处理器

在本部分中，利用 gcc 编译器的：*gcc 1.c −E −o 1.i* 指令进行预处理器分析。

将 1.c 和 2.c 进行预处理，发现生成的 1.i 与 2.i 前 735 行完全相同，为 *#include < stdio.h >* 中的内容，而后续内容则为原本 *#include* 之后的内容。对 3.cpp 进行预处理，同样可以发现类似的现象，然而 *iostream* 预处理后的结果多达 32251 行。

部分结果展示如下：

1.i

```
# 0 "1.c"
```

```
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2  //前5行在3个文件中均相同，意味着一些预定义宏的调用
# 1 "1.c"
//(省去约700行) 在这一部分中，完成了stdio.h的预处理(3.cpp完成了iostream的预处理)
# 2 "1.c" 2
# 3 "1.c"
int main() {//对于1.i 后续为编写的主函数部分(事实上，include以外的部分均在此体现，不限于函数，如在3.i中该部分包含了using namespace std)
```

不只限于 *#include*，几乎所有 # 开头的指令都会在这一部分完成处理。此外，源程序中的注释也会在这一部分被删去。

这并不受到在文中出现先后的限制，即 # 指令并不必须位于其他指令之前。注释的删除和 *#define* 宏定义的替换可以从 *4.i* 中发现，以下展示了 *4.i* 最后一部分的结果 (对应 1.i 第 10 行位置)。

4.i

```
//省略前面内容
const int t=3;
                    //这里有一个空行是因为原本的#define已经被处理
int main(){
 int p=426;
 int q=426*999 +t;//宏定义已经被成功替换
 return 0;
}
```

总的来说，预处理器完成了注释的删除以及各类 # 指令的处理，其中主要包括 *#define* 的宏定义展开，*#ifdef* 等条件编译，*#include* 的文件包含，还包括 *#pragma* 等其他具有特殊功能的 # 指令。

（四） 编译器

编译器所实现的功能主要是将预处理器处理后的 *.i* 文件处理成汇编语言（*ARM*, *RISCV* 等），共分为以下六个部分：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。

在此部分主要通过 *LLVM* 进行分析。

a) 词法分析

通过 $clang\ -E\ -Xclang\ -dump-tokens\ 1.c\ >\ 1-cf.log\ 2>\&1$ 将 *1.c* 的词法分析结果保存下来。

跳过前面内部库的词法分析，观察源程序的部分，可以发现主要包括以下几个组成部分。

1-cf.log

```
//关键字(如int)、运算符(如‘+’)、界符(如‘(’)等 一般一字一种
int 'int'       [StartOfLine]  Loc=<1.c:3:1> //startofline表示在一行的开始
```

```
l_paren '('             Loc=<1.c:3:9>
r_paren ')'             Loc=<1.c:3:10>
l_brace '{'      [LeadingSpace] Loc=<1.c:3:12> //leadingspace表示其之前有空格
equal '='        [LeadingSpace] Loc=<1.c:6:4>
eof ''          Loc=<1.c:19:2> //文档末尾终止符
//......
//标识符identifier
identifier 'main'        [LeadingSpace] Loc=<1.c:3:5>
identifier 'a'   [LeadingSpace] Loc=<1.c:4:13>
//......
//常数 numeric_constant
numeric_constant '1'     [LeadingSpace] Loc=<1.c:16:11>
//......
//字符串 string_literal
string_literal '"%d\n"'          Loc=<1.c:14:10>
//......
//一条完整的语句示例(以printf所示)
identifier 'printf'      [StartOfLine] [LeadingSpace]   Loc=<1.c:14:3>
l_paren '('             Loc=<1.c:14:9>
string_literal '"%d\n"'          Loc=<1.c:14:10>
comma ','               Loc=<1.c:14:16>
identifier 'b'          Loc=<1.c:14:17>
r_paren ')'             Loc=<1.c:14:18>
semi ';'                Loc=<1.c:14:19>
```

对 5.*c* 进行分析，数组部分结果如下:

5-cf.log

```
//以a[2]++;为例, 实际上与上文类似
identifier 'a'   [StartOfLine] [LeadingSpace]   Loc=<5.c:9:3>
l_square '['            Loc=<5.c:9:4>
numeric_constant '2'            Loc=<5.c:9:5>
r_square ']'            Loc=<5.c:9:6>
plusplus '++'           Loc=<5.c:9:7>
semi ';'                Loc=<5.c:9:9>
```

总之，词法分析是将源程序变成 *token* 的过程，而各 *token* 也已经被分类成不同的类型。

b) 语法分析

在语法分析中，主要是将词法分析中分类的 *token* 构建成抽象语法树 *AST*，构建抽象语法树需要确定好文法，然后逐步将词法分析后的单词串，通过自顶向下或者自底向上的方法构造语法分析树。

通过 $clang\ -E\ -Xclang\ -ast-dump\ 1.c\ >\ 1-yf.log\ 2>\&1$ 将 1.c 的语法分析结果保存下来，其中 while 循环部分 (对应 1.c 的 10-16 行所构建的抽象语法树如图2所示。

```
|-WhileStmt 0x1a33b98 <line:11:2, line:17:2>
| |-BinaryOperator 0x1a337c0 <line:11:9, col:13> 'int' '<'
| | |-ImplicitCastExpr 0x1a33790 <col:9> 'int' <LValueToRValue>
| | | `-DeclRefExpr 0x1a33750 <col:9> 'int' lvalue Var 0x1a32ba8 'i' 'int'
| | `-ImplicitCastExpr 0x1a337a8 <col:13> 'int' <LValueToRValue>
| |   `-DeclRefExpr 0x1a33770 <col:13> 'int' lvalue Var 0x1a32ca8 'n' 'int'
| `-CompoundStmt 0x1a33b60 <col:15, line:17:2>
|   |-BinaryOperator 0x1a33838 <line:12:3, col:7> 'int' '='
|   | |-DeclRefExpr 0x1a337e0 <col:3> 'int' lvalue Var 0x1a32c28 't' 'int'
|   | `-ImplicitCastExpr 0x1a33820 <col:7> 'int' <LValueToRValue>
|   |   `-DeclRefExpr 0x1a33800 <col:7> 'int' lvalue Var 0x1a32b28 'b' 'int'
|   |-BinaryOperator 0x1a33908 <line:13:3, col:11> 'int' '='
|   | |-DeclRefExpr 0x1a33858 <col:3> 'int' lvalue Var 0x1a32b28 'b' 'int'
|   | `-BinaryOperator 0x1a338e8 <col:7, col:11> 'int' '+'
|   |   |-ImplicitCastExpr 0x1a338b8 <col:7> 'int' <LValueToRValue>
|   |   | `-DeclRefExpr 0x1a33878 <col:7> 'int' lvalue Var 0x1a32aa8 'a' 'int'
|   |   `-ImplicitCastExpr 0x1a338d0 <col:11> 'int' <LValueToRValue>
|   |     `-DeclRefExpr 0x1a33898 <col:11> 'int' lvalue Var 0x1a32b28 'b' 'int'
|   |-CallExpr 0x1a339b8 <line:14:3, col:18> 'int'
|   | |-ImplicitCastExpr 0x1a339a0 <col:3> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
|   | | `-DeclRefExpr 0x1a33928 <col:3> 'int (const char *, ...)' Function 0x1a17a28 'printf' 'int (const char *, ...)'
|   | |-ImplicitCastExpr 0x1a33a00 <col:10> 'const char *' <NoOp>
|   | | `-ImplicitCastExpr 0x1a339e8 <col:10> 'char *' <ArrayToPointerDecay>
|   | |   `-StringLiteral 0x1a33948 <col:10> 'char[4]' lvalue "%d\n"
|   | `-ImplicitCastExpr 0x1a33a18 <col:17> 'int' <LValueToRValue>
|   |   `-DeclRefExpr 0x1a33968 <col:17> 'int' lvalue Var 0x1a32b28 'b' 'int'
|   |-BinaryOperator 0x1a33a88 <line:15:3, col:7> 'int' '='
|   | |-DeclRefExpr 0x1a33a30 <col:3> 'int' lvalue Var 0x1a32aa8 'a' 'int'
|   | `-ImplicitCastExpr 0x1a33a70 <col:7> 'int' <LValueToRValue>
|   |   `-DeclRefExpr 0x1a33a50 <col:7> 'int' lvalue Var 0x1a32c28 't' 'int'
|   `-BinaryOperator 0x1a33b40 <line:16:3, col:11> 'int' '='
|     |-DeclRefExpr 0x1a33aa8 <col:3> 'int' lvalue Var 0x1a32ba8 'i' 'int'
|     `-BinaryOperator 0x1a33b20 <col:7, col:11> 'int' '+'
|       |-ImplicitCastExpr 0x1a33b08 <col:7> 'int' <LValueToRValue>
|       | `-DeclRefExpr 0x1a33ac8 <col:7> 'int' lvalue Var 0x1a32ba8 'i' 'int'
|       `-IntegerLiteral 0x1a33ae8 <col:11> 'int' 1
```

图 2: 1.c 中 while 循环 (10-16 行) 抽象语法树

对于 5.*c* 进行分析，可以看到在文件开头成功检测到了错误。

5-yf.log

```
5.c:12:31: error: expected ';' after expression
        printf("a[2]=%d␣c=%f",a[2],c)
                                    ^
                                    ;
```

将 $5-yf.log$ 与 $6-yf.log$ 进行对比，发现（不包括报错部分）除去内存地址有变化之外转换结果并没有较大差别，以原文中第 9 行 $a[2]++$ 展开讨论。

- |-UnaryOperator 0xd22570 <line:9:3, col:7> 'int' postfix '++'

  UnaryOperator 表示是一个一元运算符，然后是它的内存地址和在源代码的位置，最后表示其是后缀 ++ 运算符。

- | | '-ArraySubscriptExpr 0xd22550 <col:3, col:6> 'int' lvalue

  表示对数组的访问。

- | | |-ImplicitCastExpr 0xd22538 <col:3> 'int *' <ArrayToPointerDecay>

  表示一个类型转换。

- | | | '-DeclRefExpr 0xd224f8 <col:3> 'int[3]' lvalue Var 0xd21ad0 'a' 'int[3]'

  表示对引用的声明。

- | | '-IntegerLiteral 0xd22518 <col:5> 'int' 2

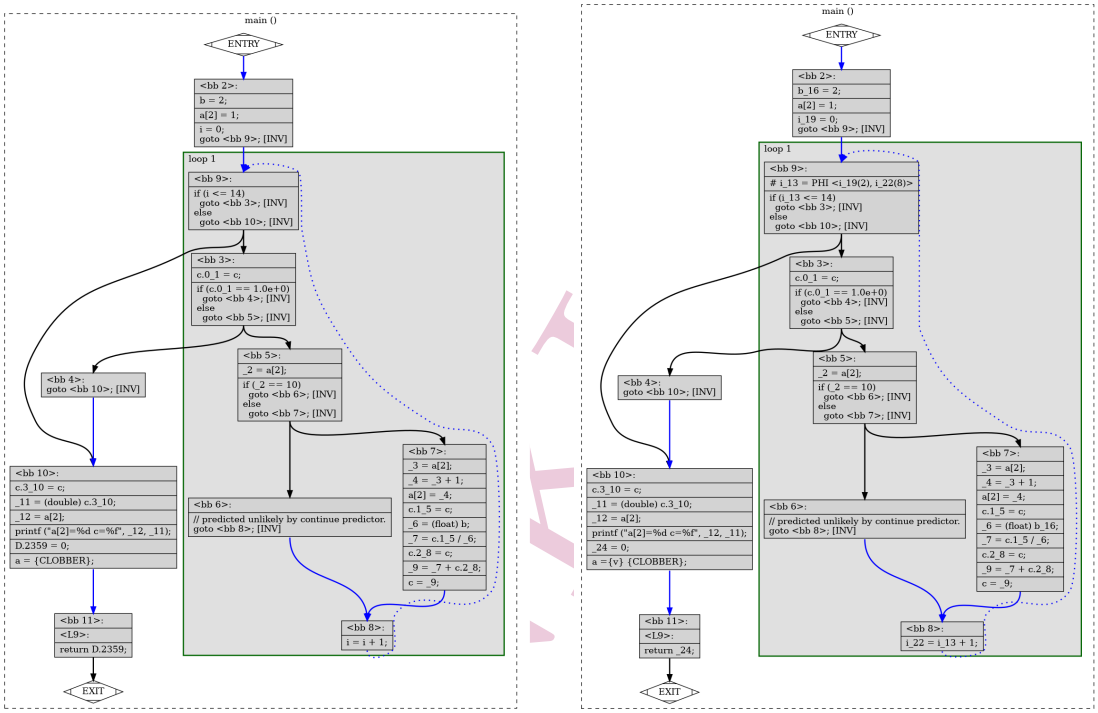
  整数字面量 2，表示数组下标。

c) 语义分析

语义分析会使用语法树和符号表中信息进行一致性检查和越界检查，完成语句的翻译。其和语法分析的区别主要在于：语法分析仅能判断句子是否合法，而语义分析可以进一步给出句子的含义。

d) 中间代码生成

使用 $gcc -fdump-tree-all-graph$ 6.c 获得中间代码生成的输出，选择 $a-6.c.015t.cfg.dot$ 与 $a-6.c.027t.fixup\_cfg2.dot$（该图与最后一次生成的 dot 图相同）通过 $graphviz$ 绘制 CFG 图，如图3所示。



(a) 稍早一些的 CFG 图 (015t)　　(b) 稍晚一些的 CFG 图 (027t)

图 3

通过 $clang -S -emit-llvm 6.c$ 生成 $LLVM$ 中间代码 $6.ll$，主要内容如下。

6.ll

```
//源文件和目标布局、平台等声明
; ModuleID = '6.c'
source_filename = "6.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"
//全局和私有变量（用于格式化字符串）定义
@c = dso_local global float 0x400F0A3D80000000, align 4
@.str = private unnamed_addr constant [13 x i8] c"a[2]=%d␣c=%f\00", align 1
//主函数，preds表示前驱结点
; Function Attrs: noinline nounwind optnone uwtable
```

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca [3 x i32], align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 2, i32* %3, align 4
  %5 = getelementptr inbounds [3 x i32], [3 x i32]* %2, i64 0, i64 2
  store i32 1, i32* %5, align 4
  store i32 0, i32* %4, align 4
  br label %6

6:                                                ; preds = %28, %0
  %7 = load i32, i32* %4, align 4
  %8 = icmp slt i32 %7, 15
  br i1 %8, label %9, label %31

9:                                                ; preds = %6
  %10 = load float, float* @c, align 4
  %11 = fcmp oeq float %10, 1.000000e+00
  br i1 %11, label %12, label %13

12:                                               ; preds = %9
  br label %31

13:                                               ; preds = %9
  %14 = getelementptr inbounds [3 x i32], [3 x i32]* %2, i64 0, i64 2
  %15 = load i32, i32* %14, align 4
  %16 = icmp eq i32 %15, 10
  br i1 %16, label %17, label %18

17:                                               ; preds = %13
  br label %28

18:                                               ; preds = %13
  %19 = getelementptr inbounds [3 x i32], [3 x i32]* %2, i64 0, i64 2
  %20 = load i32, i32* %19, align 4
  %21 = add nsw i32 %20, 1
  store i32 %21, i32* %19, align 4
  %22 = load float, float* @c, align 4
  %23 = load i32, i32* %3, align 4
  %24 = sitofp i32 %23 to float
  %25 = fdiv float %22, %24
  %26 = load float, float* @c, align 4
  %27 = fadd float %26, %25
  store float %27, float* @c, align 4
  br label %28

```

```
28:                                               ; preds = %18, %17
  %29 = load i32, i32* %4, align 4
  %30 = add nsw i32 %29, 1
  store i32 %30, i32* %4, align 4
  br label %6, !llvm.loop !6

31:                                               ; preds = %12, %6
  %32 = getelementptr inbounds [3 x i32], [3 x i32]* %2, i64 0, i64 2
  %33 = load i32, i32* %32, align 4
  %34 = load float, float* @c, align 4
  %35 = fpext float %34 to double
  %36 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([13 x
       i8], [13 x i8]* @.str, i64 0, i64 0), i32 noundef %33, double noundef
      %35)
  ret i32 0
}
//printf声明
declare i32 @printf(i8* noundef, ...) #1
//函数或全局变量的行为或优化选项
attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "
    min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-
    buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx
    ,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-
    protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8
    ,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
//元信息，一般不影响运行，可能用于优化或调试
!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
!6 = distinct !{!6, !7}
!7 = !{!"llvm.loop.mustprogress"}
```

中间代码生成指的是在之前步骤完成后，为了提高编译器的可移植性和鲁棒性而进行的一步翻译操作。中间代码生成包括有向无环图，三地址码（三元式、四元式）等形式。

e) 代码优化

代码优化指的是在生成机器码之前，对中间代码进行进一步的改进，以便生成更好的目标代码。

LLVM 提供了三种类型 (Analysis Passes、Transform Passes 和 UtilityPasses) 的众多 *pass* 对代码进行优化。

通过 $llc\ -print-before-all\ -print-after-all\ 6.ll\ >\ 6.log\ 2>\&1$ 可以完成代码优化全过程的输出，图4展示了代码优化最后一轮的部分中间代码。

```
19289 # *** IR Dump After Pseudo Probe Inserter (pseudo-probe-inserter) ***:
19290 # Machine code for function main: NoPHIs, TracksLiveness, NoVRegs, TiedOpsRewritten, TracksDebugUserValues
19291 Frame Objects:
19292   fi#-1: size=8, align=16, fixed, at location [SP-8]
19293   fi#0: size=4, align=4, at location [SP-32]
19294   fi#1: size=12, align=4, at location [SP-28]
19295   fi#2: size=4, align=4, at location [SP-16]
19296   fi#3: size=4, align=4, at location [SP-12]
19297 Constant Pool:
19298   cp#0: 1.000000e+00, align=4
19299
19300 bb.0 (%ir-block.0):
19301   successors: %bb.1
19302
19303   frame-setup PUSH64r killed $rbp, implicit-def $rsp, implicit $rsp
19304   CFI_INSTRUCTION def_cfa_offset 16
19305   CFI_INSTRUCTION offset $rbp, -16
19306   $rbp = frame-setup MOV64rr $rsp
19307   CFI_INSTRUCTION def_cfa_register $rbp
19308   $rsp = frame-setup SUB64ri8 $rsp(tied-def 0), 32, implicit-def dead $eflags
19309   MOV32mi $rbp, 1, $noreg, -24, $noreg, 0 :: (store (s32) into %ir.1)
19310   MOV32mi $rbp, 1, $noreg, -8, $noreg, 2 :: (store (s32) into %ir.3)
19311   MOV32mi $rbp, 1, $noreg, -12, $noreg, 1 :: (store (s32) into %ir.5)
19312   MOV32mi $rbp, 1, $noreg, -4, $noreg, 0 :: (store (s32) into %ir.4)
19313
19314 bb.1 (%ir-block.6):
19315 ; predecessors: %bb.0, %bb.7
19316   successors: %bb.8, %bb.2
19317
19318   CMP32mi8 $rbp, 1, $noreg, -4, $noreg, 15, implicit-def $eflags :: (load (s32) from %ir.4)
19319   JCC_1 %bb.8, 13, implicit killed $eflags
19320
19321 bb.2 (%ir-block.9):
19322 ; predecessors: %bb.1
19323   successors: %bb.4, %bb.3
19324
19325   renamable $xmm0 = MOVSSrm_alt $noreg, 1, $noreg, @c, $noreg :: (load (s32) from @c)
19326   renamable $xmm1 = MOVSSrm_alt $rip, 1, $noreg, %const.0, $noreg
19327   UCOMISSrr killed renamable $xmm0, killed renamable $xmm1, implicit-def $eflags, implicit $mxcsr
19328   JCC_1 %bb.4, 5, implicit $eflags
```

图 4: LLVM 代码优化

通过 $llvm-as$ 和 $llvm-dis$ 可以完成 ll 和 bc 两种格式的转换。通过 opt 指令可以查看某一步的优化结果，比如，通过 $opt-print-callgraph 6.bc$ 可以查看函数调用关系。

6.bc 函数调用图

```
Call graph node <<null function>><<0xf71fa0>>  #uses=0
  CS<None> calls function 'main'
  CS<None> calls function 'printf'

Call graph node for function: 'main'<<0xf72cd0>>  #uses=1
  CS<0xf6dee0> calls function 'printf'

Call graph node for function: 'printf'<<0xf72d50>>  #uses=2
  CS<None> calls external node
```

从图中可以看出，根节点 null 调用了主函数 main 和 printf 函数，主函数 main 被调用一次，并调用了 printf 函数，printf 函数被调用 2 次，并调用了外部结点。这与我们设想的结果是相符的。

f) 代码生成

代码生成将中间表示形式映射到最终需要的目标语言的工作。

最终生成的 LLVM 目标代码如下。

6.s

```
        .text
        .file   "6.c"
```

```
        .section        .rodata.cst4,"aM",@progbits,4
        .p2align        2                               # -- Begin function main
.LCPI0_0:
        .long   0x3f800000                      # float 1
        .text
        .globl  main
        .p2align        4, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq    $32, %rsp
        movl    $0, -24(%rbp)
        movl    $2, -8(%rbp)
        movl    $1, -12(%rbp)
        movl    $0, -4(%rbp)
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        cmpl    $15, -4(%rbp)
        jge     .LBB0_8
# %bb.2:                                #   in Loop: Header=BB0_1 Depth=1
        movss   c, %xmm0                        # xmm0 = mem[0],zero,zero,zero
        movss   .LCPI0_0(%rip), %xmm1           # xmm1 = mem[0],zero,zero,zero
        ucomiss %xmm1, %xmm0
        jne     .LBB0_4
        jp      .LBB0_4
# %bb.3:
        jmp     .LBB0_9
.LBB0_4:                                #   in Loop: Header=BB0_1 Depth=1
        cmpl    $10, -12(%rbp)
        jne     .LBB0_6
# %bb.5:                                #   in Loop: Header=BB0_1 Depth=1
        jmp     .LBB0_7
.LBB0_6:                                #   in Loop: Header=BB0_1 Depth=1
        movl    -12(%rbp), %eax
        addl    $1, %eax
        movl    %eax, -12(%rbp)
        movss   c, %xmm0                        # xmm0 = mem[0],zero,zero,zero
        cvtsi2ssl       -8(%rbp), %xmm1
        divss   %xmm1, %xmm0
```

```
47          addss   c, %xmm0
48          movss   %xmm0, c
49  .LBB0_7:                                # in Loop: Header=BB0_1 Depth=1
50          movl    -4(%rbp), %eax
51          addl    $1, %eax
52          movl    %eax, -4(%rbp)
53          jmp     .LBB0_1
54  .LBB0_8:                                # %.loopexit
55          jmp     .LBB0_9
56  .LBB0_9:
57          movl    -12(%rbp), %esi
58          movss   c, %xmm0                        # xmm0 = mem[0],zero,zero,
                zero
59          cvtss2sd        %xmm0, %xmm0
60          movabsq $.L.str, %rdi
61          movb    $1, %al
62          callq   printf@PLT
63          xorl    %eax, %eax
64          addq    $32, %rsp
65          popq    %rbp
66          .cfi_def_cfa %rsp, 8
67          retq
68  .Lfunc_end0:
69          .size   main, .Lfunc_end0-main
70          .cfi_endproc
71                                          # -- End function
72          .type   c,@object                       # @c
73          .data
74          .globl  c
75          .p2align        2
76  c:
77          .long   0x407851ec                      # float 3.88000011
78          .size   c, 4
79
80          .type   .L.str,@object                  # @.str
81          .section        .rodata.str1.1,"aMS",@progbits,1
82  .L.str:
83          .asciz  "a[2]=%d␣c=%f"
84          .size   .L.str, 13
85
86          .ident  "Ubuntu␣clang␣version␣14.0.0-1ubuntu1.1"
87          .section        ".note.GNU-stack","",@progbits
```

（五） 汇编器

通过 $llc\ 6.bc\ -filetype=obj\ -o\ 6.o$ 完成汇编过程，并通过 $objdump\ -d\ 6.o$ 完成反汇编。机器码，反汇编结果以及个人对汇编语言的结果分析如下。

反汇编结果

```
6.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp //保存栈底指针
   1:   48 89 e5                mov    %rsp,%rbp//栈顶->栈底
   4:   48 83 ec 20             sub    $0x20,%rsp//分配32字节内存空间，用于存储函数局部变量
   8:   c7 45 e8 00 00 00 00    movl   $0x0,-0x18(%rbp)//初始化a[3]，作用为把a[0]初始化为0
   f:   c7 45 f8 02 00 00 00    movl   $0x2,-0x8(%rbp)//b=2
  16:   c7 45 f4 01 00 00 00    movl   $0x1,-0xc(%rbp)//a[2]=1
  1d:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)//i=0
  24:   83 7d fc 0f             cmpl   $0xf,-0x4(%rbp)//比较i和15
  28:   7d 5a                   jge    84 <main+0x84>//若大于，循环结束
  2a:   f3 0f 10 04 25 00 00    movss  0x0,%xmm0//加载c
  31:   00 00
  33:   f3 0f 10 0d 00 00 00    movss  0x0(%rip),%xmm1        # 3b <main+0x3b>//加载float 1
  3a:   00
  3b:   0f 2e c1                ucomiss %xmm1,%xmm0//比较c和1的大小
  3e:   75 04                   jne    44 <main+0x44>//若不相等，继续执行
  40:   7a 02                   jp     44 <main+0x44>//若溢出，继续执行
  42:   eb 42                   jmp    86 <main+0x86>//若相等，终止循环break
  44:   83 7d f4 0a             cmpl   $0xa,-0xc(%rbp)//比较a[2]和10
  48:   75 02                   jne    4c <main+0x4c>//若不等，继续执行
  4a:   eb 2d                   jmp    79 <main+0x79>//否则跳过后续指令开始下一轮循环
  4c:   8b 45 f4                mov    -0xc(%rbp),%eax//加载a[2]到eax
  4f:   83 c0 01                add    $0x1,%eax//更新a[2]
  52:   89 45 f4                mov    %eax,-0xc(%rbp)//写回a[2]
  55:   f3 0f 10 04 25 00 00    movss  0x0,%xmm0//读取c
  5c:   00 00
  5e:   f3 0f 2a 4d f8          cvtsi2ssl -0x8(%rbp),%xmm1//把b转换为浮点数存入寄存器
  63:   f3 0f 5e c1             divss  %xmm1,%xmm0//c/b
  67:   f3 0f 58 04 25 00 00    addss  0x0,%xmm0//c+=c/b
  6e:   00 00
  70:   f3 0f 11 04 25 00 00    movss  %xmm0,0x0//写回c到内存
  77:   00 00
  79:   8b 45 fc                mov    -0x4(%rbp),%eax//读取i
  7c:   83 c0 01                add    $0x1,%eax//i+1
  7f:   89 45 fc                mov    %eax,-0x4(%rbp)//写回i
  82:   eb a0                   jmp    24 <main+0x24>//返回循环检查
  84:   eb 00                   jmp    86 <main+0x86>//循环结束
```

```
86:   8b 75 f4                mov    -0xc(%rbp),%esi //加载a[2]准备打印
89:   f3 0f 10 04 25 00 00    movss  0x0,%xmm0//加载c
90:   00 00
92:   f3 0f 5a c0             cvtss2sd %xmm0,%xmm0//调整c为双精度
96:   48 bf 00 00 00 00 00    movabs $0x0,%rdi //传递格式化字符串
9d:   00 00 00
a0:   b0 01                   mov    $0x1,%al
a2:   e8 00 00 00 00          call   a7 <main+0xa7> //调用printf
a7:   31 c0                   xor    %eax,%eax //清空eax, 设置return 为0
a9:   48 83 c4 20             add    $0x20,%rsp//释放局部变量
ad:   5d                      pop    %rbp//恢复栈帧
ae:   c3                      ret
```

可以看到，汇编器完成了指令转换，将汇编码转换为机器码。此外，汇编器还需要对定义的各种变量进行解析，并为其相应的分配内存，同时做好寄存器和栈帧的设置。除此之外，汇编器还可以实现对程序进行简单的优化。

（六） 链接器

通过 $gcc\ 6.o\ -o\ 6\ -no-pie$ 生成可执行程序。

执行结果如图5所示，a[2]=10,c=149.160248 与预期结果相符，完成验证。

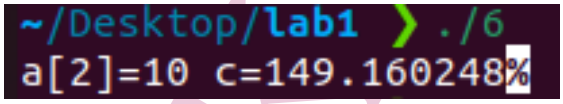


图 5: 6.exe 运行结果

```
0000000004010f0 <__do_global_dtors_aux>:
  4010f0:       f3 0f 1e fa             endbr64
  4010f4:       80 3d 39 2f 00 00 00    cmpb   $0x0,0x2f39(%rip)        # 404034 <completed.0>
  4010fb:       75 13                   jne    401110 <__do_global_dtors_aux+0x20>
  4010fd:       55                      push   %rbp
  4010fe:       48 89 e5                mov    %rsp,%rbp
  401101:       e8 7a ff ff ff          call   401080 <deregister_tm_clones>
  401106:       c6 05 27 2f 00 00 01    movb   $0x1,0x2f27(%rip)        # 404034 <completed.0>
  40110d:       5d                      pop    %rbp
  40110e:       c3                      ret
  40110f:       90                      nop
  401110:       c3                      ret
  401111:       66 66 2e 0f 1f 84 00    data16 cs nopw 0x0(%rax,%rax,1)
  401118:       00 00 00 00
  40111c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000401120 <frame_dummy>:
  401120:       f3 0f 1e fa             endbr64
  401124:       eb 8a                   jmp    4010b0 <register_tm_clones>
  401126:       66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
  40112d:       00 00 00

0000000000401130 <main>:
  401130:       55                      push   %rbp
  401131:       48 89 e5                mov    %rsp,%rbp
  401134:       48 83 ec 20             sub    $0x20,%rsp
  401138:       c7 45 e8 00 00 00 00    movl   $0x0,-0x18(%rbp)
  40113f:       c7 45 f8 02 00 00 00    movl   $0x2,-0x8(%rbp)
  401146:       c7 45 f4 01 00 00 00    movl   $0x1,-0xc(%rbp)
  40114d:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  401154:       83 7d fc 0f             cmpl   $0xf,-0x4(%rbp)
  401158:       7d 5a                   jge    4011b4 <main+0x84>
  40115a:       f3 0f 10 04 25 30 40    movss  0x404030,%xmm0
  401161:       40 00
  401163:       f3 0f 10 0d 99 0e 00    movss  0xe99(%rip),%xmm1        # 402004 <_IO_stdin_used+0x4>
  40116a:       00
  40116b:       0f 2e c1                ucomiss %xmm1,%xmm0
  40116e:       75 04                   jne    401174 <main+0x44>
```

图 6: 6.exe 反汇编结果

如图6所示，对 6.*exe* 反汇编的结果相比于先前的多了很多库文件的机器码，此外函数的初始地址也有所调整。这说明了链接器的作用是把机器代码和目标文件库文件等链接在一起，生成可执行文件。

二、 LLVM IR 编程验证 SysY 语言特性

LLVMIR 编程由聂嘉欣同学负责，在该部分中针对函数，各类型语句（赋值语句、表达式语句、语句块、if、while、break、continue、return 语句等）以及常量变量声明和数组等方面通过 *LLVMIR* 编程学习和验证了 SysY 语言特性。

(一) 函数

SysY 的语法结构与 c 语言基本一致。在 LLVM IR 函数定义时我们做如下结构的函数声明:

LLVM IR 函数定义框架

```
define <type> @<function name>()
{
    ; 函数主体
}
```

以下给出一个简单的以 SysY 语言实现的 add 函数示例，并用 LLVM IR 语言重新编写，再用 LLVM/Clang 编译成目标程序，执行验证其正确性。

SysY 实现的 add 函数

```
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    int sum = add(3, 5);
    printf("The sum is: %d\n", sum);
    return 0;
}
```

LLVM IR 实现的 add 函数及其调用

```
declare i32 @printf(i8*, ...)
@.str = private unnamed_addr constant [16 x i8] c"The sum is: %d\0A\00",
    align 1
define i32 @add(i32 %0, i32 %1) #0 {
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 %0, i32* %3, align 4
    store i32 %1, i32* %4, align 4
    %5 = load i32, i32* %3, align 4
    %6 = load i32, i32* %4, align 4
```

```
    %7 = add nsw i32 %5, %6
    ret i32 %7
}

define i32 @main() #0 {

    %1 = alloca i32, align 4
    %2 = call i32 @add(i32 3, i32 5)
    store i32 %2, i32* %1, align 4
    %3 = load i32, i32* %1, align 4
    %4 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([16
        x i8], [16 x i8]* @.str, i64 0, i64 0), i32 noundef %3)
    ret i32 0
}

attributes #0 = { noinline nounwind optnone uwtable }
```

LLVM 中定义的函数可以使用 call 语句调用，例如调用 add 函数 call i32 @add(i32 1, i32 2)。我们编写主函数调用 add 函数，使用 llc -filetype=obj t1.ll -o t1.o 命令将 t1.ll 汇编成目标文件，然后使用 clang t1.o -o t1 命令生成可执行文件 t1 并运行，结果如图所示，结果正确，验证成功。

```
~/lab 1 njx/Q2 ❯ ./t1
The sum is: 8
~/lab 1 njx/Q2 ❯
```

图 7: 运行 t1 的结果

(二) 语句

SysY 支持的语句包括赋值语句、表达式语句 (表达式可以为空)、语句块、if 语句、while 语句、break 语句、continue 语句、return 语句等等。我们尝试用 LLVM IR 语言自行实现 SysY 语言编写的程序，以熟悉 SysY 语句在 LLVM IR 中的实现形式。

SysY 原程序

```
#include <stdio.h>
int main() {
    int a = 7;
    int b = 0;
    int i = 0;

    while (i < 10) {
        if (i == a) {
            break; // 当i等于a时跳出循环
        }

        if (i == b) {
            i = i + 1; // 在跳过当前迭代之前，先自增i
            continue; // 当i等于b时跳过当前迭代
```

```
        }

        b = b + i; // 更新b的值为b加上当前的i
        i = i + 1; // i自增
    }

    printf("The final value of b is: %d\n", b);

    return 0;
}
```

LLVM IR 的实现

```
@.str = private unnamed_addr constant [29 x i8] c"The final value of b is: %d\0A\00", align 1

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
  store i32 0, i32* %1, align 4
  store i32 7, i32* %2, align 4
  store i32 0, i32* %3, align 4
  store i32 0, i32* %4, align 4
  br label %5

5:                                                ; preds = %20, %17, %0
  %6 = load i32, i32* %4, align 4
  %7 = icmp slt i32 %6, 10
  br i1 %7, label %8, label %26

8:                                                ; preds = %5
  %9 = load i32, i32* %4, align 4
  %10 = load i32, i32* %2, align 4
  %11 = icmp eq i32 %9, %10
  br i1 %11, label %12, label %13

12:                                               ; preds = %8
  br label %26

13:                                               ; preds = %8
  %14 = load i32, i32* %4, align 4
  %15 = load i32, i32* %3, align 4
  %16 = icmp eq i32 %14, %15
  br i1 %16, label %17, label %20

17:                                               ; preds = %13
  %18 = load i32, i32* %4, align 4
```

```
  %19 = add nsw i32 %18, 1
  store i32 %19, i32* %4, align 4
  br label %5, !llvm.loop !6

20:                                               ; preds = %13
  %21 = load i32, i32* %3, align 4
  %22 = load i32, i32* %4, align 4
  %23 = add nsw i32 %21, %22
  store i32 %23, i32* %3, align 4
  %24 = load i32, i32* %4, align 4
  %25 = add nsw i32 %24, 1
  store i32 %25, i32* %4, align 4
  br label %5, !llvm.loop !6

26:                                               ; preds = %12, %5
  %27 = load i32, i32* %3, align 4
  %28 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([29 x
       i8], [29 x i8]* @.str, i64 0, i64 0), i32 noundef %27)
  ret i32 0
}

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

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "
    min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-
    buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx
    ,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-
    protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8
    ,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
!6 = distinct !{!6, !7}
!7 = !{!"llvm.loop.mustprogress"}
```

使用 llc -filetype=obj t2.ll -o t2.o 命令将 t2.ll 汇编成目标文件，然后使用 clang t2.o -o t2 命令生成可执行文件 t2 并运行，结果如图所示，结果正确，验证成功。

```
~/lab 1 njx/Q2 ❯ ./t2
The final value of b is: 18
```

图 8: 运行 t2 的结果

(三) 常量与变量的声明和数组

编写 LLVM IR 探索了 SysY 编译器中常量与变量的声明、表达式和数组的语言特性的过程。以如下代码为例:

SysY 原程序

```c
#include <stdio.h>
int a;
const int b = 4;

int main() {
    int c[3];
    int m = 2, n = 3;
    int i = m * n;
    if (m != n) {
        int temp = (m && n) + b;
        printf("out:%d\n", temp);
    }
    return 0; }
```

LLVM IR 的实现

```
@a = dso_local global i32 0, align 4
@.str = private unnamed_addr constant [8 x i8] c"out:%d\0A\00", align 1

define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca [3 x 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 2, i32* %3, align 4
  store i32 3, i32* %4, align 4

  %7 = load i32, i32* %3, align 4
  %8 = load i32, i32* %4, align 4
  %9 = mul nsw i32 %7, %8
  store i32 %9, i32* %5, align 4

  %10 = load i32, i32* %3, align 4
  %11 = load i32, i32* %4, align 4
```

```
  %12 = icmp ne i32 %10, %11
  br i1 %12, label %13, label %25

13:                                               ; preds = %0
  %14 = load i32, i32* %3, align 4
  %15 = icmp ne i32 %14, 0
  br i1 %15, label %16, label %19

16:                                               ; preds = %13
  %17 = load i32, i32* %4, align 4
  %18 = icmp ne i32 %17, 0
  br label %19

19:                                               ; preds = %16, %13
  %20 = phi i1 [false, %13], [%18, %16]
  %21 = zext i1 %20 to i32
  %22 = add nsw i32 %21, 4
  store i32 %22, i32* %6, align 4

  %23 = load i32, i32* %6, align 4
  %24 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8
      x i8]* @.str, i64 0, i64 0), i32 %23)
  br label %25

25:                                               ; preds = %19, %0
  ret i32 0
}

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

attributes #0 = { noinline nounwind optnone uwtable }
attributes #1 = { "no-trapping-math"="true" "stack-protector-buffer-size"="8"
    "target-cpu"="x86-64" }
```

使用 llc -filetype=obj t3.ll -o t3.o 命令将 t3.ll 汇编成目标文件，然后使用 clang t3.o -o t3 命令生成可执行文件 t3 并运行，结果如图所示，结果正确，验证成功。

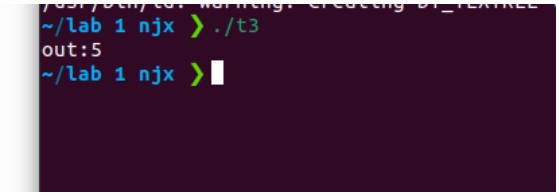


图 9: 运行 t3 的结果

三、 RISC-V 汇编程序设计

RISC − V 汇编程序设计由本人负责，在该部分中通过设计一个循环求解圆周率 π 的程序和一个冒泡排序程序对 *scanf/printf* 函数，*for* 循环，加、减、乘、除、取模、自增等基本运算，*float* 浮点数的应用以及全局变量，常量，局部变量常量的定义与使用方法，数组变量的定义与访问方法以及 *continue* 语句等进行了汇编语言编写的探讨。

(一) 程序一

1. SysY 源程序设计

程序一的设计主要功能为通过循环求解圆周率 π 的近似值。

input : 一个整数 t，表示计算的项数，t 越大，结果越接近 π。

output : π 的近似结果，保留小数点后 5 位，以换行结尾。

在这个程序中，主要涉及到以下问题：*scanf/printf* 函数，*for* 循环（含比较运算），加、减、乘、除、取模、自增等基本运算，*float* 浮点数的应用（尤其是浮点数常量的设计），全局变量和局部变量的定义和内存分配。

SysY 源程序如下：

3.1-c

```c
#include<stdio.h>
float pi=0;
int main(){
        float i=1;
        int t;
        scanf("%d",&t);
        for(int a=0;a<t;a++){
                if(a%2) pi-=1/i;
                else pi+=1/i;
                i+=2;
        }
        pi*=4;
        printf("pi=%.5f\n",pi);
        return 0;
}
```

2. RISC-V 汇编程序设计

最终编写的 RISCV 代码及个人解释如下：

3.1-s

```
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0" #架构
        .attribute unaligned_access, 0 #禁用非对齐访问
        .attribute stack_align, 16 #16字节对齐
        .text #代码区开始
        .section .data #data段（可写）
        .globl pi #全局变量pi
        .type pi,@object #pi的类型
```

```
        .size pi,4 #pi的大小
pi:
        .zero 4 #分配内存，并初始化为0
        .section .rodata #只读段
.LC0:
        .string "%d"
.LC1:
        .string "pi=%.5f\n"
.LC2:
        .word 1065353216 #1.0
.LC3:
        .word 1073741824 #2.0
.LC4:
        .word 1082130432 #4.0
        .globl main #全局函数main
        .type main,@function #类型是函数
main:
        addi sp,sp,-32 #调整栈帧
        sd ra,24(sp) #保存返回地址
        sd s0,16(sp) #保存原栈底
        addi s0,sp,32 #调整为现在的栈底
        lui a1,%hi(.LC2)
        flw fa1,%lo(.LC2)(a1) #取出1
        fsw fa1,-20(s0) #保存i=1
        lui a2,%hi(.LC0)
        addi a0,a2,%lo(.LC0) #取出scanf字符串
        addi a1,s0,-28 #调整a1，后面scanf的输出保存在a1指向的位置
        call __isoc99_scanf #调用scanf
        mv a2,zero
        sw a2,-24(s0) #a=0
.L1:
        lw a1,-28(s0)
        lw a2,-24(s0)
        blt a2,a1,.L2 #比较a，t
        j .L5
.L2:
        lw a2,-24(s0) #取出a
        andi a2,a2,1 #计算a%2
        beq a2,zero,.L3 #if分支选择
        lui a2,%hi(pi)
        flw fa2,%lo(pi)(a2) #取出pi
        lui a3,%hi(.LC2)
        flw fa3,%lo(.LC2)(a3) #取出1
        flw fa1,-20(s0) #取出i
        fdiv.s fa3,fa3,fa1 #1/i
        fsub.s fa2,fa2,fa3 #pi-=1/i
        fsw fa2,%lo(pi)(a2) #保存pi
        j .L4
```

```
.L3:
        lui a2,%hi(pi)
        flw fa2,%lo(pi)(a2)
        lui a3,%hi(.LC2)
        flw fa3,%lo(.LC2)(a3)
        flw fa1,-20(s0)
        fdiv.s fa3,fa3,fa1
        fadd.s fa2,fa2,fa3
        fsw fa2,%lo(pi)(a2) #加法分支，与上文类似
.L4:
        lui a4,%hi(.LC3)
        flw fa4,%lo(.LC3)(a4) #取出2.0
        fadd.s fa1,fa1,fa4 #i+=2
        fsw fa1,-20(s0) #保存i
        lw a1,-24(s0) #取a
        addiw a1,a1,1 #a++
        sw a1,-24(s0) #存a
        j .L1
.L5:
        lui a2,%hi(pi)
        flw fa2,%lo(pi)(a2) #取pi
        lui a4,%hi(.LC4)
        flw fa4,%lo(.LC4)(a4) #取出4.0
        fmul.s fa2,fa2,fa4 #pi*=4
        fcvt.d.s fa2,fa2 #pi的类型转换，float->double(很重要)
        fmv.x.d a1,fa2 #存入a1，用于printf
        lui a4,%hi(.LC1)
        addi a0,a4,%lo(.LC1) #取出格式化字符串
        call printf
        li a0,0 #置返回值为0
        ld ra,24(sp)
        ld s0,16(sp)
        addi sp,sp,32 #恢复栈帧
        jr ra #end
        .size    main, .-main #表明main到这里结束
```

设计中遇到过的问题和难点如下：

- 浮点型常量的定义

  在本次设计中，需要设计浮点型常量 1.0,2.0,4.0 以便计算中使用，这些结果的计算方式需要通过 IEEE 754 单精度浮点数的表示得出，其内容为：1 位符号 8 位阶码 23 位尾数，如 1.0 的转换结果为 0 10000000 00000000000000000000000 转换为 10 进制的结果就是 1065353216。2.0，4.0 同理。

- section 的声明

  虽然 section 有一些更加高级的声明，但最为基础的，需要声明可写的全局变量在 data 或其他类型可写段中，不进行声明或声明为只读段（如 rodata）则会导致段错误。

- 浮点数精度的转换

比如，代码中 81 行（将双精度浮点数存入 a1）之前的 80 行（单精度到双精度的转换）是必要的，否则会导致类型错误，从而使得 pi 计算结果错误（nan 等）。

- 调试

  相比于 c 语言的调试，汇编语言的调试更加困难，需要借助更复杂的工具或者人工设计调试方法。

3. 样例输入输出

由于本实验主要用于求解圆周率 π 的近似值，因此输入越大，结果应越接近 3.14159（由于保留五位小数的设定）。

a) 样例一

1.in & 1.out

```
//1.in
1
//1.out
pi=4.00000
```

b) 样例二

2.in & 2.out

```
//2.in
9998
//2.out
pi=3.14150
```


（二） 程序二

1. SysY 源程序设计

程序二的设计主要功能为实现一个简单的冒泡排序。

input :t+1 个整数（t<=10），第一个整数 t，表示数组的项数，后面为数组元素。

output : 冒泡排序后的数组结果。

在这个程序中，相比于上一个程序，主要涉及到以下新问题：全局数组变量的定义，*scanf*/*printf* 函数对数组数据的访问，*printf* 对单字符输出的优化，*continue* 语句的处理，循环嵌套的处理，定义新函数的处理。

3-2.c

```
#include<stdio.h>
int nums[10];
void sort(int num){
        for(int i=0;i<num;i++){
                for(int j=i+1;j<num;j++){
                        if(nums[i]<=nums[j])
```

```
                continue;
            int temp=nums[i];
            nums[i]=nums[j];
            nums[j]=temp;
        }
    }
}
int main(){
    int t;
    scanf("%d",&t);
    for(int i=0;i<t;i++)
        scanf("%d",&nums[i]);
    sort(t);
    for(int i=0;i<t;i++)
        printf("%d␣",nums[i]);
    printf("\n");
    return 0;
}
```

2. RISC-V 汇编程序设计

最终编写的 RISCV 代码及个人解释如下:

3-2.s

```
    .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
    .attribute unaligned_access, 0
    .attribute stack_align, 16
    .text
    .section .bss #未初始化数据段
    .globl  nums #全局变量数组的声明
    .type   nums, @object
    .size   nums, 40
nums:
    .zero   40
    .section .rodata #只读段
.LC0:
    .string "%d"
.LC1:
    .string "%d␣"
    .text
    .globl sort #函数sort声明
    .type sort,@function
sort:
    addi sp,sp,-32
    sd s0,24(sp)
    addi s0,sp,32 #栈帧调整
    sw a0,-20(s0) #保存参数num
    sw zero,-16(s0) #i=0
```

```
.L1:
        lw a1,-20(s0)
        lw a2,-16(s0)
        blt a2,a1,.L2 #i<num ->L2
        j .L7 #i>=num ->L7
.L2:
        addiw a2,a2,1 #j=i+1
        sw a2,-12(s0) #保存j
.L3:
        lw a1,-20(s0)
        lw a2,-12(s0)
        blt a2,a1,.L4 #j<num ->L4
        j .L6 #j>=num ->L6
.L4:
        lui a3,%hi(nums)
        addi a3,a3,%lo(nums) #读取nums数组基地址
        lw a1,-16(s0)
        slli a1,a1,2 #数组下标i左移两位表示偏移量
        add a1,a1,a3 #此时为nums[i]的位置
        lw a4,0(a1) #nums[i]
        lw a2,-12(s0)
        slli a2,a2,2
        add a2,a2,a3 #nums[j]的位置
        lw a5,0(a2) #nums[j]
        ble a4,a5,.L5 #比较大小 看是否需要互换
        sw a5,0(a1) #互换存储
        sw a4,0(a2)
.L5:
        lw a2,-12(s0) #j++
        addiw a2,a2,1
        sw a2,-12(s0)
        j .L3
.L6:
        lw a2,-16(s0) #i++
        addiw a2,a2,1
        sw a2,-16(s0)
        j .L1
.L7:
        ld s0,24(sp)
        addi sp,sp,32 #结束, 栈帧恢复
        jr ra
        .globl main
        .type main,@function
main:
        addi sp,sp,-32
        sd ra,24(sp)
        sd s0,16(sp)
        addi s0,sp,32 #栈帧调整
```

```
73          addi a1,s0,-28 #设置t保存位置为s0-28
74          lui a0,%hi(.LC0)
75          addi a0,a0,%lo(.LC0) #读取格式化字符串
76          call __isoc99_scanf
77          sw zero,-20(s0) #初始化i=0
78  .L8:
79          lw a1,-28(s0)
80          lw a2,-20(s0)
81          ble a1,a2,.L9 #在这一部分中我使用了与之前相反的写法，若不满足i<t跳转.
                L9
82          slli a2,a2,2
83          lui a3,%hi(nums)
84          addi a3,a3,%lo(nums)
85          add a1,a3,a2 #将a1改为要存储的地址
86          lui a0,%hi(.LC0)
87          addi a0,a0,%lo(.LC0) #字符串
88          call __isoc99_scanf
89          lw a2,-20(s0)
90          addiw a2,a2,1 #i++
91          sw a2,-20(s0)
92          j .L8
93  .L9:
94          lw a0,-28(s0) #将参数t传给函数sort
95          call sort #调用sort
96          sw zero,-20(s0) #初始化i=0（printf循环）
97  .L10:
98          lw a1,-28(s0)
99          lw a2,-20(s0)
100         ble a1,a2,.L11
101         slli a2,a2,2
102         lui a3,%hi(nums)
103         addi a3,a3,%lo(nums)
104         add a1,a3,a2
105         lw a1,0(a1) #上述与之前类似，注意printf要传入的a1是数组对应的数值
106         lui a0,%hi(.LC1)
107         addi a0,a0,%lo(.LC1)
108         call printf
109         lw a2,-20(s0)
110         addiw a2,a2,1 #i++
111         sw a2,-20(s0)
112         j .L10
113 .L11:
114         li a0,10 #换行的ASCII码
115         call putchar
116         li a0,0 #返回值0
117         ld ra,24(sp)
118         ld s0,16(sp)
119         addi sp,sp,32
```

```
120         jr ra
121         .size    main, .-main
```

设计中遇到过的问题和难点，及部分问题的讨论如下：

- putchar 函数调用

  查阅相关资料后，发现 putchar 函数的传参方式是将对应字符的 ASCII 码传入 a0，这与 scanf/printf 格式化字符串的方式并不相同。

- scanf/printf 参数

  时刻记得，scanf 传入的参数是要写入位置的地址，而 printf 要传入的参数是要打印的值，这一点在汇编语言的书写上更容易出错。

- addi 和 addiw 的区别

  addiw 仅处理 32 位的数字加法，结果被符号扩展到 64 位，而 addi 完成的是 64 位加法。因此，addiw 可能更广泛的应用于 32 位寄存器。

3. 样例输入输出

a) 样例一

1.in & 1.out

```
1 //1.in
2 1 426
3 //1.out
4 426
```

b) 样例二

2.in & 2.out

```
1 //2.in
2 10 1 1 2 2 5 5 -1 -1 -1 -1
3 //2.out
4 -1 -1 -1 -1 1 1 2 2 5 5
```


四、 代码链接

本实验 **GitHub** 仓库链接如下：

compilers