



南开大学  
Nankai University

南 开 大 学

计 算 机 学 院

编译系统原理实验报告

---

了解编译器及 LLVM IR 编程

---

马永田

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 10 月 1 日

以常见的编译器如 GCC、LLVM 等为研究对象，深入地探究语言处理系统的完整工作过程，了解预处理器、编译器、汇编器和链接器都对文件做了些什么，了解其实现方式及其原理，并进行总结汇报，此外和组内同学合作完成了 SysY 源码的编写和手动翻译成中间代码的工作以熟悉 LLVM IR 中间语言与 SysY 语言。

摘要

关键字：GCC LLVM 编译器 语言处理系统

目录

一、 实验目的	1
(一) 实验描述	1
二、 实验环境	1
三、 实验方案	1
(一) 实验方法	1
(二) 实验流程	2
四、 实验结果	3
(一) 预处理器	3
1. 头文件展开	3
2. 宏定义展开与条件编译	4
3. 删除代码中的注释	5
(二) 编译器	5
1. 词法分析	6
2. 语法分析与语义分析	6
3. 中间代码生成	7
4. 代码优化	9
5. 目标代码生成	10
(三) 汇编器	11
(四) 链接器/加载器	12
(五) LLVM IR 编程	14
1. 小组分工	14
2. SysY 语言特性及编写	14
3. LLVM IR 中间语言介绍	16
4. LLVM IR 中间语言编程	17
五、 实验总结	21
(一) 源码链接	21

## 一、 实验目的

### (一) 实验描述

以你熟悉的编译器，如 GCC、LLVM 等为研究对象，深入地探究语言处理系统的完整工作过程：

- 完整的编译过程都有什么？
- 预处理器做了什么？
- 编译器做了什么？
- 汇编器做了什么？
- 链接器做了什么？
- 通过编写 LLVM IR 程序，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 IR 程序小例子，用 LLVM/Clang 编译成目标程序、执行验证。

并尽可能地对其实现方式有所了解。

## 二、 实验环境

操作系统	Ubuntu 22.04.1 LTS	
平台架构	x86_64	
CPU 型号	AMD Ryzen 7 4800H	
Caches	L1d	128KiB
	L1i	128KiB
	L2	2MiB
	L3	16MiB
内存	3.8GiB	
编译器	gcc (Ubuntu 11.2.0-19ubuntu1) 11.2.0	
	Ubuntu LLVM version 14.0.0	

## 三、 实验方案

### (一) 实验方法

以一个简单的 C (C++) 源程序为例子，调整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，并以此撰写调研报告。调整关键的编译参数（如优化参数、链接选项参数），比较目标程序的大小、运行性能等。为更全面探索每一个阶段编译器进行的工作，源程序包含尽可能丰富的语言特性（如函数、全局变量、常量、各类宏、头文件...）

- 实验中细微修改程序，观察各阶段输出的变化，从而更清楚地了解编译器的工作
- 调整编译器程序的选项，例如加入调试选项、优化选项等，观察输出变化、了解编译器
- 尝试更深入的内容，例如令编译器做自动并行化，观察输出变化、了解编译器

实验中采用如下的程序样例进行实验：

样例代码

```

1 #include<stdio.h>
2 int main() {
3     //这是注释
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 }

```

## (二) 实验流程

编译器对代码的处理主要分为预处理、编译、汇编、链接几个部分，其流程如图1所示

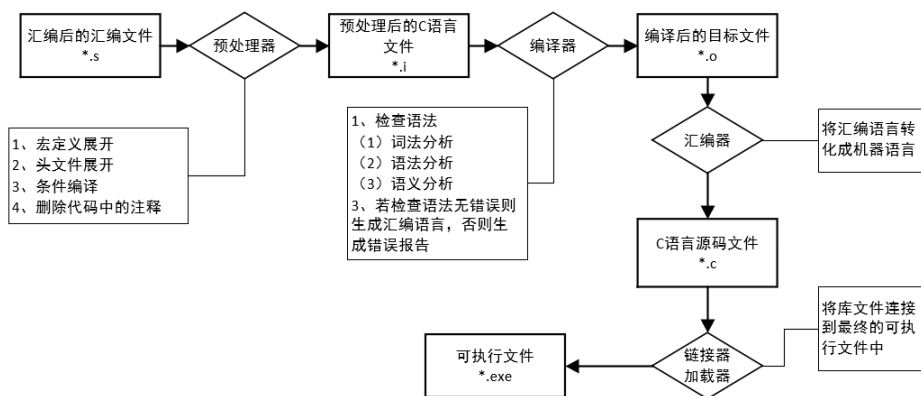


图 1: 编译流程

- 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连接在一起，形成真正能在机器上运行的目标机器代码。

## 四、 实验结果

### (一) 预处理器

编译器在预处理阶段会进行如下的操作：

- 宏定义展开
- 头文件展开
- 条件编译
- 删除代码中的注释

在该过程中编译器并不会对代码进行任何优化处理。

实验中通过对样例代码的简单调整来探究预处理阶段编译器所做的工作，使用如下命令对源代码进行预处理：

```
1 clang main.c -E -o main.i
```

首先我们将源文件中的所有宏定义、头文件等删除，得到如下的预编译结果

```
1 # 0 "main.c"
2 # 0 "<built-in>"
3 # 0 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # 0 "<command-line>" 2
6 # 1 "main.c"
7 int main() {
8     .....
9 }
```

可以发现即便我们的程序只有一个 main 函数，在预处理之后也会被添加一些其他的内容，例如上述代码中的和"main.c" 和"/usr/include/stdc-predef.h" 等。不难猜测，前者应该是源文件名，而后者则是编译器自动调用的 C 语言默认标准库；此外还有一些标志信息例如内嵌和命令行等。

#### 1. 头文件展开

对于 #include 所包含的头文件在这个阶段会被预处理器进行展开，首先我们对测试程序进行修改，添加如下内容：

```
1 #include "test.h"
2 #include <stdio.h>
3
4 //test.h:
5 void function(){
6     char test[5]="test";
7 };
```

预处理后发现文件长度相比源文件大大增加，代码足有七百多行，截取其中部分内容如下：

```

1 # 0 "main.c"
2 # 0 "<built-in>"
3 # 0 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # 0 "<command-line>" 2
6 # 1 "main.c"
7 # 1 "test.h" 1
8 void function(){
9     char test[5]="test";
10 };
11 # 2 "main.c" 2
12 # 1 "/usr/include/stdio.h" 1 3 4
13 # 27 "/usr/include/stdio.h" 3 4
14 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
15 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
16 .....
17 typedef unsigned char __u_char;
18 typedef unsigned short int __u_short;
19 .....
20 # 902 "/usr/include/stdio.h" 3 4
21 # 3 "main.c" 2
22 # 3 "main.c"
23 int main() {
24     int a, b, i, t, n;
25     .....

```

可以看到在 main 函数前增加了大量的内容，其中我们手动编写的测试头文件 test.h 被直接内嵌到了代码当中，之后则是大量的地址、定义等。查询资料后得知这应当是 stdio.h 所包含的内容，其中第一列的数字表示所需引入的代码对于路径文件的所在行，而第二列则是头文件所包含文件的地址，之后又引入相应的变量程序预定义函数等等。

其中文件名后面的数字含义为：

- “1” 表示新文件的开始
- “2” 表示返回文件（包含另一个文件后）
- “3” 表示以下文本来自系统头文件，因此应禁止某些警告
- “4” 表示以下文本应被视为包含在隐式 extern ” C” 块中

此外在实验中调整头文件的位置以及重复使用头文件后发现，处理后文件中头文件的位置与源代码中的顺序是一致的，且如果多次使用同一个头文件，处理后的文件中也会相应的对其展开多次。

## 2. 宏定义展开与条件编译

对如下代码进行预处理，观察处理后的文件以此探究宏定义展开与条件编译：

```
1 #define TEST 888
2 int main() {
3     #ifdef TEST
4     char t[5]="TEST";
5     #endif
6     int test = TEST * 10 + 8;
7     #ifndef TEST
8     char t2[10]="NoTEST"
9     #endif
10
11     #undef TEST
12     #ifdef TEST
13     char t[10]="undefTEST";
14     #endif
15 }
```

预处理后得到如下内容：

```
1 # 0 "main.c"
2 # 0 "<built-in>"
3 # 0 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # 0 "<command-line>" 2
6 # 1 "main.c"
7
8 int main() {
9
10 char t[5]="TEST";
11
12 int test = 888 * 10 + 8;
13 # 18 "main.c"
14 return 0;
15 }
```

可以看到，经过预处理后源代码中 # 后的语句均未被保留，TEST 有定义时仅有 ifdef 块内的语句被保留，ifndef 块中的语句被删除掉了，而 TEST 释放后则是 ifdef 块内的代码被删除，此外形成的文件中 TEST 宏定义释放前全部的 TEST 均被 888 替换。

### 3. 删除代码中的注释

细看示例程序的预处理结果就会发现其中的注释“//这是注释”并未被预处理器保留。

## (二) 编译器

编译部分是整个流程中十分重要的一部分，编译器会对预处理完的文件进行一系列词法分析、语法分析、语义分析及优化后，将源程序文件翻译成为标准的汇编语言以供计算机阅读。

## 1. 词法分析

词法分析是编译的第一个阶段，编译器从左至右逐个字符地对源程序进行扫描，把源代码中每个空格之间的有效符号视为一个“token”，从源代码中提取出 token 的过程就被称为词法分析。在整个词法分析的过程中形成词的方式以及所形成的词都是以字符串的形式来表现的，最终把字符串形式的源程序改造成单词符号串形式的中间程序。

为方便观察，去示例代码中的头文件 `stdio.h` 后使用如下命令进行词法分析：

```
1 clang -E -Xclang -dump-tokens main.c
```

截取部分内容后得到如下结果：

```
1 int 'int'          [StartOfLine]  Loc=<main.c:1:1>
2 identifier 'main'   [LeadingSpace] Loc=<main.c:1:5>
3 l_paren '('         Loc=<main.c:1:9>
4 r_paren ')'         Loc=<main.c:1:10>
5 l_brace '{'         [LeadingSpace] Loc=<main.c:1:12>
6 int 'int'          [StartOfLine] [LeadingSpace]  Loc=<main.c:2:2>
7 identifier 'a'      [LeadingSpace] Loc=<main.c:2:6>
8 comma ','          Loc=<main.c:2:7>
9 identifier 'b'      [LeadingSpace] Loc=<main.c:2:9>
10 comma ','          Loc=<main.c:2:10>
11 .....
12 identifier 'scanf'   [StartOfLine] [LeadingSpace]  Loc=<main.c:6:2>
13 l_paren '('         Loc=<main.c:6:7>
14 string_literal '"%d"' Loc=<main.c:6:8>
15 comma ','          Loc=<main.c:6:12>
16 amp '&' [LeadingSpace] Loc=<main.c:6:14>
17 identifier 'n'      Loc=<main.c:6:15>
18 r_paren ')'         Loc=<main.c:6:16>
19 .....
20 while 'while'       [StartOfLine] [LeadingSpace]  Loc=<main.c:9:2>
21 l_paren '('         [LeadingSpace] Loc=<main.c:9:8>
22 identifier 'i'      Loc=<main.c:9:9>
23 less '<'           [LeadingSpace] Loc=<main.c:9:11>
24 identifier 'n'      [LeadingSpace] Loc=<main.c:9:13>
```

可以看到源代码被转化成 tokens 序列输出，其中标注了单词的类型如标识符 `identifier` 等，以及单词的取值与地址。

## 2. 语法分析与语义分析

语法分析任务是在词法分析识别出单词符号的基础上，分析源程序的语法结构，即分析由这些单词如何组成各种语法成分，比如“声明”、“函数”、“语句”、“表达式”等，并分析判断程序的语法结构是否复合语法规则，将预处理器生成的 Tokens 转换为语法分析树；

编译器在生成语法分析树后，将会进行语义分析，执行类型检查和代码格式检查。语法分析的过程通常使用自顶向下或者自底向上的方式进行推导。语义分析阶段将变量与其用法关联起来，检查每个表达式是否有正确的类型，还有，将抽象的语法翻译成更简单的形式以方便生成机器语言。这个阶段负责生成大多数编译器警告以及语法分析过程的错误。最终输出 AST（抽象语法树）。



使用如下命令查看语法分析结果：

```
clang -E -Xclang -ast-dump main.c
```

截取关键部分得到如图2内容：

```

- RestrictAttr 0x12e7490 </usr/include/x86_64-linux-gnu/sys/cdefs.h:281:47> malloc
- FunctionDecl 0x12e85c0 </usr/include/stdio.h:837:1, /usr/include/x86_64-linux-gnu/sys/
cdefs.h:79:54> /usr/include/stdio.h:837:14 ctermid 'char *(char *)' extern
- NoThrowAttr 0x12e8668 </usr/include/x86_64-linux-gnu/sys/cdefs.h:79:35>
- FunctionDecl 0x12e8760 </usr/include/stdio.h:867:1, /usr/include/x86_64-linux-gnu/sys/
cdefs.h:79:54> /usr/include/stdio.h:867:13 flockfile 'void (FILE *)' extern
- NoThrowAttr 0x12e8808 </usr/include/x86_64-linux-gnu/sys/cdefs.h:79:35>
- FunctionDecl 0x12e8908 </usr/include/stdio.h:871:1, /usr/include/x86_64-linux-gnu/sys/
cdefs.h:79:54> /usr/include/stdio.h:871:12 flockfile 'int (FILE *)' extern
- NoThrowAttr 0x12e88b0 </usr/include/x86_64-linux-gnu/sys/cdefs.h:79:35>
- FunctionDecl 0x12e8870 </usr/include/stdio.h:874:1, /usr/include/x86_64-linux-gnu/sys/
cdefs.h:79:54> /usr/include/stdio.h:874:13 flockfile 'void (FILE *)' extern
- NoThrowAttr 0x12e8a18 </usr/include/x86_64-linux-gnu/sys/cdefs.h:79:35>
- FunctionDecl 0x12e8b50 </usr/include/stdio.h:885:1, col:27> col:12 __uflow 'int (FILE
*)' extern
- NoThrowAttr 0x12e8bb8 <col:21, col:26> col:27 'FILE *'
- FunctionDecl 0x12e8ea0 <line:886:1, col:35> col:12 __overflow 'int (FILE *, int)' exte
rn
- NoThrowAttr 0x12e8d08 <col:24, col:29> col:30 'FILE *'
- NoThrowAttr 0x12e8d88 <col:32> col:35 'int'
- FunctionDecl 0x12e8fa0 <main.c:2:1, line:22:1> line:2:5 main 'int ()'
- CompoundStmt 0x12ea168 <col:12, line:22:1>
- DeclStmt 0x12e92f0 <line:3:2, col:19>
- VarDecl 0x12e9058 <col:2, col:6> col:6 used a 'int'
- VarDecl 0x12e90d8 <col:2, col:9> col:9 used b 'int'
- VarDecl 0x12e9158 <col:2, col:12> col:12 used i 'int'
- VarDecl 0x12e91d8 <col:2, col:15> col:15 used t 'int'
- VarDecl 0x12e9258 <col:2, col:18> col:18 used n 'int'
- BinaryOperator 0x12e9348 <line:5:2, col:6> 'int' '='
- DeclRefExpr 0x12e9308 <col:2> 'int' lvalue Var 0x12e9058 'a' 'int'
- IntegerLiteral 0x12e9328 <col:6> 'int' 0
- BinaryOperator 0x12e93a8 <line:6:2, col:6> 'int' '='
- DeclRefExpr 0x12e9368 <col:2> 'int' lvalue Var 0x12e90d8 'b' 'int'
- IntegerLiteral 0x12e9388 <col:6> 'int' 1
- BinaryOperator 0x12e9408 <line:7:2, col:6> 'int' '='

```

图 2: 语法分析部分结果

如图2可见源代码被建成了一棵 AST 树，其中头文件过于繁杂，我们可以从 main 入口函数看起，将其作为树的根节点，可以观察到 main 函数中的一些声明、运算符等都被作为节点按照分析顺序先后构建到了 AST 的树结构中。

### 3. 中间代码生成

在-O0 优化等级下，分别使用“-fdump-tree-all-graph”和“-fdump-rtl-all-graph”两个参数获得中间代码生成的多阶段输出，生成的.dot 文件可以被 graphviz 可视化，借助 Vscod 中的插件进行绘制后得到如图3内容，由于该程序较为简单，可以看出优化前后代码流程变化不大。但仍存在一些优化的痕迹，例如加入了一些临时变量，优化循环体中的相邻指令变量依赖，使代码利于流水线运行等。

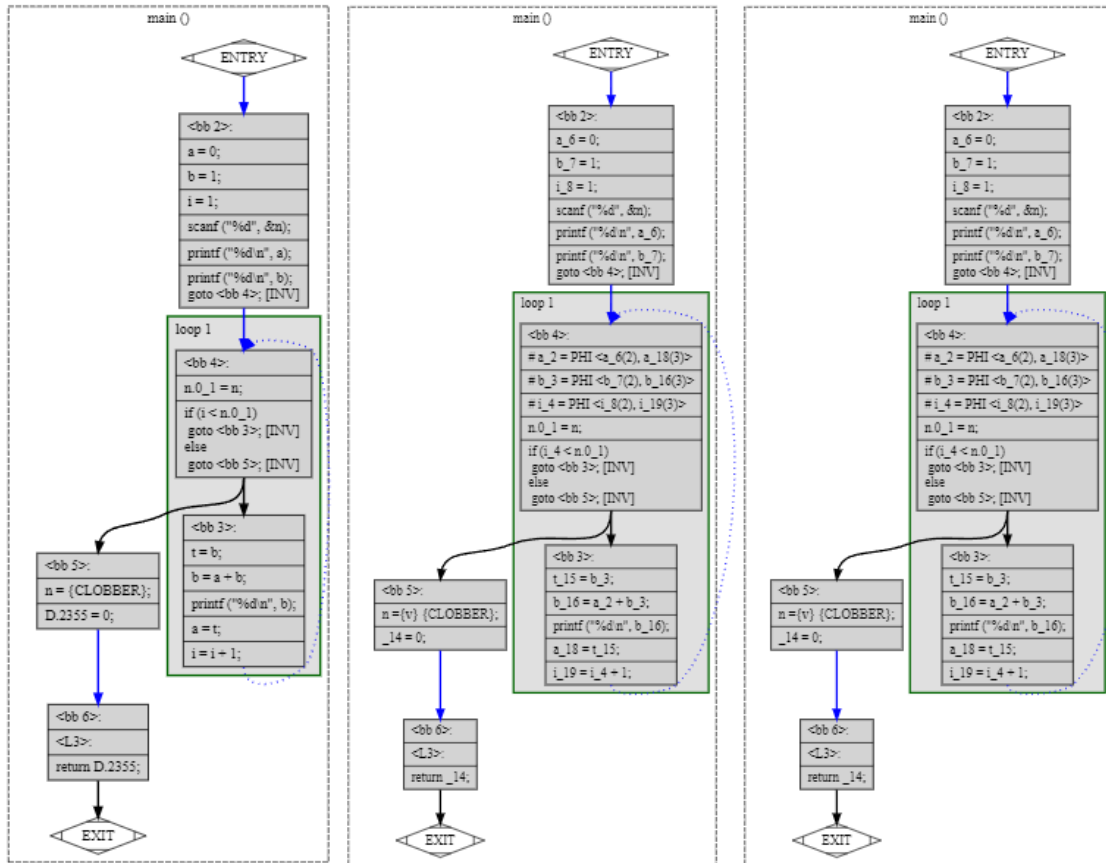


图 3: 从左至右依次为原始 CFG、SSA 后的 CFG、完全优化后的 CFG

LVM IR (Intermediate Representation) 是由代码生成器自顶向下遍历逐步翻译语法树形成的, 我们可以将任意语言的源代码编译成 LLVM IR, 然后由 LLVM 后端对 LLVM IR 进行优化并编译为相应平台的二进制程序。LLVM IR 具有 **类型化、可扩展性和强表现力**的特点。

clang 中通过如下命令生成 LLVM IR 文件

```
1 clang -S -emit-llvm main.c
```

截取关键内容得到如下代码:

```
1 ; Function Attrs: noinline nounwind optnone uwtable
2 define dso_local i32 @main() #0 {
3     %1 = alloca i32, align 4
4     %2 = alloca i32, align 4
5     %3 = alloca i32, align 4
6     %4 = alloca i32, align 4
7     %5 = alloca i32, align 4
8     %6 = alloca i32, align 4
9     store i32 0, i32* %1, align 4
10    store i32 0, i32* %2, align 4
11    store i32 1, i32* %3, align 4
12    store i32 1, i32* %4, align 4
13    %7 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds
        ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %6)
```

```

14 %8 = load i32, i32* %2, align 4
15 %9 = call i32 @i8*, ... @printf(i8* noundef getelementptr @inbounds ([4 x
    i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %8)
16 %10 = load i32, i32* %3, align 4
17 %11 = call i32 @i8*, ... @printf(i8* noundef getelementptr @inbounds ([4 x
    i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %10)
18 br label %12
19
20 12:                                ; preds = %16, %0
21 %13 = load i32, i32* %4, align 4
22 %14 = load i32, i32* %6, align 4
23 %15 = icmp slt i32 %13, %14
24 br i1 %15, label %16, label %26
25
26 16:                                ; preds = %12
27 %17 = load i32, i32* %3, align 4
28 store i32 %17, i32* %5, align 4
29 %18 = load i32, i32* %2, align 4
30 %19 = load i32, i32* %3, align 4
31 %20 = add nsw i32 %18, %19
32 store i32 %20, i32* %3, align 4
33 %21 = load i32, i32* %3, align 4
34 %22 = call i32 @i8*, ... @printf(i8* noundef getelementptr @inbounds ([4 x
    i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %21)
35 %23 = load i32, i32* %5, align 4
36 store i32 %23, i32* %2, align 4
37 %24 = load i32, i32* %4, align 4
38 %25 = add nsw i32 %24, 1
39 store i32 %25, i32* %4, align 4
40 br label %12, !llvm.loop !6
41
42 26:                                ; preds = %12
43 %27 = load i32, i32* %1, align 4
44 ret i32 %27
45 }

```

其中 @ 代表全局变量，% 代表局部变量，可以看出 IR 语言与汇编语言十分相似，也具有相对较好的可读性。

#### 4. 代码优化

编译器允许我们在编译命令中添加不同的参数来选择优化的级别并自动对程序进行优化：

- -O0 不进行任何优化，即默认选项
- -O1 优化消耗较少的编译时间，主要对代码的分支，常量以及表达式等进行优化。
- -O2 会尝试更多的寄存器级的优化以及指令级的优化，会在编译期间占用更多的内存和编译时间。

- -O3 在 O2 的基础上进行更多的优化，例如使用伪寄存器网络，普通函数的内联，以及针对循环的更多优化。

实验中选择打开-O3 优化，发现得到的 LLVM IR 语言明显更精简高效：

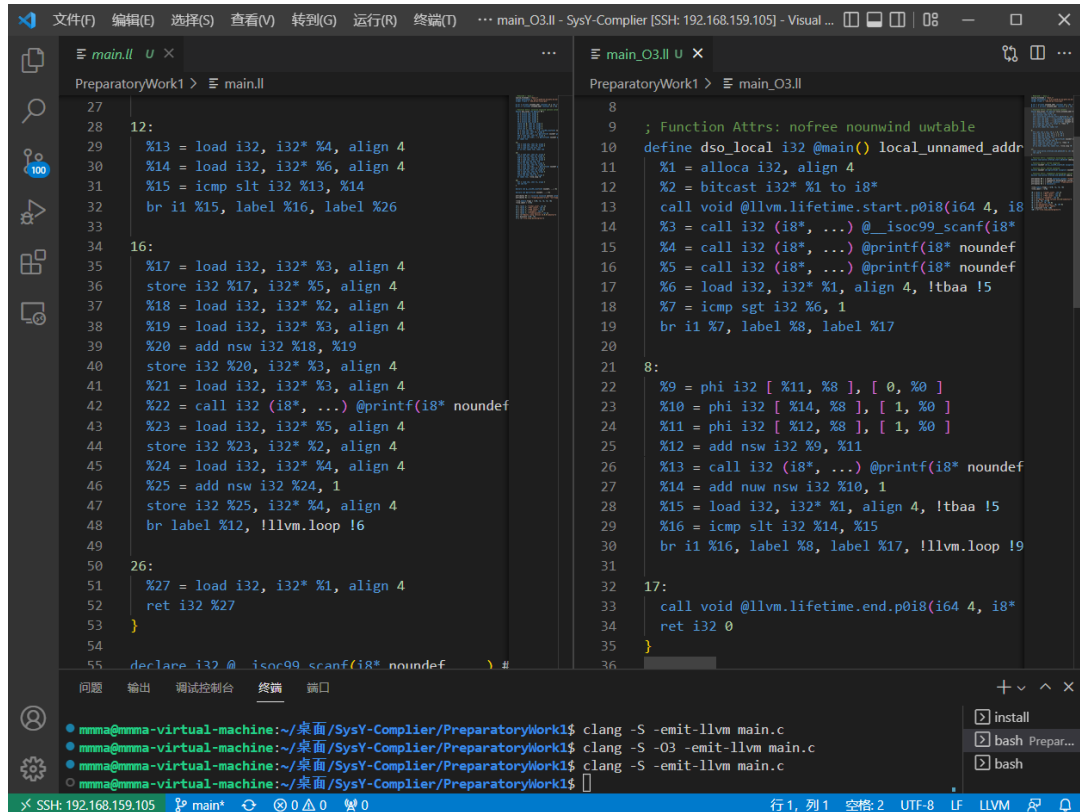


图 4: Caption

如图4可见，右边打开 O3 优化选项后得到的 IR 语言代码长度大大缩短，相比左边未优化的代码省掉了很多繁琐无用的操作。例如不再声明大量的虚拟寄存器来临时存储，而是直接使用，这样可以省略很多无用的操作；此外调整了一些指令的前后位置以及进行**循环展开**，这一点使用前文所述的样例代码进行实验时体现的并不明显，实验中又使用了一些较为容易展开的样例进行测试，例如如下的一个循环：

```
1 while( n < 10 )
2 {
3     n++;
4 }
```

打开-O3 优化后得到的 IR 语言中，n 小于 10 时并没有进入循环，而是直接将 n 赋值为 10；从图4所示代码中也可看出，打开-O3 优化后编译器对分支也进行了优化，例如使用了 phi 指令实现 phi 节点，对分支进行优化；此外编译器也会采取一些**向量化**算法进行优化等。

## 5. 目标代码生成

代码生成过程中以中间表示形式 (如前文中的 \*.i\*.ll 文件) 作为输入，将其映射到目标语言。实验中使用如下命令实现目标代码的生成：

```
1 llc main.ll -o main.S
```

如图5可见，得到了 main.S 文件，其中源代码已经被映射为汇编语言。

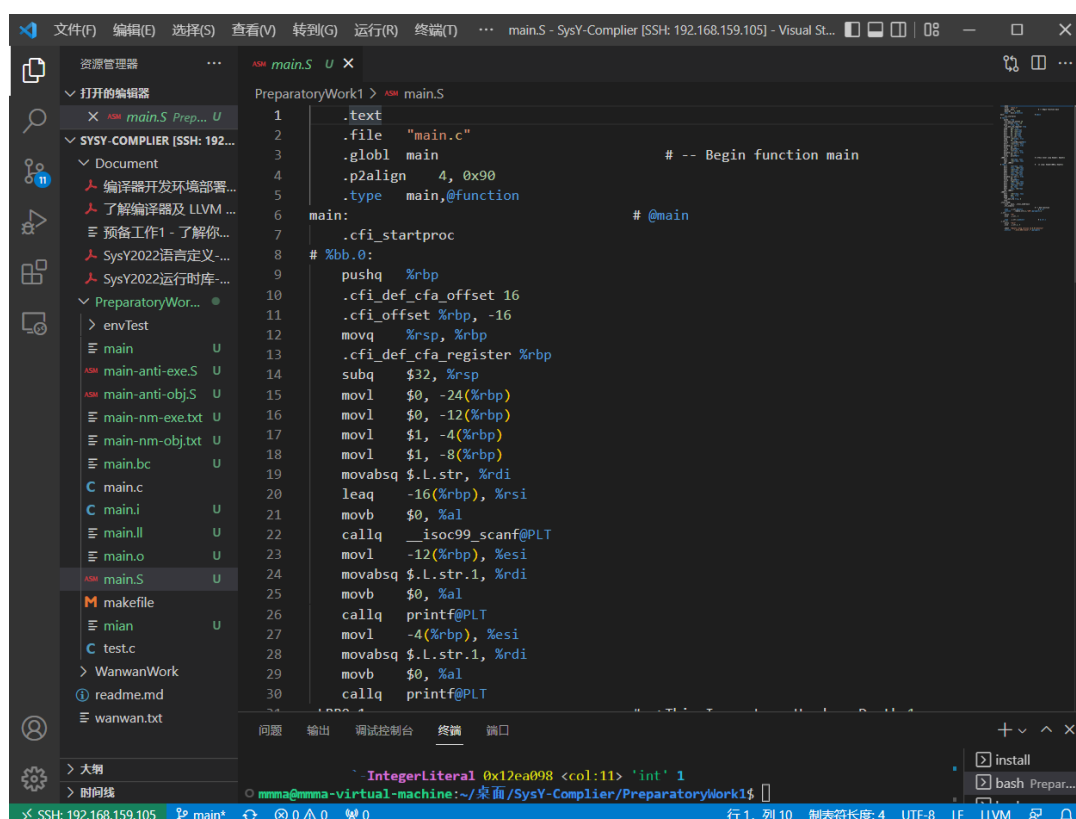


图 5: 目标文件代码

### (三) 汇编器

汇编过程调用汇编器 AS 来完成，是用于将汇编代码（即 main.S 文件）转换成机器可以执行的指令，每一个汇编语句几乎都对应一条机器指令。之后汇编器把这些指令打包成一种叫做可重定位目标程序的格式并将结果保存在新生成的二进制文件 \*.o 中。

汇编后的.o 文件是纯二进制文件。因为.o 中放的是纯二进制的机器指令。ASCII 的源码被汇编为能被 CPU 执行的机器指令，.o 文件中放的就是机器指令。但是.o 文件还无法运行，需要链接后才能运行。

使用如下命令进行汇编：

```
1 gcc -O0 -c -o main.o main.S
```

得到了一个 \*.o 文件，打开后发现是如图6所示的一堆乱码，没有任何的可读性。



图 6: 二进制文件

使用如下命令对其进行反汇编后

```
1 objdump -d main.o > main-anti-obj.S
2 nm main.o > main-nm-obj.txt
```

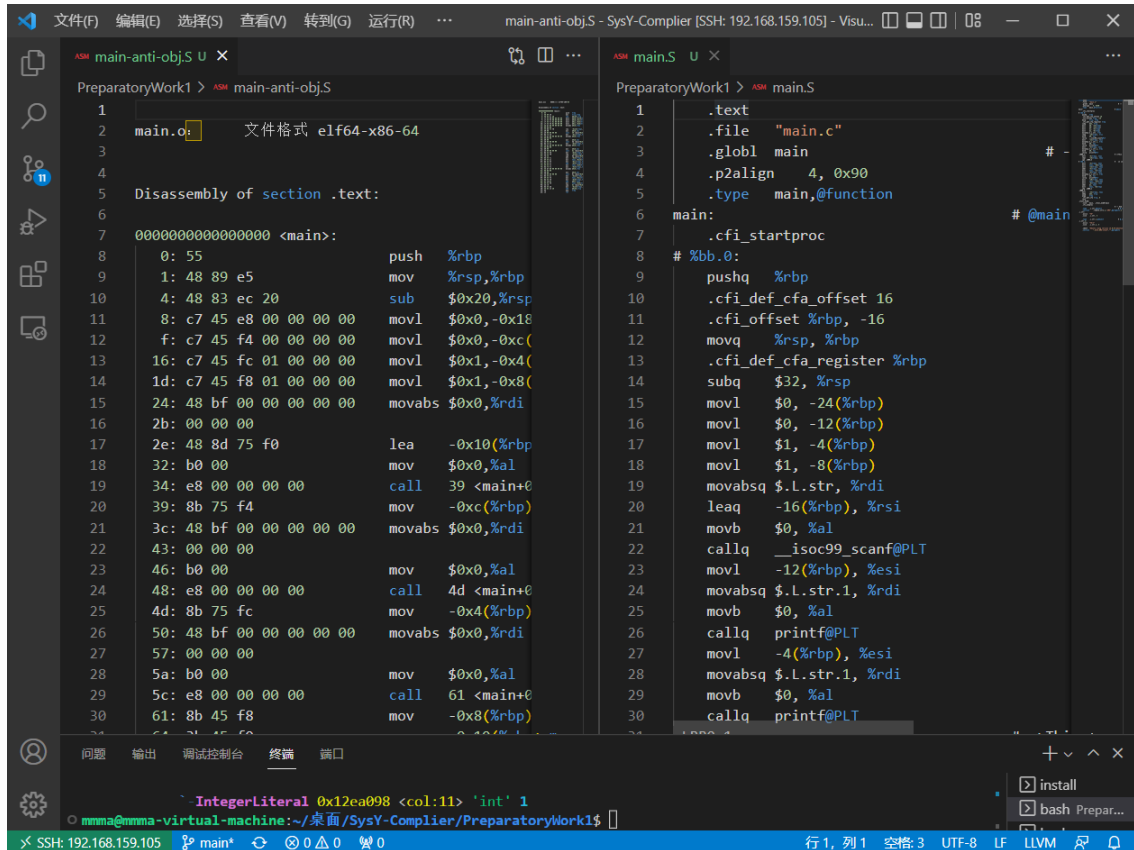


图 7: 对.o 文件反汇编

得到了如图7所示的代码片段，可以看到.o 文件反汇编后得到的内容与.S 文件中的内容十分相似，但还是有所不同的，例如.S 中的一些标志信息、伪指令、注释等，在被汇编器汇编成.o 文件时都被删除掉了，只剩下了程序所要执行的机器指令。

逐行观察两者的指令可以发现，汇编的过程中也并非是完全逐字翻译的，其中有许多指令在汇编后发生了一些变化，例如有些指令汇编后加了后缀（如 sub 和 subq），查询资料后得知是因为一些数据流通方向以及数据长度的区别，汇编过程中会根据寄存器的规格等进行相应的调整优化。

此外还可以看到，汇编后的程序中多了很多 Nop 指令，分析原因应当是为了填充指令长度使得指令按字长对齐，这样可以减少访存次数，此外也可能与流水线优化有关。

#### (四) 链接器/加载器

通过汇编得到的 main.o 文件并不是最后的可执行二进制文件。在该阶段链接器会将所有关联的可重定位目标文件组合起来，以生成一个可执行文件。

使用如下命令对可重定位文件进行处理：

```
1 gcc -O0 -o main main.o
```



得到最终的可执行文件，打开后可以看到依旧是无法阅读的乱码。使用如下命令对可执行文件进行反汇编：

```
1 objdump -d main > main-anti-exe.S
2 nm main > main-nm-exe.txt
```

最终得到如图8的内容，对比.o 文件可以发现，最终可执行文件中的 main 部分是相同的

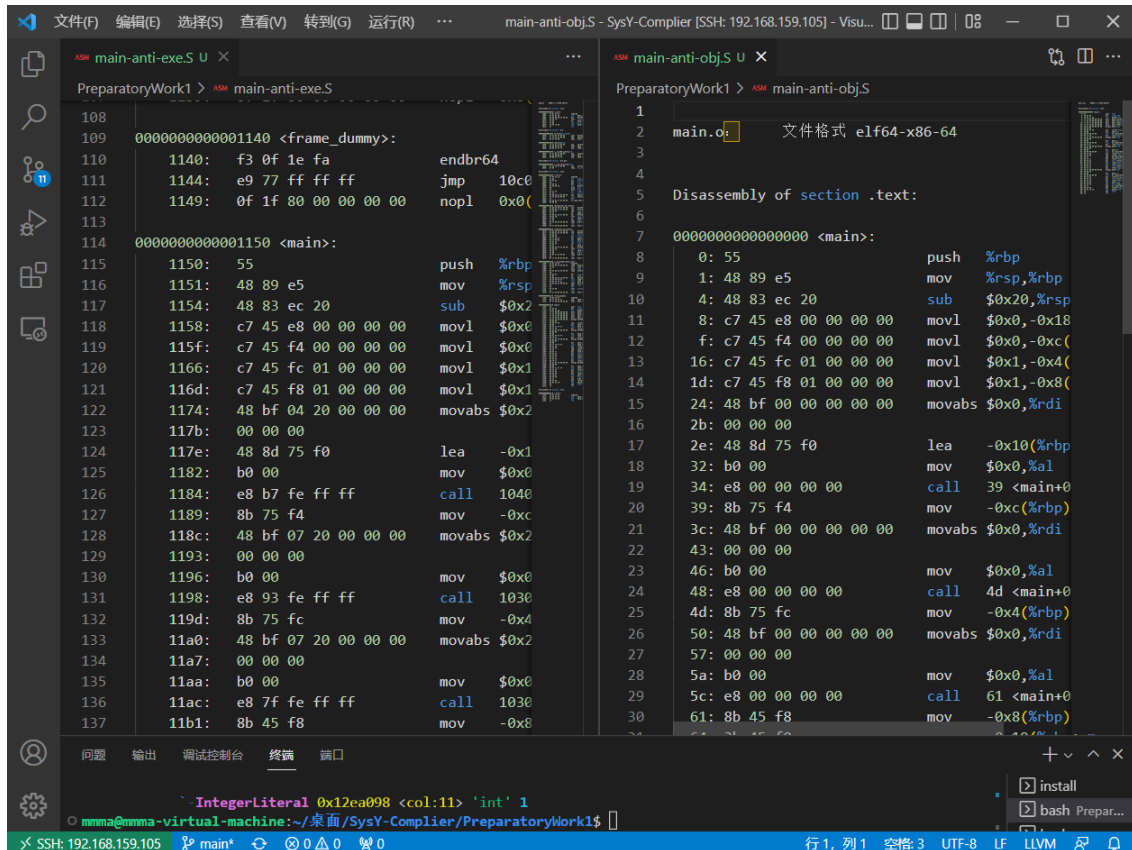


图 8: 可执行文件反汇编后 main 函数部分

除此之外最终可执行文件中多了一些其他的内容，查阅资料后得知链接器和加载器的具体工作如下：

- **链接器** 地址绑定、重定位、库查询、代码分段
- **加载器** 重定位（先由于硬件重定位而省略）

观察反汇编内容后可以看到程序中多了 init、printf、scanf 等部分。且其中的库函数都并没有具体的操作，而是一些跳转指令，且很多跳转的地址在程序中并未找到，分析原因这应当是链接器所做的工作，链接器将对应的函数名与某模块内的某处绑定，通过标明该符号的地址来解析这个符号。如此一来程序执行时可直接通过 call 来调用该库函数执行相应操作等。

```

main-anti-exe.S U X
PreparatoryWork1 > ASM main-anti-exe.S
1
2 main: 文件格式 elf64-x86-64
3
4
5 Disassembly of section .init:
6
7 0000000000001000 <.init>:
8 1000: f3 0f 1e fa      endbr64
9 1004: 48 83 ec 08      sub $0x8,%rax
10 1008: 48 8b 05 d9 2f 00 00 mov 0x2f00d905,%rax
11 100f: 48 85 c0          test %rax,%rax
12 1012: 74 02            je 1016
13 1014: ff d0            call *%rax
14 1016: 48 83 c4 08      add $0x8,%rax
15 101a: c3              ret
16
17 Disassembly of section .plt:
18
19 0000000000001020 <printf@plt-0x10>:
20 1020: ff 35 92 2f 00 00 push 0x2f009235,%rax
21 1026: ff 25 94 2f 00 00 jmp *0x2f009425,%rax
22 102c: 0f 1f 40 00      nopl 0x0(%rax,%rax,1)
23
24 0000000000001030 <printf@plt>:
25 1030: ff 25 92 2f 00 00 jmp *0x2f009225,%rax
26 1036: 68 00 00 00 00 00 push $0x0,%rax
27 103b: e9 e0 ff ff jmp 1020
28
29 0000000000001040 <_isoc99_scanf@plt>:
30 1040: ff 25 8a 2f 00 00 jmp *0x2f008a25,%rax
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2
```



- **表达式** 支持基本的算术运算 (+、-、\*、/、%)、关系运算 (==、!=、<、>、<=、>=) 和逻辑运算 (!、&&、||)，非 0 表示真、0 表示假，而关系运算或逻辑运算的结果用 1 表示真、0 表示假。算符的优先级和结合性以及计算规则 (含逻辑运算的“短路计算”) 与 C 语言一致。

根据 SysY 语言特性编写出如下的源代码：

```
1 //全局变量和常量
2 int globla_var=5;
3 const int maxn=6;
4 //函数定义
5 int mul(int a,int b){
6     return a*b;
7 }
8 int main(){
9     int n;
10    n=getint();
11    //while语句
12    while(n<=10){
13        n=getint();
14    }
15    putint(n);
16    //函数调用
17    int res;
18    res = mul(n, n);
19    putint(res);
20    //隐式转换
21    float f = 1.11;
22    int trans = f + n;
23    putint(trans);
24    //运算符
25    n = n * n;
26    n = n / 2;
27    n = n + 5;
28    n = n - 3;
29    n = n % 10;
30    //数组输入
31    int array[5];
32    int len = getarray(array);
33    //循环 更新数组
34    for(int i=0;i<len;i++){
35        array[i] = i;
36    //条件 跳转
37        if(i<=1){
38            continue;
39        }
40        else{
41            break;
42        }
```

```

43     }
44     array[4] = n;
45 //输出数组
46     putarray(5,array);
47 //判断输入的ASCII码是否是字母
48     printf("Please enter an letter:");
49     int c;
50     c = getch();
51     putchar(c);
52     if((c>=65&& c<=90)|| (c>=97&& c<=122)){
53         printf(" is a Letter!");
54     } else {
55         printf(" is not a Letter!");
56     }
57     return 0;
58 }

```

### 3. LLVM IR 中间语言介绍

传统的静态编译器分为三个阶段：前端、优化和后端。其结构如图10所示

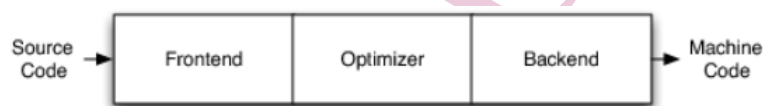


图 10: 静态编译器三阶段

LLVM 的三阶段设计如图11所示

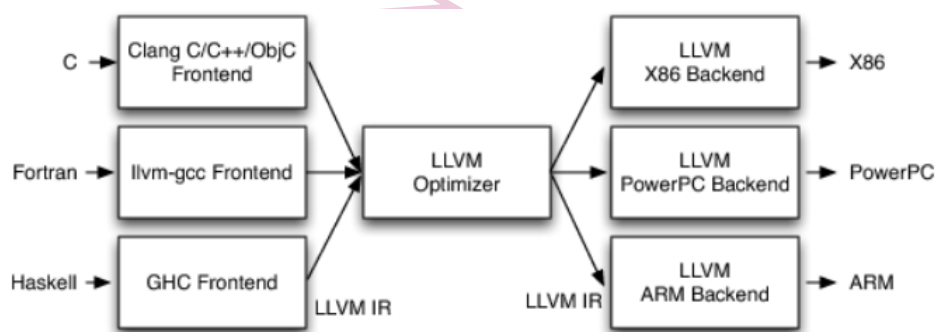


图 11: 静态编译器三阶段

它为高效编译转换和分析，提供了一个强大的中间表示。LLVM 语言轻量、底层、同时富有表现力，类型化，易于扩展。作为一种“通用中间语言”，通过足够低层次，使高级语言可以清晰的映射。

如果支持一种新的编程语言，那么我们只需要实现一种新的前端。如果我们需要支持一种新的硬件设备，那我们只需要实现一个新的后端。而优化阶段因为是针对了统一的 LLVM IR，所以它是一个通用的阶段，不论是支持新的编程语言，还是支持新的硬件设备，这里都不需要对优化阶段做修改。所以从这里可以看出 LLVM IR 的巨大优势。

而 LLVM IR 的表示主要有三种格式：

1. 在内存中的编译中间语言
2. 硬盘上存储的二进制中间语言（以.bc 结尾）
3. 可读的中间格式（以.ll 结尾）

这三种中间格式是完全相等的。

#### 4. LLVM IR 中间语言编程

实验中根据 SysY 语言的特性，设计了相应的 SysY 语言源代码，并按照该源代码手动翻译出 LLVM IR 中间语言，熟悉 SysY 语言特性的同时锻炼 LLVM IR 中间语言的编写能力。

在编写的过程中可以明显的感受到 LLVM IR 中间语言的“通用性”，无论源代码是 C 还是 SysY，都不影响 LLVM IR 中间语言的编写，我们只需要提取源代码的运行逻辑，再使用 LLVM IR 语言实现即可。

最终得到的 IR 中间语言如下：

```

1 ;手动编译
2 ;全局变量和常量声明;全局标识@声明
3 ;int globla_var=5
4 @globla_var = global i32 5, align 4
5 ;const int maxn=6;
6 @maxn = constant i32 6, align 4
7 ;字符串判断提示语句
8 @.str = private unnamed_addr constant [37 x i8] c"Please enter an letter
   :\00", align 1
9 @.str.1 = private unnamed_addr constant [14 x i8] c" is a Letter!\00", align
   1
10 @.str.2 = private unnamed_addr constant [18 x i8] c" is not a Letter!\00",
    align 1
11 ;mul函数定义
12 define i32 @mul(i32 %0, i32 %1) #0 {
13     %3 = alloca i32, align 4           ;%3 i32
14     %4 = alloca i32, align 4           ;
15     store i32 %0, i32* %3, align 4     ;*%3=%0
16     store i32 %1, i32* %4, align 4     ;*%4=%1
17     %5 = load i32, i32* %3, align 4    ;%5=load*%3=%0
18     %6 = load i32, i32* %4, align 4    ;%6=load*%4=%1
19     %7 = mul nsw i32 %5, %6
20     ret i32 %7
21 }
22 ;main函数定义
23 ; int main(){
24 define i32 @main() #0 {
25     %1=alloca i32,align 4               ;%1(i32)
26     %2 = call i32 (...) @getint()      ;%2=n
27     store i32 %2, i32* %1, align 4      ;*%1=%2=n
28     %3 = alloca i32, align 4           ;%3 i32
29     store i32 10, i32* %3, align 4     ;*%3=10
30     %4=load i32, i32* %3, align 4      ;%4=load*%3=10

```

```

31  br label %5
32
33  5:; 循环判断
34  %6=load i32, i32* %1, align 4 ;%6=load*%1=n 用n必须使用%6
35  %7 = icmp sle i32 %6, %4          ;n<=10?
36  br i1 %7,label %8,label %11
37
38  8:;n<=10 要继续循环
39  %9=alloca i32, align 4          ;
40  %10 = call i32 (...) @getint() ;%10=新n
41  store i32 %10, i32* %1, align 4 ;*%1=%10=新n
42  br label %5
43
44  11:;n>10 结束循环 输出
45  %12 = load i32, i32* %1, align 4;%12=load*%1=n
46  %13 = call i32 (i32, ...) @putint to i32 (i32, ...)*(
47      i32 %12);输出%12即n
48  br label %14
49
50  14:; 函数调用部分
51  %15 = alloca i32, align 4          ;%15 i32
52  %16 = call i32 @mul(i32 %12, i32 %12);%16=n*n
53  store i32 %16, i32* %15, align 4 ;*%15=%16
54  %17 = load i32, i32* %15, align 4 ;%17=load*15=n*n
55  %18 = call i32 (i32, ...) @putint to i32 (i32, ...)*(
56      i32 %17)
57  br label %19
58
59  19: ;隐式转换部分
60  %20 = alloca float, align 4
61  store float 0x3FF1C28F60000000, float* %20, align 4; *%20=1.11
62  %21 = load float, float* %20, align 4          ;%21=load*%20=1.11
63  ;隐式转换
64  %22 = sitofp i32 %12 to float ;转n
65  %23 = fadd float %21, %22          ;应该是float格式的f+n
66  %24 = fptosi float %23 to i32      ;再转回来
67  ;结果输出
68  %25 = alloca i32, align 4
69  store i32 %24, i32* %25, align 4; *%25=%24=f+n
70  %26 = load i32, i32* %25, align 4;%26=load*%25=f+n
71  %27 = call i32 (i32, ...) @putint to i32 (i32, ...)*(
72      i32 %26)
73
74  ;n = n * n
75  %28 = load i32, i32* %1, align 4
76  %29 = load i32, i32* %1, align 4
77  %30 = mul nsw i32 %28, %29
78  store i32 %30, i32* %1, align 4

```

```

76
77 ;n = n / 2
78 %31 = load i32, i32* %1, align 4
79 %32 = sdiv i32 %31, 2
80 store i32 %32, i32* %1, align 4
81
82 ;n = n + 5
83 %33 = load i32, i32* %1, align 4
84 %34 = add nsw i32 %33, 5
85 store i32 %34, i32* %1, align 4
86
87 ;n = n - 3
88 %35 = load i32, i32* %1, align 4
89 %36 = sub nsw i32 %35, 3
90 store i32 %36, i32* %1, align 4
91
92 ;n = n % 10
93 %37 = load i32, i32* %1, align 4
94 %38 = srem i32 %37, 10
95 store i32 %38, i32* %1, align 4
96
97 %39 = alloca [5 x i32], align 16 ;数组申请空间 int array[5]
98 %40 = alloca i32, align 4 ;int len
99 %41 = alloca i32, align 4 ;int i
100 ;分配指针
101 %42 = getelementptr inbounds [5 x i32], [5 x i32]* %39, i64 0, i64 0
102 ;len = getarray(array) 函数返回值赋给len 即40
103 %43 = call i32 @i32*, ... bitcast (i32 (...) * @getarray to i32 (i32*, ...)
    *) (i32* %42)
104 store i32 %43, i32* %40, align 4 ;len =
105 store i32 0, i32* %41, align 4 ;i=0
106 br label %44 ;for循环
107 44:
108 %45 = load i32, i32* %41, align 4 ;load i to %45
109 %46 = load i32, i32* %40, align 4 ;load len to %46
110 %47 = icmp slt i32 %45, %46 ;if(i<len)
111 br i1 %47, label %48, label %60
112 48: ;进入for循环块
113 %49 = load i32, i32* %41, align 4 ;load i for valuation
114 %50 = load i32, i32* %41, align 4 ;load i for index
115 %51 = sext i32 %50 to i64 ;signal extend
116 %52 = getelementptr inbounds [5 x i32], [5 x i32]* %39, i64 0, i64 %51 ;get
    the address of array[i]
117 store i32 %49, i32* %52, align 4 ; array[i] = i
118 ;条件 跳转
119 %53 = load i32, i32* %41, align 4 ;if(i<=1)
120 %54 = icmp sle i32 %53, 1
121 br i1 %54, label %55, label %56 ;contine or break

```

```

122 55:
123     br label %57 ;continue
124 56:
125     br label %60 ;break
126 57: ;循环条件判断
127     %58 = load i32, i32* %41, align 4 ;load i to %58
128     %59 = add nsw i32 %58, 1 ;i++
129     store i32 %59, i32* %41, align 4 ;store i
130     br label %44 ;go next for
131 60: ;循环结束
132     %61 = load i32, i32* %1, align 4 ;load n to %61
133     %62 = getelementptr inbounds [5 x i32], [5 x i32]* %39, i64 0, i64 4 ; get
        the address of array[4]
134     store i32 %61, i32* %62, align 16 ;array[4] = n
135     ;输出数组
136     %63 = getelementptr inbounds [5 x i32], [5 x i32]* %39, i64 0, i64 0 ;get
        the address of array
137     %64 = call i32 @i32 (i32, i32*, ...) bitcast (i32 (...) * @putarray to i32 (i32,
        i32*, ...) *) (i32 5, i32* %63) ;putarray(5,array)
138     br label %65
139 65:;输入字符判断
140     %66 = call i32 @i32 (i8*, ...) bitcast (i32 (...) * @putf to i32 (i8*, ...) *) (i8*
        getelementptr inbounds ([37 x i8], [37 x i8]* @.str, i64 0, i64 0))
141     %67 = alloca i32, align 4
142     %68 = call i32 @i32 (...) @getch()
143     store i32 %68, i32* %67, align 4
144     %69 = load i32, i32* %67, align 4;%69=c
145     %70 = call i32 @i32 (i32, ...) bitcast (i32 (...) * @putch to i32 (i32, ...) *) (
        i32 %69)
146     %71 = icmp sge i32 %69, 65
147     br i1 %71, label %72, label %80
148
149 72:;c>=65 -> c<=90?
150     %73 = icmp sle i32 %69, 90
151     br i1 %73, label %78, label %74
152
153 74:;c>90 -> c>=97?
154     %75 = icmp sge i32 %69, 97
155     br i1 %75, label %76, label %80
156
157 76:;c>=97 ->c<=122?
158     %77 = icmp sle i32 %69, 122
159     br i1 %77, label %78, label %80
160 78:;判断是字母
161     %79 = call i32 @i32 (i8*, ...) bitcast (i32 (...) * @putf to i32 (i8*, ...) *) (i8*
        getelementptr inbounds ([14 x i8], [14 x i8]* @.str.1, i64 0, i64 0))
162     ret i32 0
163 80:;不是字母

```

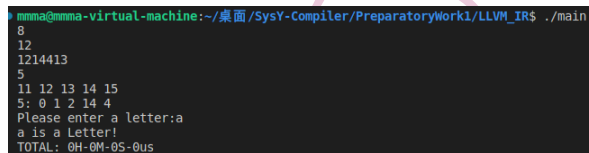
```

164 %81 = call i32 (i8*, ...) bitcast (i32 (...) * @putf to i32 (i8*, ...) *) (i8*
      getelementptr inbounds ([18 x i8], [18 x i8] * @.str.2, i64 0, i64 0))
165 ret i32 0
166 }
167 ;输入输出函数声明
168 declare dso_local i32 @getint(...) #1
169 declare dso_local i32 @putint(...) #1
170 ;数组函数声明
171 declare dso_local i32 @getarray(...) #1
172 declare dso_local i32 @putarray(...) #1
173 ;字符串判断函数声明
174 declare dso_local i32 @getch(...) #1
175 declare dso_local i32 @putch(...) #1
176 declare dso_local i32 @putf(...) #1

```

为验证编写的正确性，实验中使用如下的指令连接并生成可执行文件后运行。

```
1 clang main.ll sylib.c -o main
```



```

mma@mma-virtual-machine:~/桌面/SysY-Compiler/PreparatoryWork1/LLVM_IR$ ./main
8
12
1214413
5
11 12 13 14 15
5: 0 1 2 14 4
Please enter a letter:a
a is a Letter!
TOTAL: 0H-0M-0S-0us

```

图 12: 运行结果

运行结果如图12，可以看到程序正确运行：首先是循环输入  $n$  的值直到  $n \geq 10$ ，之后输出  $n$ 、 $\text{mul}(n,n)$ 、 $n + 1.1$  隐式转换成  $\text{int}$  的值分别为 12、144、13；下一步输入数组，先输入长度 5，之后输入五个数存储到数组中；再然后循环，为索引  $i \leq 2$  的数组元素赋值为  $i$ ，再将  $n$  值存到索引为 4 处，最终输出数组，根据数组的值可以确定之前对  $n$  的运算操作以及循环和分支跳转等部分均是正确执行的；最后输入一个字符判断是否为字母，显然这部分也是正确执行的。

## 五、实验总结

本次实验中通过几个简单的  $\text{C}$  程序样例探讨了编译器编译代码的整个流程，使得对程序从源代码到跑起来这个过程有了一个更深入的了解；此外还进行了 LLVM IR 中间语言的编写，熟悉 LLVM IR 中间语言的同时了解了 SysY 语言的特性和使用以及编译器的工作原理，收获颇丰，相信对后续的实验学习会起到很大的作用。

### (一) 源码链接

实验中的所有源码文件等均已上传到 [Github](#)