南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

编译系统原理试验报告

预备工作 1 了解你的编译器 & LLVM IR 编程

郭坤昌

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 10 月 2 日

摘要

本文主要以 LLVM 编译器为对象，探究语言处理系统的完整工作过程（预处理-编译-汇编-链接-加载）；对 SysY 语言特性进行探究，并编写对应的 LLVM 中间代码。该部分所有工作文件（程序、运行截图、实验数据）保存在代码仓库中。

编译器，SYSY，LLVM IR

目录

一、 预处理器工作

预处理过程将包含的头文件直接插入，完成宏的替换、条件编译等。测试 C++ 代码如下。

测试预处理器工作

```cpp
#include <iostream>
using namespace std;

#define X 10

int main(){
    int a, b, i, t, n;
    a = 0;
    b = 1;
    i = 1;
    cin >> n;
#if X
    cout << "a=" << a << endl;
#else
    cout << "b=" << b << endl;
#endif
    cout << "X=" << X << endl;
    while (i < n){
        t = b;
        b = a + b;
        cout << b << endl;
        a = t;
        i = i + 1;
    }
    return 0;
}
```


使用 clang -E main.cpp 进行预处理，并使用 Scirpt 将预处理结果保存到文件，观察得如下结论：

1. 头文件被嵌入。cpp 文件大小为 454B，嵌入后变为 763.7KB。如图1为部分 iostream 头文件的嵌入

2. 完成宏的替换和条件编译。如图2，main 函数中 X 被替换，条件编译的分支也被确定

```
# 1 "fib.cpp"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 404 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "fib.cpp" 2
# 1 "/usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/c++/11/iostream" 1 3
# 37 "/usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/c++/11/iostream" 3

# 1 "/usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h" 1 3
# 278 "/usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h" 3
namespace std
{
  typedef long unsigned int size_t;
  typedef long int ptrdiff_t;


  typedef decltype(nullptr) nullptr_t;

}
# 300 "/usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h" 3
namespace std
{
  inline namespace __cxx11 __attribute__((__abi_tag__ ("cxx11"))) { }
}
namespace __gnu_cxx
{
  inline namespace __cxx11 __attribute__((__abi_tag__ ("cxx11"))) { }
}
# 586 "/usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h" 3
# 1 "/usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/os_defines.h" 1 3
# 39 "/usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/os_defines.h" 3
# 1 "/usr/include/features.h" 1 3 4
# 392 "/usr/include/features.h" 3 4
# 1 "/usr/include/features-time64.h" 1 3 4
# 20 "/usr/include/features-time64.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 21 "/usr/include/features-time64.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 1 3 4
# 19 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 20 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 2 3 4
# 22 "/usr/include/features-time64.h" 2 3 4
# 393 "/usr/include/features.h" 2 3 4
# 464 "/usr/include/features.h" 3 4
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 465 "/usr/include/features.h" 2 3 4
# 486 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 559 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 560 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
```

图 1: 头文件的嵌入

```
60381 int main()
60382 {
60383  int a, b, i, t, n;
60384
60385  a = 0;
60386  b = 1;
60387  i = 1;
60388  cin >> n;
60389
60390  cout << "a=" << a << endl;
60391
60392
60393
60394  cout << "X=" << 10 << endl;
60395  while (i < n)
60396  {
60397   t = b;
60398   b = a + b;
60399   cout << b << endl;
60400   a = t;
60401   i = i + 1;
60402  }
60403 }
```

图 2: 完成宏的替换和条件编译

二、 编译器工作

(一) 词法分析

词法分析线性地扫描代码，进行分词，识别为标识符（变量）、赋值符、运算符、操作数（如常数等），并使用得到的单词、词素、词法值创建符号表。

使用 clang -E -Xclang -dump-tokens fib.cpp 获得 token 序列，部分结果如图3所示，通过观察验证得如下结论：

1. 语句被顺序地扫描，逐个得到单词
2. 函数、变量被识别为标识符，词法分析还具体地识别出了符号（此处是狭义的非字母符号）的类型（具体到运算类型、作用）
3. 每一行的后半部分记录了该单词的位置（文件名：行号：偏移量，是否为行的开头以及是否有缩进）
4. 对标识符和属性的记录构建了符号表

```
126752  identifier 'main'    [LeadingSpace] Loc=<fib.cpp:6:5>
126753  l_paren '('     Loc=<fib.cpp:6:9>
126754  r_paren ')'     Loc=<fib.cpp:6:10>
126755  l_brace '{'  [StartOfLine]  Loc=<fib.cpp:7:1>
126756  int 'int'    [StartOfLine] [LeadingSpace]   Loc=<fib.cpp:8:2>
126757  identifier 'a'    [LeadingSpace] Loc=<fib.cpp:8:6>
126758  comma ','       Loc=<fib.cpp:8:7>
126759  identifier 'b'    [LeadingSpace] Loc=<fib.cpp:8:9>
126760  comma ','       Loc=<fib.cpp:8:10>
126761  identifier 'i'    [LeadingSpace] Loc=<fib.cpp:8:12>
126762  comma ','       Loc=<fib.cpp:8:13>
126763  identifier 't'    [LeadingSpace] Loc=<fib.cpp:8:15>
126764  comma ','       Loc=<fib.cpp:8:16>
126765  identifier 'n'    [LeadingSpace] Loc=<fib.cpp:8:18>
126766  semi ';'        Loc=<fib.cpp:8:19>
126767  identifier 'a'    [StartOfLine] [LeadingSpace]   Loc=<fib.cpp:10:2>
126768  equal '='    [LeadingSpace] Loc=<fib.cpp:10:4>
126769  numeric_constant '0'     [LeadingSpace] Loc=<fib.cpp:10:6>
126770  semi ';'        Loc=<fib.cpp:10:7>
126771  identifier 'b'    [StartOfLine] [LeadingSpace]   Loc=<fib.cpp:11:2>
126772  equal '='    [LeadingSpace] Loc=<fib.cpp:11:4>
126773  numeric_constant '1'     [LeadingSpace] Loc=<fib.cpp:11:6>
126774  semi ';'        Loc=<fib.cpp:11:7>
126775  identifier 'i'    [StartOfLine] [LeadingSpace]   Loc=<fib.cpp:12:2>
126776  equal '='    [LeadingSpace] Loc=<fib.cpp:12:4>
126777  numeric_constant '1'     [LeadingSpace] Loc=<fib.cpp:12:6>
126778  semi ';'        Loc=<fib.cpp:12:7>
126779  identifier 'cin'     [StartOfLine] [LeadingSpace]   Loc=<fib.cpp:13:2>
126780  greatergreater '>>'  [LeadingSpace] Loc=<fib.cpp:13:6>
126781  identifier 'n'    [LeadingSpace] Loc=<fib.cpp:13:9>
126782  semi ';'        Loc=<fib.cpp:13:10>
126783  identifier 'cout'    [StartOfLine] [LeadingSpace]   Loc=<fib.cpp:15:2>
126784  lessless '<<'    [LeadingSpace] Loc=<fib.cpp:15:7>
126785  string_literal '"a="'    [LeadingSpace] Loc=<fib.cpp:15:10>
126786  lessless '<<'    [LeadingSpace] Loc=<fib.cpp:15:15>
126787  identifier 'a'    [LeadingSpace] Loc=<fib.cpp:15:18>
126788  lessless '<<'    [LeadingSpace] Loc=<fib.cpp:15:20>
126789  identifier 'endl'    [LeadingSpace] Loc=<fib.cpp:15:23>
126790  semi ';'        Loc=<fib.cpp:15:27>
126791  identifier 'cout'    [StartOfLine] [LeadingSpace]   Loc=<fib.cpp:19:2>
126792  lessless '<<'    [LeadingSpace] Loc=<fib.cpp:19:7>
```

图 3: 词法分析结果

(二) 语法分析

语法分析使用层次分析的方法将 token 序列转化为语法分析树，树的内部结点为抽象出的语法概念，叶结点为单词。

使用 clang -E -Xclang -ast-dump fib.cpp 生成文本形式的 AST，对其中如图4所示的 while 子树观察得到如下结论：

1. 对于函数、while 语句等，抽象出的内部结点明确指明了该语句/函数的类型、地址。
2. 对于较为简单的赋值/运算语句，其上层节点则是对应二进制操作符，下层节点是对应的表达式。特别地，对该句运算，可以看到根节点规定了运算的结果类型，子结点也说明了操作数的类型、地址。
3. 最上层的抽象定义为 TranslationUnitDecl，即声明该翻译的模块，其子结点由类型声明、函数声明（包含了 main 函数定义）等组成

```
`-WhileStmt 0x1969888 <line:17:2, line:24:2>
  |-BinaryOperator 0x1963290 <line:17:8, col:10> 'bool' '<'
  | |-ImplicitCastExpr 0x1963260 <col:8> 'int' <LValueToRValue>
  | | `-DeclRefExpr 0x1963220 <col:8> 'int' lvalue Var 0x1952180 'i' 'int'
  | `-ImplicitCastExpr 0x1963278 <col:10> 'int' <LValueToRValue>
  |   `-DeclRefExpr 0x1963240 <col:10> 'int' lvalue Var 0x1952280 'n' 'int'
  `-CompoundStmt 0x1969850 <line:18:2, line:24:2>
    |-BinaryOperator 0x1963308 <line:19:3, col:5> 'int' lvalue '='
    | |-DeclRefExpr 0x19632b0 <col:3> 'int' lvalue Var 0x1952200 't' 'int'
    | `-ImplicitCastExpr 0x19632f0 <col:5> 'int' <LValueToRValue>
    |   `-DeclRefExpr 0x19632d0 <col:5> 'int' lvalue Var 0x1952100 'b' 'int'
    |-BinaryOperator 0x19633d8 <line:20:3, col:7> 'int' lvalue '='
    | |-DeclRefExpr 0x1963328 <col:3> 'int' lvalue Var 0x1952100 'b' 'int'
    | `-BinaryOperator 0x19633b8 <col:5, col:7> 'int' '+'
    |   |-ImplicitCastExpr 0x1963388 <col:5> 'int' <LValueToRValue>
    |   | `-DeclRefExpr 0x1963348 <col:5> 'int' lvalue Var 0x1952080 'a' 'int
    |   `-ImplicitCastExpr 0x19633a0 <col:7> 'int' <LValueToRValue>
    |     `-DeclRefExpr 0x1963368 <col:7> 'int' lvalue Var 0x1952100 'b' 'int
    |-CXXOperatorCallExpr 0x19696e8 <line:21:3, col:12> 'std::basic_ostream<c
    | |-ImplicitCastExpr 0x19696d0 <col:10> 'std::basic_ostream<char>::__ostr
<FunctionToPointerDecay>
    | | `-DeclRefExpr 0x19696b0 <col:10> 'std::basic_ostream<char>::__ostrea
e CXXMethod 0x18d48b8 'operator<<' 'std::basic_ostream<char>::__ostream_type
    | |-CXXOperatorCallExpr 0x1968d38 <col:3, col:9> 'std::basic_ostream<char
    | | |-ImplicitCastExpr 0x1968d20 <col:7> 'std::basic_ostream<char>::__ost
    | | | `-DeclRefExpr 0x1968d00 <col:7> 'std::basic_ostream<char>::__ostrea

    | | |-DeclRefExpr 0x19633f8 <col:3> 'std::ostream':'std::basic_ostream<ch
    | | `-ImplicitCastExpr 0x1968ce8 <col:9> 'int' <LValueToRValue>
    | |   `-DeclRefExpr 0x1963418 <col:9> 'int' lvalue Var 0x1952100 'b' 'int
    | `-ImplicitCastExpr 0x1969698 <col:12> 'basic_ostream<char, std::char_tr
    |   `-DeclRefExpr 0x1969670 <col:12> 'basic_ostream<char, std::char_trait
n<char, std::char_traits<char>> &(basic_ostream<char, std::char_traits<char>>
    |-BinaryOperator 0x1969778 <line:22:3, col:5> 'int' lvalue '='
    | |-DeclRefExpr 0x1969720 <col:3> 'int' lvalue Var 0x1952080 'a' 'int'
    | `-ImplicitCastExpr 0x1969760 <col:5> 'int' <LValueToRValue>
    |   `-DeclRefExpr 0x1969740 <col:5> 'int' lvalue Var 0x1952200 't' 'int'
    `-BinaryOperator 0x1969830 <line:23:3, col:7> 'int' lvalue '='
      |-DeclRefExpr 0x1969798 <col:3> 'int' lvalue Var 0x1952180 'i' 'int'
      `-BinaryOperator 0x1969810 <col:5, col:7> 'int' '+'
        |-ImplicitCastExpr 0x19697f8 <col:5> 'int' <LValueToRValue>
```

图 4: Caption

(三) 语义分析

确保程序和语法定义是一致的，收集类型信息并保存到 AST 或符号表中（为中间代码生成收集信息）。

不符合语法定义的错误有：

1. 类型不匹配
2. 变量未声明
3. 错误使用自留字符作为变量（标识符）

语义分析的具体功能[1]：

1. 类型检查，确保类型一致，也会进行类型的转换
2. 标签检查，程序需要保存标签的引用（实际上就是保留了标签指向的相对位置）
3. 控制流检查，如循环、switch 外部不能使用 break

语义分析的检查方式：

1. 静态检查：编译时检查
2. 动态检查：运行时确定表达式和语句的含义

[1] semantic analysis in compiler design

（四） 中间代码生成

中间代码生成过程中，将上一步语法分析的结果，通过一系列步骤，转化为类似汇编代码的程序。使用中间代码的优点在于：

1. 生成的中间代码不依赖于实际体系结构，可移植性强，这样就不用对每一种架构都配置单独的编译器

2. 容易重定向（只需指定变量开始分配的地址为 0, 则可以通过相对位置为变量分配内存空间，重定向时只需加上基址即可）

3. 通过优化中间代码来进行性能优化。高级中间代码更接近源码，在生成中间代码时进行优化；低级中间代码更接近目标语言，针对机器特性进行优化。

使用 g++ 生成控制流图——图5为源代码的 CFG；图6为 SSA 转化后的 CFG（观察到对于同名变量，每一次声明、赋值的标识符均不相同，符合 SSA 的基本定义——static single assignment，要求每个变量在使用前被定义，且只能被赋值一次）；图7为经过一系列优化并改写为对寄存器操作的汇编代码的 CFG。

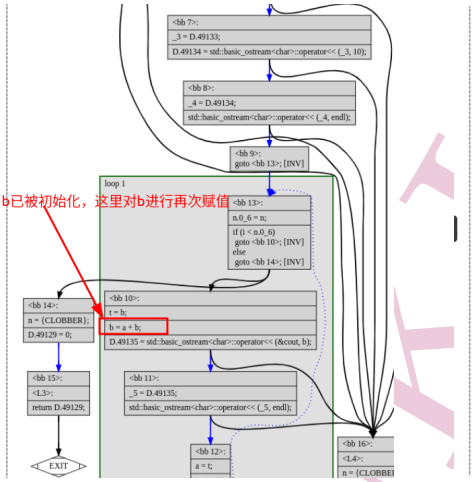


图 5: origin

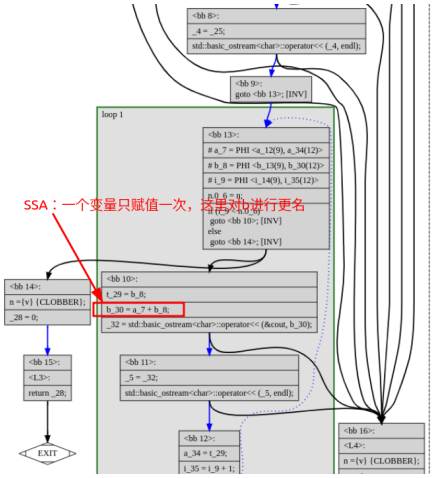


图 6: ssa

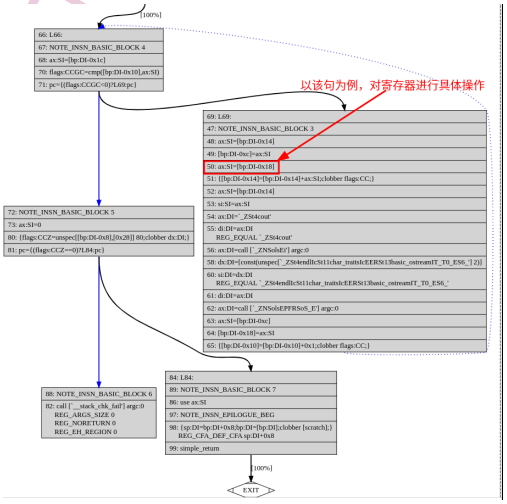


图 7: final

（五） 代码优化

代码的优化是通过对程序的部分进行分析和变换来实现的，LLVM 的 pass 分为三种策略：

1. analysis pass：为调试和可视化收集信息
2. transform pass：可选地调用 analysis pass 的结果，对程序进行变换
3. utility pass：提供一些实用程序

以如下程序为例，尝试使用三种策略中的一些 pass 进行验证。

示例 C 程序

```c
const int n = 6;
float a[6] = {1, 2, 3, 4, 5, 6};
int m;

void loop(){
    for (int i = 0; i < n; i++) {
        a[i] = a[i] * 3;
        a[i] = a[i] / 2;
    }
}

int main()  {
    if(0) m=1;
    loop();
    return 0;
}
```

使用 analysis pass 中的 cfg-dot[1]打印 CFG 控制流程图，如图17所示。通过该控制流程图可以清楚地看到每个 block 的支配，可用于构建支配节点树 (dominator tree)，它对于 ssa 过程中插入 phi 节点至关重要（确定控制流的边界，提高插入的精准度以减小开销）。

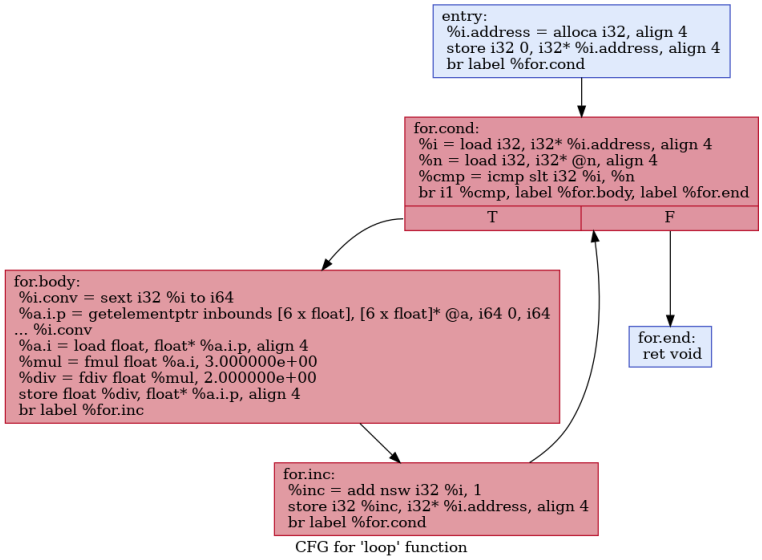


图 8: loop 函数 cfg

[1]LLVM 不生成 CFG 原因及解答

使用 transform pass 中的 mem2reg（将对内存的引用替换为对寄存器的操作，将 load 和 store 指令用 phi 语句优化），如图9所示。

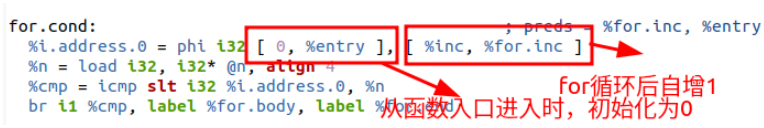


图 9: mem2reg pass

另外，设计了验证死代码删除和确定循环深度的优化，但使用 opt 直接验证时，没有取得预期结果（即使禁用了 O0 下的 optnone 属性），猜测是因为没有取得足够的 analysis 信息以支持优化过程。在 main 函数中增加 if(0) m=1; 生成的代码中删除了该句死代码，如图10所示。for 循环的循环次数被确定，如图11。

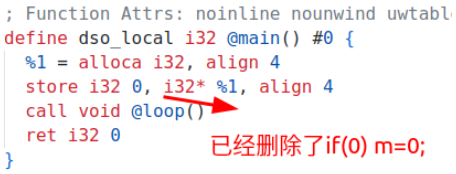


图 10: 死代码优化

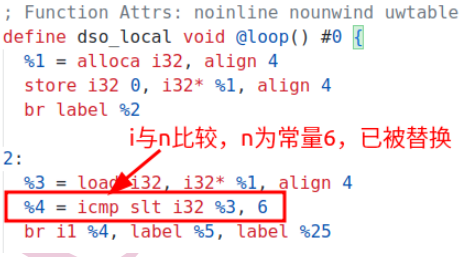


图 11: for 循环次数确定

使用 utility pass 中的-view-dom 查看支配节点树，如图12。

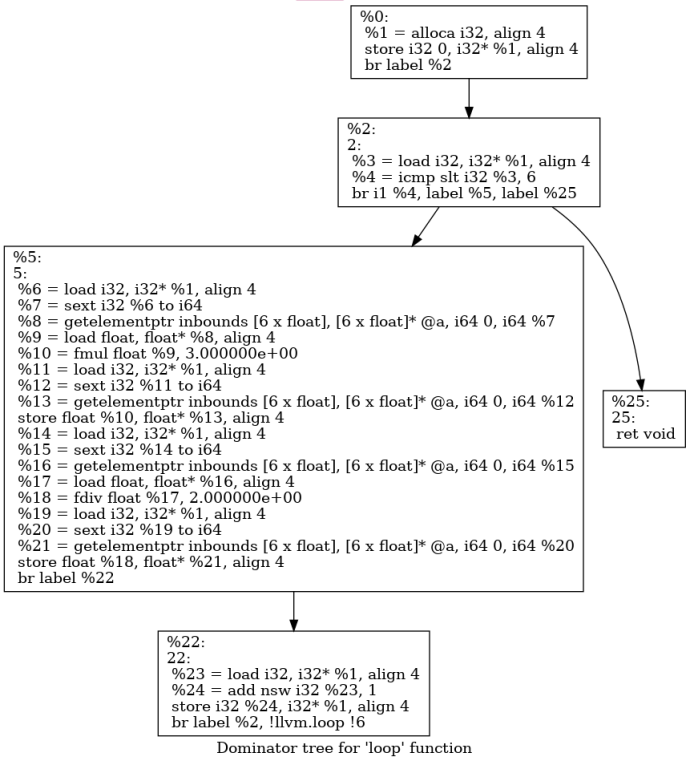


图 12: loop 函数的支配节点树

三、 汇编器工作

汇编器[1]用于将用汇编代码转换为可重定位的机器码并为链接加载程序生成信息。汇编器通过分析操作符来生成指令，并找到符号和文字的值来生成机器码，工作流程分为两趟：

第一趟：

1. 定义符号和文字，并在符号表和文字表中保存它们。当扫描到符号（变量、标签等）和文字（常数等），就将它们和对应位置（位置计数器的值，始终指向当前行）填入符号表和文字表中。
2. 追踪位置计数器（Location Counter）。位置计数器首先被初始化为 0，每次扫描完一行后，加一指向下一行。
3. 处理伪操作。伪操作没有对应的机器指令，是指示汇编过程的命令指令，如指示程序如何分段等，以实现高级功能。伪指令的操作同样保存在伪指令表中。

第二趟：

1. 使用第一趟中得到的表，将符号操作码转换为数字操作码，生成机器码。
2. 为文字生成数据并查找符号的值。

以如下 C++ 程序为例：

简单示例程序

```
1  int main()
2  {
3          int a, b, i, t, n;
4          a = 0;
5          b = 1;
6          i = 1;
7          while (i < n){
8                  t = b;
9                  b = a + b;
10                 a = t;
11                 i = i + 1;
12         }
13 }
```

得到的汇编代码如图13所示。图中标出了 main 函数的标签，while 循环的比较及何时退出的跳转关系。

从可重定位机器码反汇编，结果如图14。注意到汇编代码中的伪指令（指示编译过程的命令指令）完全去除，标签被替换为了具体的地址。这也验证了部分汇编器的工作：处理伪指令（编译命令）、为文字生成数据并查找符号的值（地址的替换）。

[1] introduction of assembler

```
1         .text
2         .file   "main.cpp"
3         .globl  main                            # -- Begin function main
4         .p2align        4, 0x90
5         .type   main,@function
6 main:                                           # @main
7         .cfi_startproc
8 # %bb.0:
9         pushq   %rbp
10        .cfi_def_cfa_offset 16
11        .cfi_offset %rbp, -16
12        movq    %rsp, %rbp
13        .cfi_def_cfa_register %rbp
14        movl    $0, -20(%rbp)
15        movl    $0, -12(%rbp)
16        movl    $1, -8(%rbp)
17        movl    $1, -4(%rbp)
18 .LBB0_1:                                        # =>This Inner Loop Header: Depth=1
19        movl    -4(%rbp), %eax
20        cmpl    -24(%rbp), %eax
21        jge     .LBB0_3
22 # %bb.2:                                        #   in Loop: Header=BB0_1 Depth=1
23        movl    -8(%rbp), %eax
24        movl    %eax, -16(%rbp)
25        movl    -12(%rbp), %eax
26        addl    -8(%rbp), %eax
27        movl    %eax, -8(%rbp)
28        movl    -16(%rbp), %eax
29        movl    %eax, -12(%rbp)
30        movl    -4(%rbp), %eax
31        addl    $1, %eax
32        movl    %eax, -4(%rbp)
33        jmp     .LBB0_1
34 .LBB0_3:
35        movl    -20(%rbp), %eax
36        popq    %rbp
37        .cfi_def_cfa %rsp, 8
38        retq
39 .Lfunc_end0:
40        .size   main, .Lfunc_end0-main
41        .cfi_endproc
42                                                  # -- End function
43        .ident  "Ubuntu clang version 14.0.0-1ubuntu1"
44        .section        ".note.GNU-stack","",@progbits
```

图 13: 汇编代码

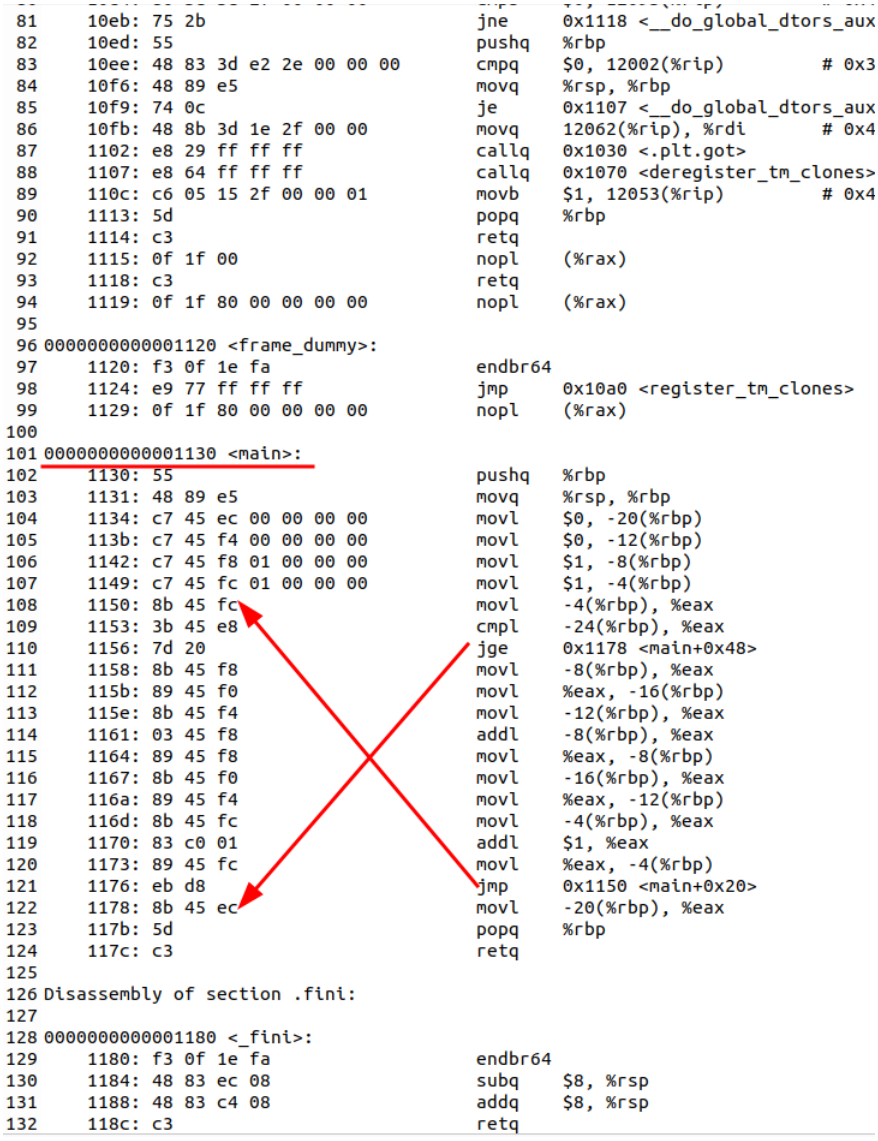
```
1
2 main.o: file format elf64-x86-64
3
4 Disassembly of section .text:
5
6 0000000000000000 <main>:
7        0: 55                                    pushq   %rbp
8        1: 48 89 e5                              movq    %rsp, %rbp
9        4: c7 45 ec 00 00 00 00                  movl    $0, -20(%rbp)
10       b: c7 45 f4 00 00 00 00                  movl    $0, -12(%rbp)
11      12: c7 45 f8 01 00 00 00                  movl    $1, -8(%rbp)
12      19: c7 45 fc 01 00 00 00                  movl    $1, -4(%rbp)
13      20: 8b 45 fc                              movl    -4(%rbp), %eax
14      23: 3b 45 e8                              cmpl    -24(%rbp), %eax
15      26: 7d 20                                 jge     0x48 <main+0x48>
16      28: 8b 45 f8                              movl    -8(%rbp), %eax
17      2b: 89 45 f0                              movl    %eax, -16(%rbp)
18      2e: 8b 45 f4                              movl    -12(%rbp), %eax
19      31: 03 45 f8                              addl    -8(%rbp), %eax
20      34: 89 45 f8                              movl    %eax, -8(%rbp)
21      37: 8b 45 f0                              movl    -16(%rbp), %eax
22      3a: 89 45 f4                              movl    %eax, -12(%rbp)
23      3d: 8b 45 fc                              movl    -4(%rbp), %eax
24      40: 83 c0 01                              addl    $1, %eax
25      43: 89 45 fc                              movl    %eax, -4(%rbp)
26      46: eb d8                                 jmp     0x20 <main+0x20>
27      48: 8b 45 ec                              movl    -20(%rbp), %eax
28      4b: 5d                                    popq    %rbp
29      4c: c3                                    retq
```

图 14: 从可重定位机器代码反汇编结果

四、 链接器与加载器工作

链接器将多个可重定位机器码文件合并，查找相关的库文件，并组织每个模块在内存中的位置，最终得到可执行文件。静态链接和动态链接的区别在于：静态链接将引用的库文件中的代码包括在可执行文件中；动态链接则不必将动态链接库文件包含，而在运行时动态地链接和卸载动态链接库文件。

如图15为将上一节代码生成的可执行文件反汇编得到的结果，注意到其与图14将机器代码反汇编结果的不同（即使在这里没有使用静态编译，也能看到链接的效果），在于增加了许多模块，并组织了它们在内存中的地址。图15标注了 main 函数模块在内存的地址及其中 while 循环的跳转关系。



```
 81     10eb: 75 2b                         jne     0x1118 <__do_global_dtors_aux
 82     10ed: 55                            pushq   %rbp
 83     10ee: 48 83 3d e2 2e 00 00 00       cmpq    $0, 12002(%rip)         # 0x3
 84     10f6: 48 89 e5                      movq    %rsp, %rbp
 85     10f9: 74 0c                         je      0x1107 <__do_global_dtors_aux
 86     10fb: 48 8b 3d 1e 2f 00 00          movq    12062(%rip), %rdi       # 0x4
 87     1102: e8 29 ff ff ff                callq   0x1030 <.plt.got>
 88     1107: e8 64 ff ff ff                callq   0x1070 <deregister_tm_clones>
 89     110c: c6 05 15 2f 00 00 01          movb    $1, 12053(%rip)         # 0x4
 90     1113: 5d                            popq    %rbp
 91     1114: c3                            retq
 92     1115: 0f 1f 00                      nopl    (%rax)
 93     1118: c3                            retq
 94     1119: 0f 1f 80 00 00 00 00          nopl    (%rax)
 95
 96 0000000000001120 <frame_dummy>:
 97     1120: f3 0f 1e fa                   endbr64
 98     1124: e9 77 ff ff ff                jmp     0x10a0 <register_tm_clones>
 99     1129: 0f 1f 80 00 00 00 00          nopl    (%rax)
100
101 0000000000001130 <main>:
102     1130: 55                            pushq   %rbp
103     1131: 48 89 e5                      movq    %rsp, %rbp
104     1134: c7 45 ec 00 00 00 00          movl    $0, -20(%rbp)
105     113b: c7 45 f4 00 00 00 00          movl    $0, -12(%rbp)
106     1142: c7 45 f8 01 00 00 00          movl    $1, -8(%rbp)
107     1149: c7 45 fc 01 00 00 00          movl    $1, -4(%rbp)
108     1150: 8b 45 fc                      movl    -4(%rbp), %eax
109     1153: 3b 45 e8                      cmpl    -24(%rbp), %eax
110     1156: 7d 20                         jge     0x1178 <main+0x48>
111     1158: 8b 45 f8                      movl    -8(%rbp), %eax
112     115b: 89 45 f0                      movl    %eax, -16(%rbp)
113     115e: 8b 45 f4                      movl    -12(%rbp), %eax
114     1161: 03 45 f8                      addl    -8(%rbp), %eax
115     1164: 89 45 f8                      movl    %eax, -8(%rbp)
116     1167: 8b 45 f0                      movl    -16(%rbp), %eax
117     116a: 89 45 f4                      movl    %eax, -12(%rbp)
118     116d: 8b 45 fc                      movl    -4(%rbp), %eax
119     1170: 83 c0 01                      addl    $1, %eax
120     1173: 89 45 fc                      movl    %eax, -4(%rbp)
121     1176: eb d8                         jmp     0x1150 <main+0x20>
122     1178: 8b 45 ec                      movl    -20(%rbp), %eax
123     117b: 5d                            popq    %rbp
124     117c: c3                            retq
125
126 Disassembly of section .fini:
127
128 0000000000001180 <_fini>:
129     1180: f3 0f 1e fa                   endbr64
130     1184: 48 83 ec 08                   subq    $8, %rsp
131     1188: 48 83 c4 08                   addq    $8, %rsp
132     118c: c3                            retq
```

图 15: 可执行文件反汇编结果

加载器则将可执行文件加载到内存中并为其分配地址，同时调度程序中的引用（与动态链接有关）。

五、 SysY 语言特性与对应 LLVM IR 程序编写

该部分工作由本人郭坤昌和队友吴晨宇共同合作完成。具体分工为:

1. 共同研究 SysY 语言特性，设计相应 C++ 程序

2. 郭坤昌编写 loop 函数和 print 函数；吴晨宇编写 fib 函数和 main 函数。并各自与生成的 IR 代码对照，分析其特点。

实验思路：根据 SysY 文法和语义约束设计应用思路，设计相应 c++ 程序，编写对应的 LLVM 中间代码，并通过将 c++ 程序翻译为中间代码，对照分析，验证手写中间代码的正确性。

设计如下的 C++ 程序。

示例 C++ 程序

```cpp
#include <stdio.h>

const int n = 6;
float a[6] = {1, 2, 3, 4, 5, 6};

void loop() {
    for (int i = 0; i < n; i++) {
        a[i] = a[i] * 3;
        a[i] = a[i] / 2;
    }
}

void print() {
    for (int i = 0; i < n; i++)
        printf("%f ", a[i]);
}

// 求斐波那契数列
int fib(int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}

int main() {
    loop();
    print();
    return fib(10);
}
```


该程序使用了如下的 SysY 语言特性:

1. 编译单元。以常量/变量/函数声明开始，且必须有唯一的 main 函数。

2. 常量和变量的声明。常量声明以 const 开始，声明为基本类型（int 和 float）。

3. 函数定义。需要指明函数返回值类型，名称，形参列表，函数块。参数类型包括 int、float 和 void，可以传递数组；块由大括号括起。

4. 语句由赋值语句、表达式、语句块、if 语句、while 语句、控制跳转语句、返回语句等组成。
5. 表达式的基本类型为加减表达式。
6. 函数实参由一个或多个表达式组成。
7. 相等表达式表示相等判断的结果。

……

手写为对应的 LLVM IR code 如下。

全局变量的声明对应为指针类型。

全局变量声明

```
@n = dso_local constant i32 6, align 4
@a = dso_local global [6 x float] [float 1.000000e+00, float 2.000000e+00,
    float 3.000000e+00, float 4.000000e+00, float 5.000000e+00, float
    6.000000e+00], align 16
@.str = private unnamed_addr constant [4 x i8] c"%f \00", align 1
```

取数组元素的过程借鉴了自动翻译得到的 llvm ir 代码，其中需要将偏移量指定为 i64 类型（也就指明了元素间的间隔其实是 8 个字节，float 其实只占 4 个字节），否则无法正常取到 float 数组中的元素。

这里故意留下了一些优化的空间：

1. n 的值是全局指定的，可以在编译时确定循环的深度。以下的代码优化过程，将在 analyze 中获取循环的次数，并在 transform 中应用该种优化。

2. 可以通过循环展开、SIMD 方式优化提高运行速度。

loop 函数

```
define void @loop() {
entry:
    %i.address = alloca i32, align 4
    store i32 0, i32* %i.address, align 4
    br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
    %i = load i32, i32* %i.address, align 4
    %n = load i32, i32* @n, align 4
    %cmp = icmp slt i32 %i, %n
    br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
    %i.conv = sext i32 %i to i64
    %a.i.p = getelementptr inbounds [6 x float], [6 x float]* @a, i64 0, i64
        %i.conv
    %a.i = load float, float* %a.i.p, align 4
    %mul = fmul float %a.i, 3.000000e+00
    %div = fdiv float %mul, 2.000000e+00
    store float %div, float* %a.i.p, align 4
    br label %for.inc

for.inc:                                          ; preds = %for.body
    %inc = add nsw i32 %i, 1
```

```
    store i32 %inc, i32* %i.address, align 4
    br label %for.cond

for.end:                                          ; preds = %for.cond
    ret void
}
```

调用 stdio.o 中的函数 printf 完成输出。

print 函数

```
define void @print() {
entry:
    %i.address = alloca i32, align 4    ;i=0
    store i32 0, i32* %i.address, align 4
    br label %for.cond

for.cond:                                         ; preds = %for.inc, %
    entry
    %i = load i32, i32* %i.address, align 4
    %cmp = icmp slt i32 %i, 6
    br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
    %0 = sext i32 %i to i64
    %arrayidx = getelementptr inbounds [6 x float], [6 x float]* @a, i64 0,
        i64 %0
    %1 = load float, float* %arrayidx, align 4
    %2 = fpext float %1 to double
    %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
         [4 x i8]* @.str, i64 0, i64 0), double %2)
    br label %for.inc

for.inc:                                          ; preds = %for.body
    %3 = load i32, i32* %i.address, align 4
    %inc = add nsw i32 %3, 1
    store i32 %inc, i32* %i.address, align 4
    br label %for.cond

for.end:                                          ; preds = %for.cond
    ret void
}

declare i32 @printf(i8* noundef, ...)
```

求斐波那契数列。

fib 函数

```
define i32 @fib(i32 %n) {
entry:
```

```
  %cmp = icmp eq i32 %n, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  ret i32 0

if.else:                                          ; preds = %entry
  %cmp1 = icmp eq i32 %n, 1
  br i1 %cmp1, label %if.then2, label %if.else3

if.then2:                                         ; preds = %if.else
  ret i32 1

if.else3:                                         ; preds = %if.else
  %sub = sub i32 %n, 1
  %call = call i32 @fib(i32 %sub)
  %sub4 = sub i32 %n, 2
  %call5 = call i32 @fib(i32 %sub4)
  %add = add i32 %call, %call5
  ret i32 %add
}
```

main 函数执行 loop 函数，并展示结果；最后返回 fib(10) 的返回值。

main 函数

```
define dso_local i32 @main() {
entry:
    call void @loop()
    call void @print()
    %call = call i32 @fib(i32 10)
    ret i32 %call
}
```

解释并执行中间代码，使用 echo $? 输出上一次运行返回值，如图16所示数组的输出均为原值的 1.5 倍，fib(10)=55, 执行正确。

```
(base) bill@bill-Lenovo-V15-IWL:~/Desktop/compilation/lab/lab1
1.500000 3.000000 4.500000 6.000000 7.500000 9.000000 (base) b
/lab/lab1/program/loop$ echo $?
55
```

图 16: ir 执行结果

对比翻译得到的 IR 代码和编写的 IR 代码，除去增加了较多属性外，在主要的代码运行逻辑上，主要有如图17和如图18的不同。

在 loop 函数中，对于数组 a 中元素的乘除运算，每次均取出了操作数和目的的地址，且没能利用好局部性原理，重复取数，其实可以连续利用。有优化的地方在于确定了循环的深度，也就是将全局 n 的值直接带入，减少了访问次数（label 2 的第二行，用于比较的第二个参数为 6 即全局变量 n 的值）。

在 fib 函数中，则首先指定了存储返回值的临时变量，在每一种分支条件下，均能将返回值存在该变量中，另外用一个临时变量保存了传入的参数 n，在每次使用 n 时重新保存到新的变量中，与编写的方式相比，不够简洁。

```
define dso_local void @loop() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  br label %2

2:                                                ; preds = %22, %0
  %3 = load i32, i32* %1, align 4
  %4 = icmp slt i32 %3, 6
  br i1 %4, label %5, label %25

5:                                                ; preds = %2
  %6 = load i32, i32* %1, align 4
  %7 = sext i32 %6 to i64
  %8 = getelementptr inbounds [6 x float], [6 x float]* @a, i64 0, i64 %7
  %9 = load float, float* %8, align 4
  %10 = fmul float %9, 3.000000e+00
  %11 = load i32, i32* %1, align 4
  %12 = sext i32 %11 to i64
  %13 = getelementptr inbounds [6 x float], [6 x float]* @a, i64 0, i64 %12
  store float %10, float* %13, align 4
  %14 = load i32, i32* %1, align 4
  %15 = sext i32 %14 to i64
  %16 = getelementptr inbounds [6 x float], [6 x float]* @a, i64 0, i64 %15
  %17 = load float, float* %16, align 4
  %18 = fdiv float %17, 2.000000e+00
  %19 = load i32, i32* %1, align 4
  %20 = sext i32 %19 to i64
  %21 = getelementptr inbounds [6 x float], [6 x float]* @a, i64 0, i64 %20
  store float %18, float* %21, align 4
  br label %22
```

取乘法操作数

取乘法地址

取除法操作数

取除法地址

图 17: loop 函数中的不同

```
define dso_local i32 @fib(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  %4 = load i32, i32* %3, align 4
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %7

6:                                                ; preds = %1
  store i32 0, i32* %2, align 4
  br label %19

7:                                                ; preds = %1
  %8 = load i32, i32* %3, align 4
  %9 = icmp eq i32 %8, 1
  br i1 %9, label %10, label %11

10:                                               ; preds = %7
  store i32 1, i32* %2, align 4
  br label %19

11:                                               ; preds = %7
  %12 = load i32, i32* %3, align 4
  %13 = sub nsw i32 %12, 1
  %14 = call i32 @fib(i32 noundef %13)
  %15 = load i32, i32* %3, align 4
  %16 = sub nsw i32 %15, 2
  %17 = call i32 @fib(i32 noundef %16)
  %18 = add nsw i32 %14, %17
  store i32 %18, i32* %2, align 4
  br label %19

19:                                               ; preds = %11, %10, %6
  %20 = load i32, i32* %2, align 4
  ret i32 %20
}
```

%2为存储返回值的临时变量

%3为存储参数n的临时变量

图 18: fib 函数中的不同