预备工作 1——了解编译器，LLVM IR 编程及汇编编程

杨俣哲 李煦阳
孙一丁 李世阳 杨科迪
周辰霏 尧泽斌 时浩铭
贺祎昕 张书睿
华志远 李帅东

2020 年 9 月—2024 年 9 月

目录

1 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：

1. 完整的编译过程都有什么？
2. 预处理器做了什么？
3. 编译器做了什么？
4. 汇编器做了什么？
5. 链接器做了什么？
6. 通过编写 LLVM IR 程序，熟悉 LLVM IR 中间语言。
7. 通过编写汇编程序，熟悉 ARM/RISC-V 汇编语言

并尽可能地对其实现方式有所了解。

1.1 方法

以一个简单的 C（C++）源程序为例，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，以此撰写调研报告。二进制文件或许需要利用某些系统工具理解，如 `objdump`、`nm`。

进一步地，可以调整你认为关键的**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。

你的源程序可以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...），以更全面探索每一个阶段编译器进行的工作。

1.2 实验要求

要求：

撰写调研报告（符合科技论文写作规范，包含完整结构：题目、摘要、关键字、引言、你的工作和结果的具体介绍、结论、参考文献，文字、图、表符合格式规范，建议使用 latex 撰写）[1]

（可基于此模板，该模板所在网站是一个很流行的 latex 文档协同编辑网站，copy 此 project 即可成为自己的项目，在其上编辑即可，更多 latex 参考资料[2]）。

期望： 不要当作“命题作文”，要更多地发挥主观能动性，将其当做实验进行更多探索。如：

1. 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；
2. 调整编译器的程序选项，例如加入调试选项、优化选项等，观察输出变化、了解编译器；
3. 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。
4. 与预习作业 1 中的优化问题相结合等等。

[1] 你可以搜索“vscode+latex workshop”以配置 `latex` 环境，再进一步了解“如何用 latex 书写中文”。

[2] LaTeX 入门、LaTeX 命令与符号汇总、LaTeX 数学公式等符号书写

1.3 基础样例程序

阶乘

```
int main()
{
    int i, n, f;
    cin >> n;
    i = 2;
    f = 1;
    while (i <= n)
    {
        f = f * i;
        i = i + 1;
    }
    cout << f << endl;
}
```

斐波那契数列

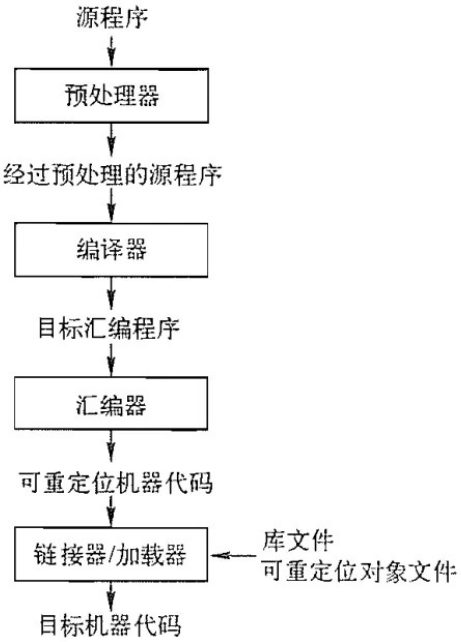
```
int main()
{
    int a, b, i, t, n;

    a = 0;
    b = 1;
    i = 1;
    cin >> n;
    cout << a << endl;
    cout << b << endl;
    while (i < n)
    {
        t = b;
        b = a + b;
        cout << b << endl;
        a = t;
        i = i + 1;
    }
}
```

2 参考流程

以下内容仅供参考，更多的细节希望同学们亲自动手体验，详细了解各阶段的作用。

以一个 C 程序为例，整体的流程如图所示：



简单来说，不同阶段的作用如下：

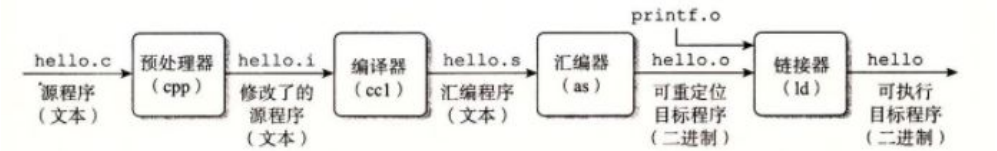
预处理器 处理源代码中以 # 开始的预编译指令，例如展开所有宏定义、插入 `#include` 指向的文件等，以获得经过预处理的源程序。

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

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

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

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



下面将详细介绍每个阶段的实验方法（源程序用 main.c 表示）。

2.1 预处理器

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

对于 gcc，**通过添加参数-*E* 令 gcc 只进行预处理过程**，参数-*o* 改变 gcc 输出文件名，因此通过命令得到预处理后文件：

```
gcc main.c -E -o main.i
```

观察预处理文件，可以发现文件长度远大于源文件，这就是将代码中的头文件进行了替代导致的结果。另外，实际上预处理过程是 gcc 调用了另一个程序（C Pre-Processor 调用时简写作 cpp）完成的过程，有兴趣的同学可以自行尝试。

2.2 编译器

编译过程是我们整门课程着重讲述的过程，具体来说分为六步，详细解释可以查看课程的预习 PPT，简单来说分别为：

词法分析 将源程序转换为单词序列。对于 LLVM，你可以通过以下命令获得 token 序列：

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

语法分析 将词法分析生成的词法单元来构建抽象语法树（Abstract Syntax Tree，即 AST）。对于 gcc，你可以通过 `-fdump-tree-original-raw` flag 获得文本格式的 AST 输出。LLVM 可以通过如下命令获得相应的 AST：

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

语义分析 使用语法树和符号表中信息来检查源程序是否与语言定义语义一致，进行类型检查等。

中间代码生成 完成上述步骤后，很多编译器会生成一个明确的低级或类机器语言的中间表示。

你可以通过`-fdump-tree-all-graph`和`-fdump-rtl-all-graph`两个 gcc flag 获得中间代码生成的多阶段的输出。生成的`.dot` 文件可以被 `graphviz` 可视化，vscode 中直接有相应插件。你可以看到控制流图（CFG），以及各阶段处理中（比如优化、向 IR 转换）CFG 的变化。你可以额外使用`-Ox`、`-fno-*` 等 flag 控制编译行为，使输出文件更可读、了解其优化行为。

LLVM 可以通过下面的命令生成 LLVM IR：

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

代码优化 进行与机器无关的代码优化步骤改进中间代码，生成更好的目标代码。

在第一周的预习作业中，很多同学对编译器如何进行代码优化感到疑问，在这个步骤中你可以通过 LLVM 现有的优化 pass 进行代码优化探索。

在 LLVM 官网对所有 pass 的分类[3]中，共分为三种：Analysis Passes、Transform Passes 和 Utility Passes。Analysis Passes 用于分析或计算某些信息，以便给其他 pass 使用，如计算支配边界、控制流

[3]LLVM 对所有 pass 的简述

图的数据流分析等；Transform Passes 都会通过某种方式对中间代码形式的程序做某种变化，如死代码删除，常量传播等。

LLVM 可以通过下面的命令生成每个 pass 后生成的 LLVM IR，以观察差别：

```
llc -print-before-all -print-after-all a.ll > a.log 2>&1
# 因为输出的内容过长，在命令行中无法完整显示，这时必须要对输出进行重定向
# 0、1、2 是三个文件描述符，分别表示标准输入 (stdin)、标准输出 (stdout)、标准错误 (stderr)
# 因此 2>&1 的具体含义就不难理解，你也可以试试去掉重定向描述，看看实际效果
```

同样，你也可以通过下面的命令指定使用某个 pass 以生成 LLVM IR，以特别观察某个 pass 的差别：

```
opt -<module name> <test.bc> /dev/null
```

所有的 module name 对应的命令行参数也可以在[4]查到。

上面的指令需要用到 bc 格式，即 LLVM IR 的二进制代码形式，而我们之前生成的是 LLVM IR 的文本形式（ll 格式）。当然我们也可以通过添加额外命令行参数的方式直接使用 ll 格式的 LLVM IR，这里留给同学们自行探究。

我们可以通过下面的命令让 bc 和 ll 这两种 LLVM IR 格式互转，以统一文件格式：

```
llvm-dis a.bc -o a.ll # bc 转换为 ll
llvm-as a.ll -o a.bc # ll 转换为 bc
```

如果你认为本部分的探索内容过于“黑盒”，你也可以尝试去阅读各大编译器，如 LLVM 各个 pass 的源码[5]。尽管你现在可能并不了解大多数 pass 实际上是怎么工作的，但可能对你大作业的代码优化部分程序的编写有帮助。

代码生成 以中间表示形式作为输入，将其映射到目标语言

```
gcc main.i -S -o main.S # 生成 x86 格式目标代码
arm-linux-gnueabihf-gcc main.i -S -o main.S # 生成 arm 格式目标代码
llc main.ll -o main.S # LLVM 生成目标代码
```

2.3 汇编器

汇编过程实际上把汇编语言程序代码翻译成目标机器指令的过程。其最终生成的是可重定位的机器代码。这一步一般被视为编译过程的“后端”，你可以在一些网上资料，比如这里，进行宏观的了解。

[4] LLVM 的 pass 参数

[5] LLVM 源码所在仓库

希望同学们在报告中详细分析并阐述汇编器处理的结果以及汇编器的具体功能分析。你可能会用到反编译工具（你可以在文后的 Makefile 中找到简单的使用示例）。

x86 格式汇编可以直接用 gcc 完成汇编器的工作，如使用下面的命令：

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

arm 格式汇编需要用到交叉编译，如使用下面的命令：

```
arm-linux-gnueabihf-gcc main.S -o main.o
```

LLVM 可以直接使用 llc 命令同时编译和汇编 LLVM bitcode：

```
llc test.bc -filetype=obj -o test.o
```

2.4 链接器加载器

由汇编程序生成的目标文件不能够直接执行。大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真正在机器上运行的代码。进而链接器对该机器代码进行执行生成可执行文件。可以尝试对可执行文件反汇编，看一看与上一阶段反汇编结果的不同。

在这一阶段，你可以尝试调整链接相关参数，如`-static`。

```
gcc main.o -o main
```

当你执行可执行文件时，便会使用到加载器，以将二进制文件载入内存。这不是我们要研究的范围了。

3 LLVM IR 编程

3.1 LLVM IR 概述

LLVM 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，你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。LLVM IR 是相对于 CPU 指令集更为高级、但相对于源程序更为低级的代码中间表示的一种语言。从上述介绍中可以看出 LLVM 后端支持相当多的平台，我们无须担心操作系统等平台的问题，而且我们只需将代码编译成 LLVM IR，就可以由优化水平较高的 LLVM 后端来进行优化。此外，LLVM IR 本身更贴近汇编语言，指令集相对底层，能灵活地进行低级操作。

LLVM IR 代码存在三种表示形式：在内存中的表示（BasicBlock、Instruction 等 cpp 类）、二进制代码形式（用于编译器加载）、可读的汇编语言表示形式。除了上面提到的`clang -S -emit-llvm main.c`，你也可以通过`clang -c -emit-llvm main.c -o mian.bc`生成 bitcode 形式的 LLVM IR 文件。

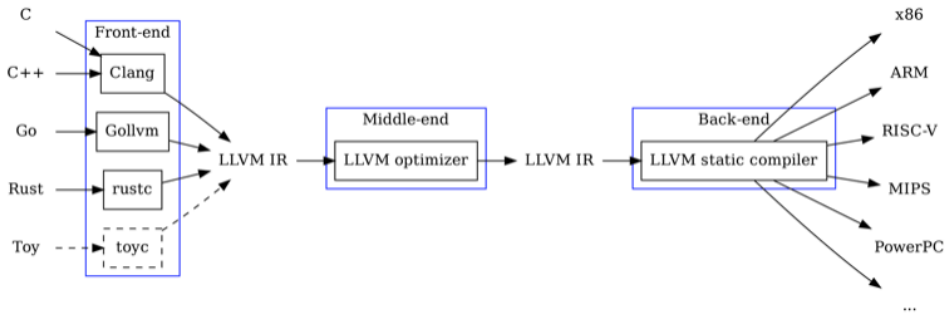


图 3.1: LLVM IR 设计架构，可参考。

3.2 实验案例

下面以一个基础样例程序（示例给的**阶乘**程序）为例对 LLVM IR 特性进行简单介绍。更多有关 LLVM IR 的结构问题，可以参考LLVM Programmer Manual。

首先你需要在命令行中输入`clang -emit-llvm -S main.c -o main.ll`,打开同目录下的 `main.ll` 文件，你可以得到以下内容（本指导书已删除无用语句，加入 llvm IR 相关注释及其与 SysY 语言特性的对应关系）：

```
; 所有的全局变量都以 @ 为前缀，后面的 global 关键字表明了它是一个全局变量
; SysY 语言中注释的规范与 C 语言一致
; 函数定义以 `define` 开头，i32 标明了函数的返回类型，其中 `main`是函数的名字，`@` 是其前缀
; FuncDef ::= FuncType IDENT "(" [FuncFParams] ")" Block; FuncDef 表示函数定义，FuncType 指明了函数的返回类型，FuncParam 是函数定义的形参列表
define i32 @main() #0 {
  ; 以 % 开头的符号表示虚拟寄存器，你可以把它当作一个临时变量（与全局变量相区分），或称之为临时寄存器
  %1 = alloca i32, align 4
  ; 为 %1 分配空间，其大小与一个 i32 类型的大小相同。%1 类型即为 i32*，align 4 可以理解为对齐方式为 4 个字节
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  ; 将 0 (i32) 存入 %1 (i32*)
  store i32 0, i32* %1, align 4
  ; 调用函数 @scanf ， i32 表示函数的返回值类型
  %5 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* %3)
  store i32 2, i32* %2, align 4
  store i32 1, i32* %4, align 4
  ; 这里的 br 是无条件分支，label 可以理解为一个代码标签，指代下面那个代码块
  br label %6
```

```
20
21  6:                                                ; preds = %10, %0
22    %7 = load i32, i32* %2, align 4
23    %8 = load i32, i32* %3, align 4
24    ; icmp 会根据不同的比较规则（这里是 sle，小于等于）比较两个操作数%7 和%8，i32 是操作
    ↪ 数类型
25    %9 = icmp sle i32 %7, %8
26    ; 这里的 br 是有条件分支，它根据 i1 和两个 label 的值，用于将控制流传输到当前函数中
    ↪ 的不同基本块。
27    ; i1 类型的变量%cmp 的值如果为真，那么执行 label%10，否则执行 label%16
28    br i1 %9, label %10, label %16
29
30  10:                                               ; preds = %6
31    %11 = load i32, i32* %4, align 4
32    %12 = load i32, i32* %2, align 4
33    %13 = mul nsw i32 %11, %12
34    store i32 %13, i32* %4, align 4
35    %14 = load i32, i32* %2, align 4
36    %15 = add nsw i32 %14, 1
37    store i32 %15, i32* %2, align 4
38    br label %6, !llvm.loop !10
39
40  16:                                               ; preds = %6
41    %17 = load i32, i32* %4, align 4
42    %18 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
      ↪ i8]* @.str.1, i64 0, i64 0), i32 %17)
43    ret i32 0
44  }
45  ; 函数声明
46  declare dso_local i32 @__isoc99_scanf(i8*, ...)
47
48  declare dso_local i32 @printf(i8*, ...)
49
```

根据上述.ll 文件，我们对 LLVM IR 及 SysY 特性做以下总结：

1. LLVM IR 的基本单位称为 module（只要是单文件编译就只涉及单 module），对应 SysY 中的 CompUnit——CompUnit ::= [CompUnit] (Decl | FuncDef)，一个 CompUnit 中有且仅有一个 main 函数定义，是程序的入口。

2. 一个 module 中可以包含多个顶层实体，如 function 和 global variable，CompUnit 的顶层变量/常量声明语句（对应 Decl），函数定义（对应 FuncDef）都不可以重复定义同名标识符（IDENT），即便标识符的类型不同也不允许。

3. 一个 `function define` 中至少有一个 `basicblock`。`basicblock` 对应 SysY 中的 `Block` 语句块，语句块内声明的变量的生存期在该语句块内。`Block` 表示为
   Block ::= “{” BlockItem “}”;
   BlockItem ::= Decl | Stmt;

4. 每个 `basicblock` 中有若干 `instruction`，且都以 `terminator instruction` 结尾。SysY 中语句表示为
   Stmt ::= LVal “=” Exp ”;“
   | [Exp] “;”
   | Block
   | “if” “(” Exp “)” Stmt [“else” Stmt]
   | “while” “(” Exp “)” Stmt
   | “break” “;”
   | “continue” “;”
   | “return” [Exp] “;”;

5. llvm IR 中注释以 “`;`” 开头，而 SysY 中与 C 语言一致。

6. llvm IR 是静态类型的，即每个值的类型在编写时是确定的。

7. llvm IR 中全局变量和函数都以 `@` 开头，且会在类型（如 i32）之前用 `global` 标明，局部变量以`%` 开头，其作用域是单个函数，临时寄存器（上文中的%1 等）以升序阿拉伯数字命名。

8. 函数定义的语法可以总结为：define + 返回值 (i32) + 函数名 (@main) + 参数列表 ((i32 %a,i32 %b)) + 函数体 (ret i32 0)，函数声明你可以在 `main.ll` 的最后看到，即用 `declare` 替换 `define`。SysY 中函数定义表示为`FuncDef ::= FuncType IDENT "(" [FuncFParams] ")" Block`。

9. 终结指令一定位于一个基本块的末尾，如 `ret` 指令会令程序控制流返回到函数调用者，`br` 指令会根据后续标识符的结果进行下一个基本块的跳转，`br` 指令包含无条件（`br+label`）和有条件（`br+` 标志符 `+truelabel+falselabel`）两种。

10. `i32` 这个变量类型实际上就指 32 bit 长的 integer，类似的还有 void、label、array、pointer 等。

11. 绝大多数指令的含义就是其字面意思，load 从内存读值，store 向内存写值，add 相加参数，alloca 分配内存并返回地址等。

关于 LLVM IR 可以从官方文档进行更多了解，一些常用的指令[6]。有关 SysY 语言更多内容可以参考官方文档[7][8]。

3.3 LLVM IR 的 OpaquePointers 介绍

由于我们 RISC-V 架构编译器使用的 clang 版本为 15.0 以上，该版本的 clang 的 LLVMIR 支持 OpaquePointers 特性，该特性可以简化我们编译器实现数组和指针的难度，同时也可以降低我们编写

[6] LLVM IR 常用指令
[7] SysY 语言定义
[8] SysY 运行时库

优化的难度，尤其是 mem2reg(同时也是我们本学期作业代码优化环节进阶要求必须实现的优化)，指针分析，内存操作的公共子表达式提取等优化 Pass。

在原先的 LLVM IR 中，指针类型需要包含指向的类型，例如 i32* 表示一个指向 32 位整型的指针。[3 x [4 x float]]* 表示一个指向 [3 x [4 x float]] 数组类型的指针。但是在新版本的 LLVM 中，指针不再需要指定类型，统一使用 ptr 表示。

这时候你也许会问，统一用 ptr 类型表示，那 load 的时候怎么确定要读取多少字节呢。实际上，新版本的 LLVM 中只需要在 load，getelementptr 等涉及到内存操作的指令时，在前面加上类型，表示读取多少字节。下面是一个例子:

```
int f(float A[][3][4])
{
    return A[2][1][1];
}
```

旧版本对应的 LLVMIR 为:

```
define i32 @f(float (*) [3][4])([3 x [4 x float]]*  %0) {
  %2 = getelementptr inbounds [3 x [4 x float]], [3 x [4 x float]]* %0, i64 2, i64
↪ 1, i64 1
  %3 = load float, float* %2, align 4
  %4 = fptosi float %3 to i32
  ret i32 %4
}
```

新版本对应的 LLVMIR 为:

```
define i32 @f(ptr %0) {
  %2 = getelementptr inbounds [3 x [4 x float]], ptr %0, i64 2, i64 1, i64 1
  %3 = load float, ptr %2, align 4
  %4 = fptosi float %3 to i32
  ret i32 %4
}
```

你可以在链接LLVM OpaquePointers 文档中了解到关于 OpaquePointers 的更多信息。同时如果你选择 Cpp 语言编写的 RISC-V 架构框架作为你的实验大作业，你可以在 LLVM IR 编程作业中就尝试使用这一特性。

3.4 样例 Makefile 文件

```
.PHONY: pre, lexer, ast-gcc, ast-llvm, cfg, ir-gcc, ir-llvm, asm, obj, exe, antiobj,
antiexe
```

```
3
4  pre:
5      gcc main.c -E -o main.i
6
7  lexer:
8      clang -E -Xclang -dump-tokens main.c
9
10 # 生成`main.c.003t.original`
11 ast-gcc:
12     gcc -fdump-tree-original-raw main.c
13
14 # 生成`main.ll`
15 ast-llvm:
16     clang -E -Xclang -ast-dump main.c
17
18 # 会生成多个阶段的文件 (.dot)，可以被 graphviz 可视化，可以直接使用 vscode 插件
19 # (Graphviz (dot) language support for Visual Studio Code)。
20 # 此时的可读性还很强。`main.c.011t.cfg.dot`
21 cfg:
22     gcc -O0 -fdump-tree-all-graph main.c
23
24 # 此时可读性不好，简要了解各阶段更迭过程即可。
25 ir-gcc:
26     gcc -O0 -fdump-rtl-all-graph main.c
27
28 ir-llvm:
29     clang -S -emit-llvm main.c
30
31 asm:
32     gcc -O0 -o main.S -S -masm=att main.i
33
34 obj:
35     gcc -O0 -c -o main.o main.S
36
37 antiobj:
38     objdump -d main.o > main-anti-obj.S
39     nm main.o > main-nm-obj.txt
40
41 exe:
42     gcc -O0 -o main main.o
43
44 antiexe:
```

```
45      objdump -d main > main-anti-exe.S
46      nm main > main-nm-exe.txt
47
48  clean:
49      rm -rf *.c.*
50
51  clean-all:
52      rm -rf *.c.* *.o *.S *.dot *.out *.txt *.ll *.i main
```

4 汇编编程

我们用下面 C 代码为例来介绍 arm 和 RISC-V 汇编编程。

```c
#include<stdio.h>

int a = 0;
int b = 0;

int max(int a, int b) {
    if(a >= b) {
        return a;
    } else {
        return b;
    }
}

int main() {
    scanf("%d %d", &a, &b);
    printf("max is: %d\n", max(a, b));
    return 0;
}
```

4.1 ARM 汇编编程

你可以在这里，或者这里（简易版）获得你需要了解的 arm 架构的知识。它们可能包括，区分 arm 与 thumb 模式（我们应不会使用 thumb 模式）、理解 arm 架构各寄存器（与各状态位）的含义、了解要使用的指令集、理解函数栈是如何增长的，和一些必要的汇编代码编写技巧。

上一任笔者估计，若您对 arm 完全不了解，那么大概需要 3 小时的专注时间以理解你需要的全部知识。

代码示例

```
1         .arch armv7-a
2         @ comm section   save global variable without initialization
3         .comm    a, 4    @global variables
4         .comm    b, 4
5         .text
6         .align    2
7         @ rodata section  save constant
8         .section    .rodata
9         .align    2
10    _str0:
11        .ascii    "%d %d\0" @\000 is also one representation for `null character`
12        .align    2
13    _str1:
14        .ascii    "max is: %d\n"
15        @ text section   code
16        .text
17        .align    2
18
19        .global    max
20    max:  @ function  int max(int a, int b)
21        str    fp, [sp, #-4]!   @ pre-index mode, sp = sp - 4, push fp
22        mov    fp, sp
23        sub    sp, sp, #12      @ allocate space for local variable
24        str    r0, [fp, #-8]    @ r0 = [fp, #-8] = a
25        str    r1, [fp, #-12]   @ r1 = [fp, #-12] = b
26        cmp    r0, r1
27        blt    .L2
28        ldr    r0, [fp, #-8]
29        b    .L3
30    .L2:
31        ldr    r0, [fp, #-12]
32    .L3:
33        add    sp, fp, #0
34        ldr    fp, [sp], #4     @ post-index mode, pop fp, sp = sp + 4
35        bx    lr                @ recover sp fp pc
36        @ do you know the difference between `bx` and `bl`?
37        @ and if max function is non-leaf, what should we do with the `lr` register?
38        .global    main
39    main:
40        push    {fp, lr}
41        add    fp, sp, #4
42        ldr    r2, _bridge      @ r2 = &b
43        ldr    r1, _bridge+4    @ r1 = &a
44        ldr    r0, _bridge+8    @ *r0 = "%d %d\000"
45        bl    __isoc99_scanf    @ scanf("%d %d", &a, &b)
46        ldr    r3, _bridge+4    @ r3 = &a
47        ldr    r0, [r3]         @ r0 = a
48        ldr    r3, _bridge      @ r3 = &b
49        ldr    r1, [r3]         @ r1 = b
50        bl    max
51        mov    r1, r0           @ r1 = r0
52        ldr    r0, _bridge+12   @ *r0 = "max is: %d\0"
53        bl    printf            @ printf("max is: %d", max(a, b));
54        mov    r0, #0
55        pop    {fp, pc}         @ return 0
56
57    _bridge:
58        .word    b
59        .word    a
60        .word    _str0
61        .word    _str1
62
```

```
63          .section    .note.GNU-stack,"",%progbits  @ do you know what's the use of this :-)
```

代码说明

这其中有一系列的指令是编译器指令，作用是告知编译器要如何编译，通常以. 开始，其他指令则为汇编指令。

对每个函数的声明，观察可以发现一般首先为

```
.text
.global functionname
.type functionname, %function
```

即声明为代码段，将函数名添加到全局符号表中，声明类型为函数。

对全局变量与常量的声明，示例中已经给得比较详细，另外对于数组的使用，可以看到在声明时是毫无特殊的，而在使用时地址则为 *varname+offset*，其中偏移量即为数据类型大小乘个数。

另外值得说明的是，在进行函数调用时，一般前四个函数参数使用 r0-r3 号寄存器进行传参，其余参数压入栈中进行传参，往往按照从右至左的顺序逐个压栈；在函数返回时，一般来讲默认将函数的返回值放到 r0 寄存器中。

我们可以发现汇编与 C 的不同：汇编的语言要素就是“标签”（指示地址）、寄存器移动/计算指令。尤其标签的灵活使用：上述汇编代码中利用 `_bridge` 标签，“桥接”了在 C 代码中隐性的全局变量的地址。

某前辈曾说，编程语言理论，建立在“组合”之上——组合意味着复用，意味着抽象。在理解汇编代码时，我们希望将“理解其抽象、其作为整体的语义”作为思考目标（操作系统课或许会接触一些乍一看难理解汇编代码）；在编写程序时，也常常是自顶向下的思维过程。

代码逐行解析

1. 定义目标架构：`.arch armv7-a` 表示使用 ARMv7-a 指令集。
2. 定义数据区：`.comm a, 4` 定义了一个全局变量 a，大小为 4 字节。`.comm b, 4` 定义了一个全局变量 b，大小为 4 字节。
3. 定义文本区：`.text` 表示接下来是代码区。
4. 对 rodata 数据区进行对齐：`.align 2` 表示下一个数据项的地址应该是对齐的，且偏移量应为 2 的 2 次方字节。建议使用`.p2align 2`来表示。如果对齐的目标不正确，代码运行的速度会降低。但是需要注意，在栈中需要严格对齐，不然会存在各种各样的错误。
5. 定义 rodata 数据区：`.section .rodata` 表示接下来是 rodata 数据区。
6. 定义常量字符串：`.ascii "%d %d\0"` 定义了一个包含两个整数格式化字符串的常量字符串，用于在后续的`scanf`和`printf`函数中使用。
7. 定义文本区：`.text` 表示接下来是代码区。
8. 定义全局变量 max：`.global max` 表示将变量 max 声明为全局变量。
9. max 函数实现：

- 保存寄存器：`str fp, [sp, #-4]!` 表示将寄存器 fp 压入栈中，同时更新 sp 指针。
- 设置寄存器：`mov fp, sp` 表示将 sp 指针设置为当前栈指针。
- 分配栈空间：`sub sp, sp, #12` 表示为局部变量分配 12 字节的栈空间。
- 保存参数：`str r0, [fp, #-8]` 表示将参数 a 压入栈中，偏移量为-8。
- 保存参数：`str r1, [fp, #-12]` 表示将参数 b 压入栈中，偏移量为-12。
- 比较参数：`cmp r0, r1` 表示比较参数 a 和 b 的大小。
- 分支：`blt .L2` 表示如果 a 小于 b，则跳转到标签.L2。
- 返回最大值：`ldr r0, [fp, #-8]` 表示将参数 a 从栈中弹出，并赋值给寄存器 r0。
- 分支：`b .L3` 表示如果 a 大于等于 b，则跳转到标签.L3。
- 返回最大值：`ldr r0, [fp, #-12]` 表示将参数 b 从栈中弹出，并赋值给寄存器 r0。
- 恢复栈空间：`add sp, fp, #0` 表示将栈指针恢复到原始值。
- 恢复寄存器：`ldr fp, [sp], #4` 表示从栈中弹出 fp 寄存器，并更新 sp 指针。
- 返回：`bx lr` 表示恢复 pc 指针，并返回。

10. main 函数实现：

- 保存寄存器：`push {fp, lr}` 表示将寄存器 fp 和 lr 压入栈中，并更新 sp 指针。
- 定义局部变量：`add fp, sp, #4` 表示为局部变量分配 4 字节的栈空间。
- 加载常量字符串：`ldr r2, _bridge` 表示将常量 b 的地址赋值给寄存器 r2。
- 加载常量字符串：`ldr r1, _bridge+4` 表示将常量 a 的地址赋值给寄存器 r1。
- 加载常量字符串：`ldr r0, _bridge+8` 表示将常量字符串 _str0 的地址赋值给寄存器 r0。
- 调用函数：`bl __isoc99_scanf` 表示调用`scanf`函数读取两个整数参数 a 和 b。
- 加载参数 a：`ldr r3, _bridge+4` 表示将常量 a 的地址赋值给寄存器 r3。
- 加载参数 b：`ldr r1, [r3]` 表示将 r3 地址处的值（即 a 的值）赋值给寄存器 r1。
- 调用函数：`bl max` 表示调用`max`函数计算 a 和 b（分别存在 r0，r1）的最大值，返回结果在 r0（默认返回 r0）。
- 保存结果：`mov r1, r0` 表示将结果赋值给寄存器 r1。
- 加载常量字符串：`ldr r0, _bridge+12` 表示将常量字符串地址 _str1 赋值给寄存器 r0。
- 调用函数：`bl printf` 表示调用`printf`函数输出，r0 为字符串参数，r1 为 max 结果，两者作为函数调用的参数。
- 返回 0：`mov r0, #0` 表示将 0 赋值给寄存器 r0，作为 main 函数的返回值。
- 恢复寄存器：`pop {fp, pc}` 表示将 fp 寄存器弹出并恢复，并更新 pc 指针。

11. 定义`_bridge`的符号表项

- `.word b`：这是一个引用，表示符号`b`的 32 位 (.word) 地址。
- `.word a`：表示符号`a`的地址。
- `.word _str0`：表示符号`_str0`的地址。

- `.word _str1`：表示符号_str1的地址。
- `.section .note.GNU-stack,"",%progbits`：这一行代码是用于 GNU Assembler（GAS）的，通常出现在汇编语言源文件中。它用于指定一个特殊的节（section）`.note.GNU-stack`。这个节没有实际的代码或数据，而是用于向链接器和操作系统传递有关程序堆栈的信息。具体来说，`.note.GNU-stack`用于控制生成的可执行文件的堆栈是否是可执行的。在默认情况下，Linux 操作系统通常会将堆栈标记为不可执行，这有助于阻止堆栈执行攻击。这是一个安全性措施，旨在防止攻击者通过注入可执行代码到程序堆栈来攻击程序。

代码测试 当你写完汇编程序（比如 `example.S`）后，使用下述指令即可测试它。

```
arm-linux-gnueabihf-gcc example.S -o example.out
qemu-arm -L /usr/arm-linux-gnueabihf ./example.out
```

当然，你可以让测试过程更“自动化”些，将它加入 Makefile，并利用管道测试默认样例、生成结果。[9]。

```
.PHONY:test,clean
test:
    arm-linux-gnueabihf-gcc example.S -o example.out
    qemu-arm -L /usr/arm-linux-gnueabihf ./example.out
clean:
    rm -fr example.out
```

4.2 RISC-V 汇编编程

为了便于对比学习，我们仍然以 ARM 汇编编程当中的 C 代码为例来介绍 Risc-V 汇编编程。你需要注意我们本学期实验中，RISC-V 的编译器框架使用的均为 64 位 RISC-V 指令集，不要弄错看成 32 位的了 (不过 32 位和 64 位差别很小，你可以通过先学习 32 位再转 64 位的方式来学习)。你可以通过查阅RISC-V 指令手册来获取完整的 RISC-V 指令集的知识。还有一个中文版本的指令手册，RISC-V 指令手册中文版本。你还可以自行使用搜索引擎搜索或者询问 chat-gpt 来了解快速掌握 RISC-V 指令集的方法。

在本学期的所有实验中，我们只需要了解 RISC-V 的基础指令集，以及 M 扩展 (乘除法)，F 和 D 扩展 (浮点数指令集) 即可，所以你只需要阅读手册的一部分即可。

代码示例

```
    .file   "test.c"
    .option nopic
    .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
    .attribute unaligned_access, 0
    .attribute stack_align, 16
    .text
    .align  1
    .globl  max
    .type   max, @function
max:
    mv  a5,a0
```

[9]若您还没有配置好 arm 环境，请参考《编译器开发环境》实验指导。

```
    bge a0,a1,.L2
    mv  a5,a1
.L2:
    sext.w  a0,a5
    ret
    .size   max, .-max
    .section    .rodata.str1.8,"aMS",@progbits,1
    .align  3
.LC0:
    .string "%d %d"
    .align  3
.LC1:
    .string "max is: %d\n"
    .section    .text.startup,"ax",@progbits
    .align  1
    .globl  main
    .type   main, @function
main:
    addi    sp,sp,-32
    sd  s0,16(sp)
    sd  s1,8(sp)
    lui s0,%hi(a)
    lui s1,%hi(b)
    lui a0,%hi(.LC0)
    addi    a2,s1,%lo(b)
    addi    a1,s0,%lo(a)
    addi    a0,a0,%lo(.LC0)
    sd  ra,24(sp)
    call    __isoc99_scanf
    lw  a1,%lo(b)(s1)
    lw  a0,%lo(a)(s0)
    call    max
    mv  a1,a0
    lui a0,%hi(.LC1)
    addi    a0,a0,%lo(.LC1)
    call    printf
    ld  ra,24(sp)
    ld  s0,16(sp)
    ld  s1,8(sp)
    li  a0,0
    addi    sp,sp,32
    jr  ra
    .size   main, .-main
    .globl  b
    .globl  a
    .section    .sbss,"aw",@nobits
    .align  2
    .type   b, @object
    .size   b, 4
b:
    .zero   4
    .type   a, @object
    .size   a, 4
a:
    .zero   4
    .ident  "GCC: () 12.2.0"
    .section    .note.GNU-stack,"",@progbits
```

下面对一些关键代码进行解释

1. `.option nopic` 表示不使用位置无关的代码。此时，汇编器生成的代码将假定它会被加载到固定

的内存地址，对于生成静态链接的二进制文件比较重要，有时可以提高代码的执行效率。

2. `.attribute arch, "rv64i2p1_m2p0"`是用于指定汇编代码所遵循的架构特性和扩展的指令。

   `rv64i2p1` 表示代码遵循 RISC-V 64 位基础证书指令集，版本为 2.1.

   `m2p0`表示支持乘法和除法指令，版本为 2.0. 其余的指令大家可以通过搜索均可查到相关意义，此处不再赘述。

3. “`.attribute stack_align, 16`” 表示栈空间需要 16 字节对齐，在我们 RISC-V 版本的编译实验作业中，同样需要遵循这一规定.

4. “`.text`” 表示接下来是代码区

5. 接下来我们介绍 max 函数中出现的汇编指令

   首先我们需要了解一点，RISC-V 函数调用约定中参数的传递首先使用 a0-a7 寄存器，如果寄存器不够用，则使用栈进行传递。所以在函数入口处，a0 存放着变量 a 的值，a1 存放着变量 b 的值。

   `"mv a5 a0"`的含义为 a5 = a0，现在 a5 中也存放着变量 a 的值

   `"bge a0,a1,.L2"`的含义为如果 a0 ≥ a1，则跳转至.L2，对应源代码的 if(a>=b)

   `"mv a5 a1"`的含义为如果 a5 = a1, 这一条语句在 a>=b 为假的分支执行，表示将最大值 a1(存放着变量为 b 的值，此时有 b > a) 赋值给 a5。

   `"sext.w a0,a5"`表示将 32 位的 a5 符号扩展到 64 位，值存放到 a0 中。

   `"ret"`表示函数返回，即源代码中的 return。根据 RISC-V 函数调用约定，使用 a0 寄存器存放函数的返回值。

   max 函数最后的.size 和.section 在我们的编译实验作业中并不需要生成，大家如果感兴趣可以自行查阅资料了解含义。`.align 3` 表示之前有一段未对齐的代码或数据，汇编器会在这段内容前插入适当数量的填充字节，以确保接下来的数据或代码从一个 8 字节对齐的地址开始。这样做可以提高内存访问效率和程序的运行性能。

6. “`.LC0`和`.LC1`” 这些标签定义了字符串常量

7. 接下来介绍 main 函数中出现的汇编指令

   `"addi sp,sp,-32"`首先我们需要知道 sp 寄存器指向栈顶，该条指令的含义为开辟 32 字节的栈空间

   `"sd s0,16(sp)"`和`sd s1,8(sp)`首先我们需要知道 RISC-V 的函数调用约定中，被调用者需要保存 sp，s0-s11 寄存器，该函数中使用到了 s0 和 s1 寄存器，所以我们在函数开头需要进行保存。sd s0,16(sp) 表示将 s0 寄存器存储到地址为 16+sp 的内存中。

   `"lui s0,%hi(a)"`表示将 a 的标签 (即全局变量 a 的地址) 的高 20 位放到 s0 寄存器中，后两条指令同理。

   `"addu a2,s1,%lo(b)"`表示 a2 = s1 + b 的标签 (即全局变量 b 的地址) 的低 12 位，后两条指令同理，现在 a2 寄存器中存放着全局变量 b 的地址，同理，a1 寄存器中存放着全局变量 a 的地址，a0 寄存器中存放着字符串常量的地址。

"sd ra,24(sp)"表示保存 ra 寄存器的值，原因是 RISC-V 函数调用约定需要调用者保存 ra 寄存器的值。

“call __isoc99_scanf” 表示 scanf 函数的调用，此时我们回忆一下刚才 a0，a1，a2 寄存器中变量的含义以及 RISC-V 函数调用约定，你大概就明白了此时 call 的含义。(call 指令实际上是一条伪指令，具体做了什么就交给同学们自己探索了)

"lw a1,%lo(b)(s1)"含义为读取内存地址为 s1 + %lo(b) 的值 (实际上，即为全局变量 b 的地址)，并放到 a1 寄存器中，现在 a1 寄存器中为全局变量 b 的值。下一条指令同理，a0 寄存器中为全局变量 a 的值。

"call max"表示调用 max 函数，此时推荐再回忆一下 a0，a1 寄存器分别表示什么。

“mv a1,a0” 在函数调用完毕后，a0 存放着函数返回值，此时我们将 a0 赋值给 a1，即 a1 保存着 max(a,b) 的结果，也许你会问为什么这里要多此一举将 a0 赋值给 a1，当你在下面看到 printf 函数的调用时你会明白一切 (这也是 gcc O3 优化效果的体现)。

"lui a0, %hi(.LC1)和addi a0,a0,%lo(.LC1)"的含义为将.LC1 中字符串常量的地址赋值给 a0。

"call printf"表示调用 printf 函数，不妨再回忆一下此时 a0，a1 寄存器的含义。

"ld ra 24(sp)"表示恢复之前保存的 ra 寄存器。

"ld s0,16(sp)和ld s1,8(sp)"表示将 s0 和 s1 寄存器恢复至调用前的状态。

"li a0, 0"表示将 a0 赋值为 0，表示 main 函数的返回值。

"addi sp,sp,32"表示恢复栈空间

"jr ra"表示函数返回，指令含义为跳转到 ra 寄存器中的地址 (ra 寄存器会保存函数的返回地址，此时即使你不知道 call 这一条伪指令的含义，应该也能猜到 call 做了什么了)

8. 最后还有一些全局变量的定义，从文本上很容易理解其含义，这里就不赘述了。

9. 上述的 RISC-V 汇编由 GCC O3 优化选项编译而成。如果你想了解编译器不做任何优化生成的汇编是什么样的，可以自己使用在环境配置环节安装的 riscv 交叉编译 GCC，并使用编译选项 O0 进行编译。(添加编译选项-S 表示生成汇编代码)

对编写的 RISC-V 汇编进行测试 假设你编写了一个 RISC-V 汇编文件，命名为 test.s，使用下面的命令即可进行测试。

```
riscv64-unknown-linux-gnu-gcc test.s -o test -static
# 如果你好奇为什么要加-static，可以取消-static 后使用 qemu 运行试试，看看会报什么错误
# 你可以再根据错误信息去搜索引擎上搜索或者询问 chat-gpt 来了解原因。
qemu-riscv64 test
```