

# 编译器开发环境部署

杨俣哲 李煦阳

杨科迪 费迪

周辰霏

谢子涵 杨科迪

李君龙 华志远 李帅东

唐显达 鲁恒泽 卢艺晗

2020 年 9 月—2025 年 9 月

# 目录

<b>1 工具简要介绍</b>	<b>3</b>
1.1 Lex & Flex . . . . .	3
1.2 Yacc & Bison . . . . .	3
1.3 工程辅助工具 . . . . .	3
<b>2 实验环境</b>	<b>4</b>
2.1 架构的选择 . . . . .	4
2.2 系统环境的安装 . . . . .	4
2.2.1 Windows . . . . .	4
2.2.2 Mac . . . . .	5
<b>3 工具安装</b>	<b>5</b>
3.1 基本工具 . . . . .	5
3.2 词法分析和语法分析辅助工具 . . . . .	6
3.3 工具链 . . . . .	6
3.3.1 arm 工具链 . . . . .	6
3.3.2 Risc-V 工具链 . . . . .	6
<b>4 环境测试</b>	<b>8</b>
4.1 测试 GCC/make/Flex/Bison . . . . .	8
4.2 熟悉 arm 工具链 . . . . .	8
4.3 熟悉并测试 Risc-V 工具链 . . . . .	8

# 1 工具简要介绍

在课程中我们要求每组两位同学协作实现一个简单的编译器，为此我们需要一些工具。根据预习内容，我们知道编译过程分为词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成这六个阶段，字符流被解析为 token 流、形成一棵语法树，最终生成某目标汇编文件。在实现我们的编译器过程中，我们将利用 Lex（词法分析器生成器）/ Yacc（语法分析器生成器）辅助完成词法分析与语法分析，最后用相应汇编器检验生成的汇编程序。本篇指导对这部分工具的安装与使用做简要说明。

此外，这次的编译课程作业或许是同学们第一个较大的 C/C++ 工程作业，我们也希望同学能充分利用辅助工具，提高开发调试效率。<sup>1</sup>

## 1.1 Lex & Flex

在 Wiki 中对于 Lex 的描述如下：

在计算机科学里面，Lex 是一个生成词法分析器 (lexical analyzer, 也称为“扫描器” (scanners) 或者 “lexers”) 的程序。Lex 常常与 Yacc 语法分析器生成程序 (parser generator) 一起使用。Lex (最早是埃里克 · 施密特和迈克 · 莱斯克于 1975 年编写并发布) 是许多 UNIX 系统上的标准词法分析器 (lexical analyzer) 生成程序，POSIX 标准中指定了一个等效工具。。

Lex 读入一个指明词法分析器规则的输入流，然后输出用 C 语言编写的词法分析器源代码。

<sup>2</sup>

简单来说，Lex 是用来辅助生成词法分析程序的，它可以将大量重复性的工作自动计算，通过相对简单的代码可以生成词法分析程序。

而 Flex 则是 Lex 的继承者，在 Linux 中被更广泛的使用，也更容易获得。

## 1.2 Yacc & Bison

同样在 Wiki 中：

Yacc (Yet Another Compiler Compiler)，是 Unix/Linux 上一个用来生成编译器的编译器 (编译器代码生成器)<sup>3</sup>。Yacc 生成的编译器主要是用 C 语言写成的语法分析器 (parser)，一般需要与 Lex 生成的词法分析器一起使用，两部分生成的 C 程序一并编译。Yacc 本来只在 (类) Unix 系统上才有，但目前已普遍移植到 Windows 及其他平台。

而 Bison 是 Yacc 的 GNU 扩展/实现，同样作为自由软件而较容易获得与使用。

## 1.3 工程辅助工具

`gcc` 与 `llvm/clang` `gcc` 和 `llvm` 是我们使用的主要编译工具。不仅用于编译我们的编译器，也用于将我们编译器生成的汇编代码编译成可执行文件。

`clang` 是 `llvm` 编译器工具集的前端，将源代码转换为抽象语法树 (AST)，由后端使用 `llvm` 编译成平台相关机器代码。

---

<sup>1</sup> 我们不希望看见大家在 shell 中手动敲入好长好长的指令:-)。

<sup>2</sup> 这个 C 程序可以将输入字符串流按其词法分析规则分析为 token 流，每个 token 有它的类型、值、和它的字符串内容本身。

<sup>3</sup> 由此可知，语法分析是编译过程中很核心的部分。

**git** 项目往往需要进行版本控制，比如进行代码回溯、分支管理，**git** 便是最常使用的工具。你可以[在这里](#)对它了解更多。同时为了避免实验环境的损坏等原因导致的意外，希望同学们将代码托管至云上（如[github](#)）；此外，commit 记录也是独立完成作业的一种证明。

**make** 一个大型项目往往有着复杂的文件依赖关系，编译也是一个耗时的工作。**make** 便是用于辅助管理复杂的编译的依赖，并指定编译指令的程序。当然，为了程序员的方便，它引入了更多特性如伪文件，方便项目的管理与测试。你可以在[这个网站](#)或[这个网站](#)上了解更多。

**qemu** 为了能够提供一个标准的运行环境，在 Linux 系统下我们往往使用 **qemu**，这是一款开源的硬件模拟器，支持多种架构。**qemu** 支持多种模式，我们的课程和操作系统的课程中主要用到的是 User mode 和 System mode，在 User mode 即用户模式下 **qemu** 会运行针对不同指令编译的单个 Linux 程序，系统调用与 32/64 位接口适应。在这种模式下，**qemu** 能够运行不同架构下的可执行文件，功能类似于一个中间层，接受程序的指令并翻译为当前系统能够识别的系统调用等。而在 System mode 即系统模式下，**qemu** 会模拟一个完整的计算机系统，包括外围设备等，可以理解为抽象出了一个完整计算机系统。

## 2 实验环境

1. 我们将在 Linux 系统环境中完成试验。如 Windows 平台的 WSL、docker、双系统、虚拟机等。对于机器不支持 WSL 的同学，也可使用 cygwin 完成实验。
2. 我们要求目标汇编代码为 64 位 arm 格式或 64 位 RISC-V 格式，并可用 GCC assembler 生成可执行文件。

在实验环境出现问题时，大家需要对几项信息保持敏感：操作系统、环境变量、程序版本、程序位数、目标平台，某些 Linux 常用指令可以帮助你快速获取相关信息。<sup>4</sup>

### 2.1 架构的选择

arm 架构与 x86 架构分别是典型的 RISC 指令集和 CISC 指令集。x86 相信同学们都有了解，因为大家的机器大多是 x86 架构（笑）。对于 arm，你可以在[这里](#)学到你需要的全部内容；你可以部分地将它与同为 RISC 指令集的 MIPS 类比。

在本学期的实验中，我们选择 arm 架构或 RISC-V 架构来进行相关实验。目前我们常用的笔记本、PC 等基本都是 x86 架构的硬件，想要运行其它架构的可执行文件往往需要通过额外的中间层，同学们需要在 Linux 系统上安装 Qemu 来运行程序。

### 2.2 系统环境的安装

#### 2.2.1 Windows

**WSL 的安装** WSL 是 Windows Subsystem for Linux 的简写，也是在 Windows 上最推荐的实验环境。它允许开发者在 Windows 上方便的使用 Linux 程序，避免了安装运行虚拟机的复杂性。

WSL 对于系统有一定需求，你可以在[这里](#)找到相应说明。你可以通过升级操作系统以满足需求；但若你的机器是 32 位的，则无法使用 WSL。WSL 的安装说明可以在[这里](#)找到。

---

<sup>4</sup>本次实验所用工具均为常用工具，环境搭建不困难；配置、维护系统环境也是程序员的基本技能。

你可能会依次遇到这个错误和这个错误。你可能需要升级 WSL kernel，并借助官方工具将 Windows 升级至最新版本。

此外，vscode 有 WSL 相关插件可以帮助你在 windows 平台上进行开发。

**cygwin 的安装** 你可以在这里获取 cygwin，并在安装时点选好你需要的程序（GCC/Flex/Bison/make/git/vim 等）。你可能想要了解 cygwin 与 wsl 的区别，或者与 mingw 的区别，以减少实验中踩坑几率。cygwin 中的 GCC 与 Linux 中 GCC 参数会有差异（比如是否拥有-m32 参数），借此机会，你可以练习查阅手册与探索 stackoverflow。

为了在 64 位 cygwin 中编译与运行 32 位程序，我们需要安装一些额外的工具，具体的可以看这里的讨论，不用担心没有一次安装全部内容，cygwin 的安装程序可以非常便捷的增加或减少安装的程序包。

完成后你可以直接进入测试步骤。

## 2.2.2 Mac

**Docker** 实际上 Docker 是在 Windows 和 Mac 通用的，但是 Windows 上有更加方便的解决方式；而对于 Mac 的同学，Docker 要比虚拟机要方便一些的。

你可以看这一份教程启用 Ubuntu 容器<sup>5</sup>。了解 docker 与虚拟机的差异也会很有帮助。

Vscode 拥有 remote-container 插件，或许可以帮助你进行开发。

## 3 工具安装

我们提供了自动化脚本进行安装基本工具及词法分析、语法分析工具，注意 \* 并不包括 \* 框架特定工具。

---

```
1 sudo bash build_support/packages.sh -y
2 # 注意，此自动化脚本可能并不包括所有需要的库，所以如果需要其他包，请自行安装。且你需要继续
3 # 按照指导书的指示安装 qemu 及特定框架的编译工具。
```

---

### 3.1 基本工具

由于需要在 Linux 环境完成作业，开发工具链的安装部署就是非常重要的了。你只需要使用你的 Linux 发行版的相应包管理器<sup>6</sup>安装即可。比如在 Ubuntu 上，你可以通过以下指令获得它们：

---

```
1 sudo apt update
2 sudo apt install build-essential
3 sudo apt install gcc-multilib
4 sudo apt install llvm clang
5 # 如果你选择 RISC-V 架构进行实验，* 先不要 * 使用 apt 安装 qemu，安装方法见下面的 3.3.2 节
6 sudo apt install qemu
```

---

<sup>5</sup> 使用 latest ubuntu 即可。

<sup>6</sup> 如果下载速度太慢的话记得换下载源哦，具体方法请自行搜索。

---

```

7   sudo apt install qemu-system
8   sudo apt install qemu-user
9   # whatever you need
10  # 如果接触过 Linux 的同学们可能会发现，我们有时候用到 apt-get 有时候用 apt,
11  # 那么两者具体有什么差异吗
12  # 其实这是非常简单的问题，同学们感兴趣的话自行查询可以很容易得到答案

```

---

## 3.2 词法分析和语法分析辅助工具

当程序员考虑要写一个文本解析程序时，他可能会想到 Flex+Bison 或 antlr。本次实验建议使用 Flex+Bison 来完成词法解析与语法分析，但如果你对其它工具更感兴趣，仍可以自行采用其它工具完成，只需保证解析结果可以正确转换到抽象语法树即可。你可以利用搜索引擎进一步了解它们的差异。实际上，LL/LR 之外，还有其他 parsing techniques，同学们也可拓展了解。

---

```

1  sudo apt install flex
2  sudo apt install bison

```

---

## 3.3 工具链

### 3.3.1 arm 工具链

之前我们说过，目前我们使用的电脑往往是 x86 架构的，那么我们要是想要编译出 arm 架构的可执行文件该怎么办呢？这时候就会用到交叉编译了，在 Ubuntu 的 terminal 中，你可以使用以下指令安装 arm 交叉编译器：

---

```

1  # 实际上我们可以看到有 gcc-arm-linux-gnueabi 和 gcc-arm-linux-gnueabihf 两种格式,
2  # 具体的差异同样推荐同学们自行查阅了解
3  # 2025 年的编译赛道更新到了 ARMv8 架构，因此推荐使用 gcc-aarch64-linux-gnu
4  sudo apt install gcc-aarch64-linux-gnu

```

---

若你使用其他 Linux 发行版，相信聪明的你可以找到相应的安装方法：-）。

### 3.3.2 Risc-V 工具链

推荐在 Ubuntu22.04 下进行本学期的 RISC-V 架构编译系统原理实验。

---

```

1  sudo apt-get install build-essential clang gcc git vim
2  sudo apt-get install ninja-build pkg-config libglib2.0-dev libpixman-1-dev
3  #(如果中间有提示缺少其他工具，可以自行安装)
4
5  #qemu-riscv64 安装
6  wget https://download.qemu.org/qemu-7.0.0.tar.xz
7  tar xvJf qemu-7.0.0.tar.xz

```

---

```

8 cd qemu-7.0.0/
9
10 ./configure --enable-slirp
11 --target-list=riscv64-softmmu,riscv64-linux-user --prefix=/opt/qemu
12 # 上面两行为一整条命令，输入时中间不要有换行，但是需要空格
13
14 make -j`nproc`
15 # (如果上一步执行失败并且报错不是缺少某些库，尝试 rm -rf build 后
16 # 重新执行上一条命令后再 make -j4，如果缺少某些库则自行根据报错信息安装)
17 sudo make install
18 echo 'export PATH=$PATH:/opt/qemu/bin:$PATH' >> ~/.bashrc
19 source ~/.bashrc
20 qemu-system-riscv64 --version
21 # 如果正确输出版本，则说明安装成功
22
23 riscv64-unknown-elf-gcc -v

```

---

特别的，对于 RISC-V 工具链，我们推荐在本学期的实验中使用 medany 内存模型完成实验作业，后续实验中提供的评测脚本默认会将程序基址放到 0x90000000 处，若使用其他内存模型的链接库可能导致 Link Error。你可以自由选择下面两种工具链：

- medlow：使用该工具链可直接使用 apt 安装：

---

```

1 sudo apt install gcc-riscv64-unknown-elf
2 # 若你使用该工具链，请在后续实验中将实验指导书以及工具脚本中的
3 # -Ttext=0x90000000 去掉或改成 -Ttext=0x10000000

```

---

- medany：使用该工具链需要自行编译安装，或者使用实验资料提供的二进制文件：

---

```

1 # 方法一：自行编译
2 # 需注意，编译时间较长，且需预留 20G 以上空间
3 # 将 {TARGET_PATH} 换为你希望存放 riscv-gnu-toolchain 编译结果的路径
4 git clone https://github.com/riscv/riscv-gnu-toolchain
5 cd riscv-gnu-toolchain
6 git submodule update --init --recursive
7 ./configure --prefix={TARGET_PATH} --with-arch=rv64gc --with-abi=lp64d --with-cmodel=m
8 make -j$(nproc)
9 # 编译完成后，请自行将 {TARGET_PATH}/bin 添加到 PATH 中，
10 # 将 {TARGET_PATH}/lib 添加到 LD_LIBRARY_PATH 中
11
12 # 方法二：使用实验资料提供的二进制文件
13 # 自行下载飞书群提供的 riscv-elf-toolchains.tar.gz

```

---

```
14      # 若你系统的 glibc 版本较低 (如使用 Ubuntu 18)  
15      # 可能无法正常使用, 需要自行编译安装  
16      tar -xzvf riscv-elf-toolchains.tar.gz  
17      cd riscv-elf-toolchains  
18      # 下面两个步骤为设置环境变量, 默认导入到 .bashrc  
19      # 对于使用其它 shell 的同学, 我相信你有自行修改的能力  
20      chmod +x ./init.sh  
21      ./init.sh
```

---

## 4 环境测试

### 4.1 测试 GCC/make/Flex/Bison

在实验环境搭建成功后, 你可以 clone 这个仓库进行测试。测试方法即根据 README.md。  
若你不在 Linux 平台上测试 (比如 cygwin), 可能需要微调 Makefile 中的编译参数。

### 4.2 熟悉 arm 工具链

假设在某文件夹中, 你写了一个测试用的 main.c 文件。你可以先将它编译为某要求的汇编文件, 再将它编译为可执行程序, 再运行它做测试。

```
1  # 在实验中你可能遇到“不知道汇编代码应该是什么样子”的情况。那么就让交叉编译器生成一个  
2  # 标准的给你看!  
3  # 这里的编译工作只是为了让你熟悉整个工作流而进行的, 之后真正写作业时汇编代码会是你手写  
4  # 的或者通过你编写的编译器生成的。  
5  # 你可能会认为可以以这种方式逃避汇编编程作业, 但我们会问你你写的每一条代码的含义, 包括  
6  # 伪指令, 以确保你的了解。  
7  # 编译参数是什么含义, 就由你大展身手了。  
8  
9  aarch64-linux-gnu-gcc -o arm.s -S -O0 main.c -fno-asynchronous-unwind-tables  
10  # -static 参数是在编译的哪个阶段起作用呢  
11  aarch64-linux-gnu-gcc arm.s -o arm -static  
12  # 你可能会神奇地发现不加 qemu-aarch64 也可以运行, 是进行隐式调用的原因  
13  qemu-aarch64 ./arm
```

---

### 4.3 熟悉并测试 Risc-V 工具链

```
1  git clone https://github.com/CentaureaHO/SysY_Compiler_2025.git  
2  cd SysY_Compiler_2025  
3  make -j$(nproc)  
4
```

---

```

5  # 随后运行四组测试样例，如果环境配置没有问题，所有样例均会通过
6  python test.py testcase/functional_test/Basic test_output/asm 0 S
7  python test.py testcase/functional_test/Advanced test_output/asm 0 S
8  python test.py testcase/functional_recover/functional test_output/asm 0 S
9  python test.py testcase/functional_recover/h_functional test_output/asm 0 S
10
11 # 如果很多样例出现 Link Error, 请尝试使用后续的命令测试其中任意一个测试文件
12 # 如果在打印到错误中有很多 `Relocation truncated to fit: R_RISCV_HI20` 的信息
13 # 那么说明你使用的交叉编译器并不支持 medany 内存模型，请参考上面的安装步骤重新安装交叉编译器
14
15 # 你可以在 testcase/functional_test 找到样例输入。
16 # 你可以在 test_output/asm 文件夹下找到代码的输出。
17 # 你可以将上述指令中的 0 改为 1, 来测试带优化的代码生成
18 # 你可以将上述指令中的 S 改为 llvm, 来测试中间代码生成, 此时建议将 test_output/asm
19 # 改为 test_output/ir
20 # 此时你可以对比一下输入和输出, 来大致了解本学期我们需要做什么样的工作。
21 # 如果你还想尝试编写 sysy 程序并运行, 项目内提供了单文件的编译脚本, 你可以这样使用它:
22 bin/compiler in.sy -llvm -o out.ll -O0
23 ./ll2bin.sh out.ll exec.bin
24 ./exec.bin
25
26 bin/compiler in.sy -S -o out.s -O0
27 ./asm2bin.sh out.s exec.bin
28 qemu-riscv64 exec.bin

```

---

如果你在测试目标代码生成时发现有一部分样例通过，但是大部分无法通过，并且 qemu 报错非法指令，这可能是你 qemu 版本不匹配造成的（原因大概率是你的环境之前已经安装过 qemu）。如果你只是想完成编译系统的实验作业，不想做进一步的优化或者参加编译系统比赛，可以忽略这些报错。如果你有兴趣，可以自己探究一下是什么指令导致了旧版本的 qemu 出现 Illegal instruction。这些指令的功能是什么。

如果出现其他报错，推荐先检查以下自己环境与上述仓库中的 readme.md 所要求的是否匹配，如果依旧无法运行，可以向助教寻求帮助，并附上详细的报错信息。

最后还需要测试一下 flex 和 bison 在 RISC-V 框架下版本是否匹配

---

```

1 make lexer
2 # 如果这条命令没有任何报错，则说明环境没有问题。

```

---

好啦，到此为止就算是我们的开发环境部署完毕啦！优秀的程序员会在开发环境部署完毕后保存镜像，以防止后续粗心导致环境出现错乱。