# 秋季学期实验指导书

实验版

课程教学用

**2022 年秋季**

《计算机体系结构》课程实验指导书

# 实验要求与注意事项

1. 实验分组进行，每 6-7 人一组，实验评分包括现场验收和实验报告两部分。
2. 实验时间从实验任务布置日开始至本学期末结束，考核时间暂定为十二月底。
3. 现场验收由助教根据现场硬件情况核定，按照小组进行验收（要求小组所有成员必须在场），现场验收成绩占个人课设总成绩 50%。
4. 实验报告内容包括：实验目的，设计方案（原理说明及框图），关键代码及文件清单，仿真结果及分析，综合情况（面积和时序性能），硬件调试情况，成员分工，实验收获等。实验报告占个人课设总成绩 50%。注：需要在实验报告对应部分标明完成人姓名。
5. 实验报告提交方式：实验报告（pdf）和设计代码打包后发送至助教老师邮箱，文件名按照“2022 体系结构课程设计+实验小组编号”命名。
6. 硬件实验板将在实验开始后，由院里发放到各实验小组，实验验收后上缴。
7. 同学可在实验室完成实验，也可在寝室或其他地方自行完成，没有考勤限制，现场验收必须在指定时间段内的教室或机房进行。
8. 根据综合结果，在流水线设计功能正确的所有小组中，具有最高时钟频率的前 5 个小组将可获得 5-10%的加分，申请加分的小组需要单独提交申请，并需另提交一份代码和设计说明。
9. 实验严禁抄袭，有抄袭嫌疑（实验报告或者设计代码出现雷同、回答问题反映出相关工作明显非本人完成等）的现场实验或者实验报告按零分处理。

# 秋季学期实验：体系结构课程铁人三项实验设计

## 实验名称：

计算机体系结构铁人三项实验设计

## 实验目的：

培养学生的计算机基本功：

1、 熟悉现代处理器的基本工作原理；掌握流水线处理器的设计方法；基于流水线扩展更多功能。

2、 针对特定函数任务下的精简编译器与操作系统设计，形成处理器、操作系统以及编译器的完备体。

## 实验工具：

**HDL**：Verilog；

**IDE**：Vivado；

**FPGA 开发板。**

## 实验内容：

# RISC-V 处理器设计部分

本次课设任务中的处理器，需要与后续介绍的简易操作系统以及编译器兼容，可先约定指令集后与队友同步执行三部分设计任务。处理器架构、系统以及编译器可基于开源工作进行开发，而课设所指定的基础部分如指令选型与编写、流水线为同学必须独立完成选项，动态调度、分支预测、Ceche 等（加分项）设计可依据小组实力自行设计，需指明独立完成部分与开源复用部分，禁止完全照搬套用。具体设计细节当前开源资料较多，同学们可结合本指南给出的参考链接与资料多方查阅（如《手把手教你设计 CPU-RISC-V 处理器》）。

## RISC-V 指令集选型

RISC-V 指令集分为特权指令和非特权指令，单纯运行程序，我们只需要实现部分非特权指令，但为了支持操作系统，我们在实验中要实现部分特权指令，用于支持内核态的程序。

RISC-V 并非是一个整体，而是采取了模块化的设计，其将非特权指令划分成多个不同模块(module)，每个模块中都包含一些指令，一个模块用来完成一件事情(有点类似 Unix 的哲学，每个东西仅做最简单的事情)，比如整数指令模块(一般用字母“I”表示,"I"是 Integer 的首字母)，该模块包含了最简单的整数加减比较，分支跳转等指令，在整数模块中甚至没有任何与乘法有关的指令。如果我们实现整数指令模块中的所有指令，我们就可以说自己的处理器是 RISC-V 的，当然这是最简单的 RISC-V 处理器，一般商用的 RISC-V 会实现很多的指令模块，来提升性能。

RISC-V 指令集中包含了对 32 位指令和 64 位指令(32 和 64 主要指的是寄存器位宽，指令本身长度都是 32bit 的)，我们用 RV32 表示处理 32bit 位宽数据的指令，用 RV64 表示处理 64bit 位宽的指令。

用位宽+指令模块的简写来表示一款处理器具体支持的指令，例如 RV32I，这表示处理器实现的是 32bit 寄存器位宽，"I"模块，即该处理器实现了整数指令

模块中所有指令，RV32IM 则表示在上述基础上还实现"M"模块(乘除法指令模块)中所有的指令。

目前大多数 RISC-V 的芯片都采取 RV64IMAFDC 这一套组合作为自己完整支持的指令，亦称之为 RV64GC。相应的软件配套(编译器，操作系统)也都为 RV64GC 进行支持，涉及浮点单元等部件，但实现 RV64GC 难度较大，因此在本次实验中无需实现这么多指令。实验的要求是最低实现 RV32IMC 中所有指令，“I”是整数指令模块，“M”是乘除法指令模块，“C”是压缩指令模块，其压缩了指令的字长。具体这些模块包含那些指令在[非特权指令文档]可以找到。此外要求实现 zicsr 模块中要求的 CSR 寄存器，以及相应对 CSR 寄存器操作的指令。

为支持操作系统，实现 RV32IMC，zicsr 之后还需要实现一些特权指令。在特权指令中，需支持 Machine-Level 和 Supervisor-Level 的指令，具体可以参考[特权指令文档]，以提供对操作系统的支持。

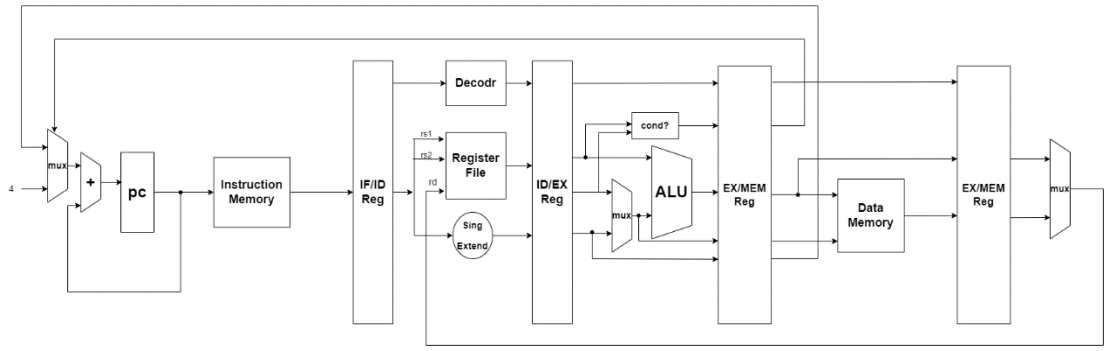
如果有能力，可实现 RV32IMAC，RV32GC 或 RV64GC，其包含更多的指令，如果能实现 RV64GC 基本能与现在的 RISC-V 生态兼容及其软件栈。不过也不用太担心，RV32IMC 的软件也不少，大多是编译器都提供了相应的支持。

# 参考

[非特权指令集文档]、[特权指令集]
[RISC-V 手册--一本开源指令集的指南]
[硬件描述语言]
[指令测试]：1、riscv-tests；2、riscv-tools
[模拟器]：1、riscv-isa-sim

# RISC-V 流水线设计

案例参考：




参考图：（仅作为 RISC-V 流水线架构理解，实际结构同学可自行设计）

按照教材所介绍的架构，依据图中的大致结构图，写出取指、译码、执行、访存、写回这个模块，组成基础的 RISC-V 流水线，建议可先实现 RV32I Base Integer Instructions 的基础指令。

**模块介绍：**

取指模块:从 Instruction Memory 中取出指令，送入 IF/ID Reg。

（可以进行分支预测，如记录当前指令是否跳转过，若跳转过，则直接跳转，更多方式请同学们自行选择）

译码模块：进行指令译码操作，同时对寄存器堆进行取操作数以及对立即数进行 Sign Extend，将所有的结果送入 ID/EX Reg。

执行模块：选择 ALU 的输入是寄存器还是立即数，然后计算，同时实现对寄存器的比较操作，最后将结果送入 EX/MEM Reg。

访存模块：对 Date Memory 进行取数据，和 ALU 运算结果送入 EX/MEM Reg 将需要写入寄存器堆的数据，写入寄存器堆。

参考资料:

tinyriscv

蜂鸟

木心

# 编译器部分

编译器，负责将编写的源代码文件转换成能够在对应机器上运行的二进制可执行文件，对于编译器的设计要求，考虑到编译器非本课程主要内容，期望设计上尽可能简易。

**关于考核，我们希望各位同学设计的编程语言与编译器能够实现的功能如下：**

1. **支持数组，实现数组求最大公约数算法。**
2. **实现快速排序。**
3. **实现图算法中的最短路径算法。**

对于我们的系统，我们期待各位能够实现的效果是，手动编写一个源代码文件，如：

```
int fib(int n) {
  if (n <= 2) {
    return 1;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}

int main() {
  int input = getint();
  putint(fib(input));
  putch(10);
  return 0;
}
```

关于目标文件的语法，由各位同学自己指定，可以是 C/C++语言的子集，也可以是类 Python 的语法。能够通过你的编译器生成对应的 RISC-V 汇编指令，如:

```
  .text
  .align  2
  .globl fib
fib:
  sw    ra, -4(sp)
  addi  sp, sp, -16
  li    t1, 2
  bgt   a0, t1, .l0
  li    a0, 1
```

```
  addi  sp, sp, 16
  lw    ra, -4(sp)
  ret
.l0:
  addi  s4, a0, -1
  sw    a0, 0(sp)
  mv    a0, s4
  call  fib
  mv    a3, a0
  lw    a0, 0(sp)
  addi  s4, a0, -2
  sw    a3, 0(sp)
  mv    a0, s4
  call  fib
  mv    s4, a0
  lw    a3, 0(sp)
  add   s4, a3, s4
  mv    a0, s4
  addi  sp, sp, 16
  lw    ra, -4(sp)
  ret

  .globl main
main:
  sw    ra, -4(sp)
  addi  sp, sp, -16
  call  getint
  call  fib
  call  putint
  li    a0, 10
  call  putch
  li    a0, 0
  addi  sp, sp, 16
  lw    ra, -4(sp)
  ret
```

并且能够生成可执行程序（executables），在目标处理器上完成运行。关于目标的 RISC-V 汇编语言，也取决于各位在设计处理器是选取的指令集与编译器的具体实现。

## 关于交叉编译

对于编译器我们可以关注两个架构，一个是编译器作为软件在什么架构的机器上运行，另一个是编译器的输出结果，也就是源码经编译器编译后的可执行文件在什么架构的机器上运行。如果两个架构是一样的，这就是我们最常见的编译方式，编译后的程序直接在本机上就能跑了，如果两个架构是不一样的，那么就是交叉编译(cross compile),我们用到的编译器就是交叉编译器. 本次实验我们需要做的就是一个交叉编译器，我们编写的编译器(后面称为 TinyCompiler)源码，经过编译后生成的是 x86_64 程序，在 x86_64 的 cpu 上运行，然后 TinyComplier 的输入是你自定义的语言源码，运行 TinyCompiler 之后生成的是 RISC-V 程序，这个程序是无法直接在你的电脑上运行的，如果想调试可以借助 RISC-V 模拟器进行测试，或者直接跑在你自己设计的 RISC-V 核上。

## 实验指南

一个编译器主要由前端、中端和后端组成，接下来我们准备分这三个部分给出一定的介绍与资料，供大家参考和借鉴。

## 前端

前端负责对输入的源代码文件进行词法分析、语法分析、语义分析。在这个阶段你需要定义编译器的词法，例如有哪些关键字（for, while, int）？通过一个状态机将源文件转化为抽象语法树以便于后续的分析（如交给语法分析器分析语法，分析语义，中端将抽象语法树转换成三地址码进行静态分析等），定义编译器的语法，什么样的语法是正确的(大括号是左开右闭表示一段代码)？并能给出一些修改建议。这部分我们仅要求能够正确定义词法和语法，能够将源代码文件解析成抽象语法树即可。

如何设计一门语言？其实前端的问题本质上是一个语言学的问题和一个数学问题，怎么定义一个语言？以及怎么样去写一个文本解析器（parser）把各种词素（token）都提取出来，这些问题目前有诸多研究，有很多现成的工具可以

供大家使用，当然你也可以基于 if else 直接写出来，也可以使用大部分编译原理教程中使用的 lex 和 yacc，也可以使用一些现代黑科技，比如 parser combinator（组合子技术）。

参考资料：

1. 写给前端的编译原理科普
   更详细的讲解了为什么前端的设计是长这样的，为什么编译器中需要这么多中间表示。
2. Write text parsers with yacc and lex
3. 知乎上关于 lex 和 yacc 的搜索结果]
4. Lexy: parser combinator library for C++17
   先进的组合子技术。
5. tinylang
   Learn LLVM 12 的第一个 section，使用及简单的循环实现了一个语言的前端。

## 中端

编译器的中端是大家做 research 和各种优化的重点，经典的各种编译优化包括活跃变量分析（死代码删除），循环不变量提取，强度削弱，指针分析等，当你使用 gcc 和 clang -O 选项时，-O 之后跟的数字就代表着启用这些优化的强度，**本课程设计不需要各位完成中端的设计，但是感兴趣的同学仍然可以自行探索和研究。**

参考资料：

1. 南京大学 软件分析
   2021 年公开的课程，讲的相当不错。
2. CSCD 70
   多伦多大学的经典课程
   （https://www.youtube.com/watch?v=S_OeRTePeXg&list=PL0qj8UdnOw3OZGMcM6DwvM1J2tttyy_D6）。

## 后端

一方面，后端需要根据你的程序生成正确的，可以执行的代码，但一般这样的代码执行效率很低，因为直接翻译生成的代码往往不够简洁，比如会生成大量的临时变量，指令数量也较多。因为翻译程序首先照顾的是正确性，很难同时兼顾是否足够优化，另一方面，由于高级语言本身的限制和程序员的编程习惯，也会导致代码不够优化，不能充分发挥计算机的性能。所以后端需要做一些依赖于

具体指令的优化，比如寄存器优化，软件流水，指令选取等，但是这部分内容不在本课程设计的考虑范围之内，各位只要能够将自己的源程序翻译成具体的指令，并且最好在目标设备上运行即可。

当然，如果你选择接入 LLVM IR，就可以很好的完成后续的各种事情，这也是我们所推荐的做法（不过学习 LLVM 有点难度，但是这项技术很有用），这部分内容可以参考的资料有很多，如:

1. miniSysY
2. YuLang
3. tinylang
4. toy-riscv-backend

   基于 LLVM 实现的 riscv 后端

虽然这样你需要将 RISC-V 的指令实现的标准一些，或者可以自己实现一个简单的编译器后端，看起来代码量也不是很大，如我们在下面给出的 kobayshi-compiler-backend。

## 完整的简易编译器实现参考

在开始设计之前，**你需要选择合适的编程语言与编程环境。**推荐各位使用 C/C++/Rust 完成编译器设计，比较推荐 C++，使用 C++实现编译器的版本较多，并且可以轻松接入 LLVM，当然市面上基于 JAVA 等语言的教程也有一些。编译器不是一个小项目，简易版本的编译器也不是，完成一个编译器设计必然需要一个项目管理工具，推荐各位使用 Makefile、CMake 来完成项目的管理，关于构建工具，我们也给出一些参考：

1. Makefile Tutorial
   Make 的基础使用方法
2. CMake Tutorial
   CMake 的基础使用教程
3. CMakeList Examples
   CMake 配置文件案例
4. xmake
   现代化一些的构建工具
5. bazel
   谷歌推出的一款构建工具

推荐各位使用 CMake 完成项目的构建。

接下来我们给出一些完备的简易编译器实现文档，供大家参考，上述提到的各种问题也都有一些更详细的说明，大家自行选择学习。

1. Learn LLVM 12
   以 LLVM 12 为例，阅读 LLVM 源代码的英文书籍，网上可以找到 pdf，在这本书中教会了大家如何实现一个 tinyC 编程语言，同样是一个 C 语言的子集，并且将该语言的抽象语法树对接到 LLVM IR 上，交给 LLVM 生成目标设备的代码（这是基本操作，基本上实现 CPU 上的 compiler 都会这么做，每个通用处理器的指令集设计出来都会想办法给 LLVM 递交后端的代码，这样只需要兼容 LLVM ，就可以完成目标设备的代码生成，兼容很多应用了）。
   Learn LLVM 12 中文版
   代码
2. 北大编译实践在线文档
   北京大学编译原理课程，一个面向 RISC-V 的简易的编译器实现。实际上，这份文档的作者 MaxXSoft，一个人实现了：
   一个 Rsic-V 指令集 CPU Fuxi
   一个面向 Fuxi 的编译器 YuLang
   一个可以在上面运行的操作系统 GeeOS
   和我们的课程有一些异曲同工之妙，也许可以给大家一些参考吧,但是这份文档中实现的编译器是 SysY，各位可以在 github 上搜索到一些参加课程的学生的作品。如 kobayshi-compiler。
3. BUAA miniSysY Tutorial
   北京航空航天大学的课程设计内容，实现了一个 minisys 编程语言，并且带你了解一些编译优化的内容跟，如果你想顺带学习 LLVM，选他。

## 实验最低要求

最低限度应该完成一个编译器的前端，然后借助 LLVM IR 之类的项目实现一个完整的程序编译，生成汇编代码，保证设计的语言能够支持最初提出的三个算法。

如果你认为接入 LLVM IR 太复杂，可以自行实现一个简易的后端，同样完成完整的程序编译，得到汇编程序。

如果你还有时间，可以考虑基于 LLVM 或者自己设计的后端添加一些优化算法，提升编译后程序的性能。

## 其他资料

其他架构的编译器

1. MIPS C Compiler

MIPS 的 C 编译器 采用了开源前端工具 flex 和 Bison

2. MiniCompiler
   MIPS 架构的 C 编译器，实现了前端，LLVM 后端和自行设计的后端

支持 RISC-V 的编译器

1. Rust 支持 RV32IMC 官方支持
2. Clang LLVM 支持 RV32IMC 官方支持
3. GCC 支持 RV32IMC riscv-gnu-toolchain

# 操作系统部分

在前面的实验中，我们设计了一个基于 RISC-V 的 CPU，有了处理器，则可以在该处理器跑程序。而在团队规约了将要完成设计的基础指令集后，同样可同步完成操作系统部分的开发工作，在开发过程中则需要两方对接，相互迭代。计算机的运行离不开操作系统。本部分实验中，我们需要设计一个简单的基于 RISC-V 的操作系统。由于在前面的设计中，同学们已经了解到了 RISC-V 架构设计的相关知识，并规定好基础指令集，则操作系统可基于约定好的 RISC-V 的指令进行设计。因为本门课程主要讲的是计算机体系结构，涉及到操作系统相关的实践设计可能对一些同学来说比较困难，有困难的同学可以参考一些开源的 OS 设计教程。

（基于 RISC-V 的 OS 框架设计，相关的教程视频讲解）

实际上，真实的操作系统设计是一个相当庞大复杂的代码工程，我们的课程中仅抽取操作系统设计工作中的核心部分，感兴趣的同学可以阅读更多相关的文献资料，了解真实操作系统的运行机制。推荐感兴趣的同学们可以阅读 Linux 操作系统的源码和相关源码分析的读物，如赵炯老师的《Linux 内核完全注释》。

CPU 上电后，执行地址会跳到一个固定的可执行代码，这段代码的主要任务是将存储设备上的第一个扇区（512B）的内容，拷贝到一个固定的位置。这 512B 的数据就是 Boot Loader。当拷贝完成后，跳转到 Boot Loader 代码的开头部分，开始执行 Boot Loader。Boot Loader 的代码通过 BIOS 读取 SD 上的操作系统内核，并放置到内存的指定位置，并最终跳转到操作系统的入口代码开始执行。至此，操作系统的引导程序完成了它的工作，真正的操作系统开始运行，进行各种初始化。

## 设计一个操作系统引导程序

本章节中，需要实现一个简单的操作系统引导程序 start.S 的编写，模拟 CPU 上电后，从默认地址执行 Boot Loader 的代码，拷贝内核程序到内存中，并跳转到 C 语言执行环境中。

start.S 文件需要实现：

1、调用 BIOS 打印“It’s bootblock!”

2、调用 BIOS 读取操作系统内核并放置到内存中

3、跳转到操作系统内核程序 kernel.c 并执行

其中,操作系统内核程序 kernel.c 目前仅需要完成简单的打印“Hello OS!”信息。

## 操作系统中的进程调度

在前面的设计中，了解操作系统的引导过程，并且跳转到内核程序中执行。但是内核程序只能打印简单的“Hello OS!”，不具备操作系统应有的基本功能。在本章节中，将完善现有的代码，使其具备任务调度的功能。

在进入到 kernel 函数后，打印出来“Hello OS!”，可以认为此时的操作系统已经有了一个内核线程,但是想要运行其他的程序,就需要开启新的用户进程。为此，需要进行 PCB 初始化和任务切换。

PCB 初始化：为了描述和控制进程的运行，操作系统需要为每个进程定义一个描述进程的数据结构,这就是进程控制块(Process Control Block,简称 PCB)。PCB 需要记录操作系统用于描述进程的当前状态和进程的全部信息，包括：进程号、进程状态、发生任务切换时保存的现场（通用寄存器的数值）、栈地址空间等信息。操作系统根据进程的 PCB 来感知进程的存在，并依次对进程进行管理和控制。在本实验中，同学们需要调研相关的资料，思考 PCB 中需要存储的信息，设计一个 PCB 的数据结构。

任务切换：操作系统根据进程的 PCB 来管理进程，从而实现进程的切换。进程发生切换的时候，操作系统将当前正在运行的进程的现场保存到 PCB 中，然后从其他进程的 PCB 中选择一个，恢复这个 PCB 中的现场，并跳转到下一个进程的操作。在本实验中，同学们只需要一个全局的 PCB 指针 current_running，它指向当前正在运行的任务。在切换任务的时候，首先将保存当前进程的现场（寄存器的值）到 current_running 指向的 PCB，然后修改 current_running 指向调度算法所选择的 PCB，最后从 current_running 新指向的 PCB 中恢复现场。

任务调度：一个 CPU 要处理许多任务的时候，就需要一个调度算法来调度每个任务。我们将介绍两种调度方法。第一种是在操作系统不具备中断处理能力时，

通过进程自己使用调度方法去“主动”交出控制权的非抢占式调度，第二种是操作系统具备了中断处理能力时，通过周期性触发时钟中断触发调度算法，从而使进程“被动”交出控制权的抢占式调度。

## 设计一个操作系统任务的进程切换的调度

在本章节中，需要同学们实现一个非抢占式调度的调度算法。

必做：在前一个实验的基础上，同学们需要丰富操作系统内核 kernel.c 函数的功能，实现如下的要求：

1、设计三个进程，print_task，count_task 和 draw_task，实现简单的三个功能。

| Task | Function |
| --- | --- |
| print_task | 在屏幕中打印”this is task [1]!” |
| count_task | 在屏幕中打印”this is task[2] schedule[count] times”，其中，count 初始值为 0，每次调用 count_task 的时候数值加 1 |
| draw_task | 在屏幕中打印出一个正方形 |

2、设计进程相关的数据结构 PCB，对 PCB 进行初始化操作

3、实现进程 PCB 的现场保存和现场恢复

4、实现 print_task，count_task 和 draw_task 的非抢占式调度方法。

**选做**：实际上，让进程自己主动交出控制权是并不现实的。让程序员在设计程序时并不考虑主动交出程序控制权，而是默认程序在运行时为尽量“独占”。因此，学有余力的同学们可以设计一个基于时钟中断的抢占式调度的任务调度算法，模拟更加真实的操作系统任务调度。

补充材料：

部分操作系统设计时 target 为 RV64GC，可以通过简单的修改，或者不修改直接编译为 RV32IMC 的 target。

https://github.com/manbing/mini-riscv-os

https://github.com/swetland/os-workshop

https://gist.github.com/cb372/5f6bf16ca0682541260ae52fc11ea3bb

https://github.com/mit-pdos/xv6-riscv

https://github.com/chyyuu/os_kernel_lab