

# 1 引论

潘志铭



### 课程介绍



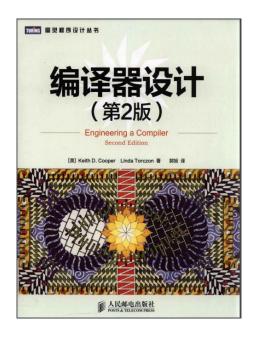
## 课程概要

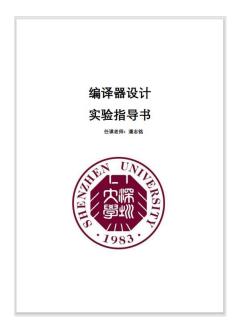
### • 教材

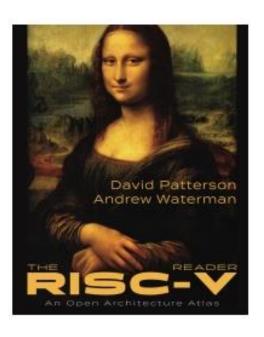
• 《编译器设计:第二版》

• 《编译器设计:实验教程书》

• The RISC-V Reader: An Open Architecture Atlas









## 课程概要

### • 安排

• 网站: <a href="https://github.com/xicongye/compiler-design">https://github.com/xicongye/compiler-design</a>

• 学时: 36

• 学分: 2

• 先修课程: C语言、计算机组成原理

### • 课程结构

• 理论部分: 上课听讲,下课作业,交书面作业

• 实践部分:按照《实验指导书》步骤完成实验报告



## 课程内容

• 1 引论

- 2 RISC-V汇编语言介绍
- 3 编译原理的基础知识

• 4 RISC-V工具链的开发



## 开始本课程之前需要做的准备

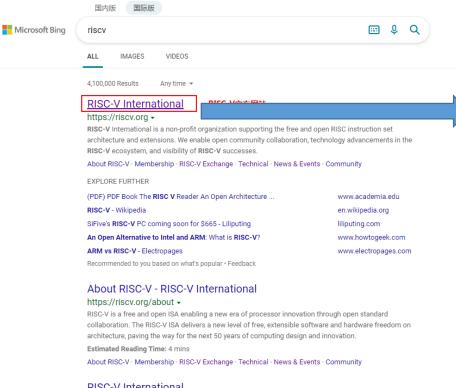
- 安装Ubuntu系统
  - WSL
  - VMware
- Git的基本操作
  - git clone, git diff, git status, git checkout
  - .....
- GitHub
  - 网页版使用界面
  - •
- Bash Shell的基本命令
  - Is, cp, mv, mkdir, vi
  - .....



### RISC-V介绍



### RISC-V的定义

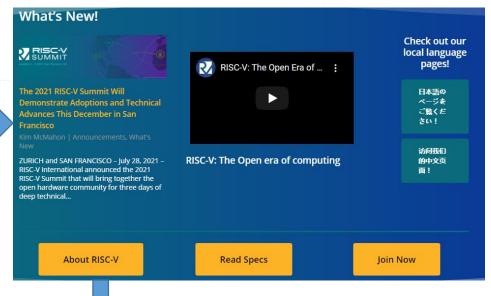


#### RISC-V International

https://community.riscv.org

RISC-V: The Free and Open RISC Instruction Set Architecture RISC-V is a free and open ISA enabling a new era of processor innovation through open standard collaboration. RISC-V ISA delivers a new level of open, extensible software and hardware freedom on architecture, paving the way for the next 50 years of computing design and innovation.

RISC-V(读作"risk five")是基于 精简指令集原理建立的开放指令 集架构。



RISC-V is a free and open ISA enabling a new era of processor innovation through open standard collaboration.

The RISC-V ISA delivers a new level of free, extensible software and hardware freedom on architecture, paving the way for the next 50 years of computing design and innovation.



### RISC-V的特点

### • 完全开源

开源采用宽松的BSD协议,企业完全自由免费使用,同时也容许企业添加自有指令集拓展而不必开放共享以实现差异化发展。

### • 架构简单

• RISC-V基础指令集则只有40多条,加上其他的模块化扩展指令总共几十条指令。RISC-V的规范文档仅有145页。

### · 易于移植\*nix

• RISC-V提供了特权级指令和用户级指令,使开发者能非常方便的移植linux和unix系统到RISC-V平台。

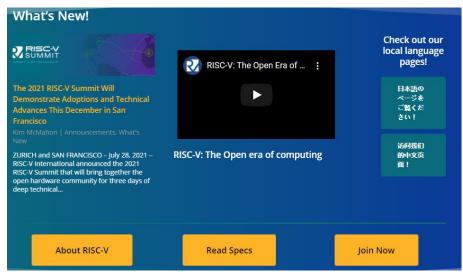


### RISC-V的特点

- 模块化设计
  - 用户能够灵活选择不同的模块组合,来实现自己定制化设备的需要。
- 完整的工具链
  - RISC-V社区提供了完整的工具链,并且RISC-V基金会持续维护该工具链。当前RISC-V的支持已经合并到主要的工具中,比如编译工具链gcc, 仿真工具qemu等。
- 社区贡献
  - 完整的工具链维护,大量的开源项目。risc-v的google 讨论组(名称: RISC-V ISA Dev)吸引各地自愿者参与讨论来不断改进risc-v架构。



### RISC-V的规范文档



#### Specifications



The RISC-V instruction set architecture (ISA) and related specifications are developed, ratified and maintained by RISC-V international contributing members within the RISC-V international Technical Working Groups. Work on the specification is performed on GitHub, and the GitHub issue mechanism can be used to provide input into the specification.

If you would like more information on becoming a member, please see the membership page.

#### ISA Specification

The specifications shown below represent the current, ratified releases. Work is being done on GitHub.

- Volume 1, Unprivileged Spec v. 20191213 [PDF]
- Volume 2, Privileged Spec v. 20190608 [PDF]

Past ratified releases include the term "ratified" in the release tag.

#### **Debug Specification**

This is the currently ratified specification:

• External Debug Support v. 0.13.2 [PDF] [GitHub]

This is the current stable draft:

 External Debug Support v. 1.0.0-STABLE [PDF]

#### Trace Specification

The processor trace specification was **approved** on March 20, 2020.

Trace Specification v.
 1.0 [PDF] [GitHub]

#### essor trace The RISC-V Compliance

The RISC-V Compilance Framework Version 0.1 is now available. This framework compares arbitrary models against a reference signature, and currently covers RV32IMC unprivileged spec only.

Compliance Framework

Work on Version 0.2 framework is underway which will compare two arbitrary models against each other (one of which can be a reference model), expand the configurations covered, and will automatically select tests according to the model configuration.

### **ISA Specification**

- Volume 1, Unprivileged Spec
- Volume 2, Privileged Spec

Unprivileged Spec描述的是RISC-V用户级(user-level)架构,例如RISC-V指令编码和用法;

Privileged Spec描述的是RISC-V特权架构,例如机器模式(machine-level)、管理员模式(supervisor-level);

### **Debug Specification**

External Debug Support

### **Trace Specification**

Trace Specification

Debug和Trace描述的是RISC-V软硬件的调试机制规范



## 开源的RISC-V处理器

#### Rocket Core

- 5级顺序流水线
- 使用Chisel语言开发
- https://github.com/chipsalliance/rocket-chip

#### Boom Core

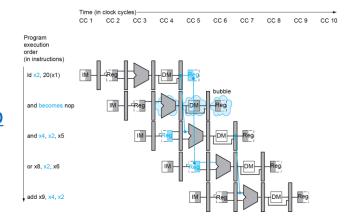
- 超标量乱序发射、乱序执行
- 使用Chisel语言开发
- https://github.com/riscv-boom/riscv-boom/

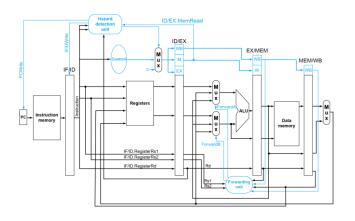
#### PULPino Core

- 4级顺序流水线
- 使用SystemVerilog语言开发
- https://github.com/pulp-platform/pulpino

### • 蜂鸟E203 Core

- 2~3级顺序流水线
- 使用Verilog语言开发
- https://github.com/riscv-mcu/e203 hbirdv2

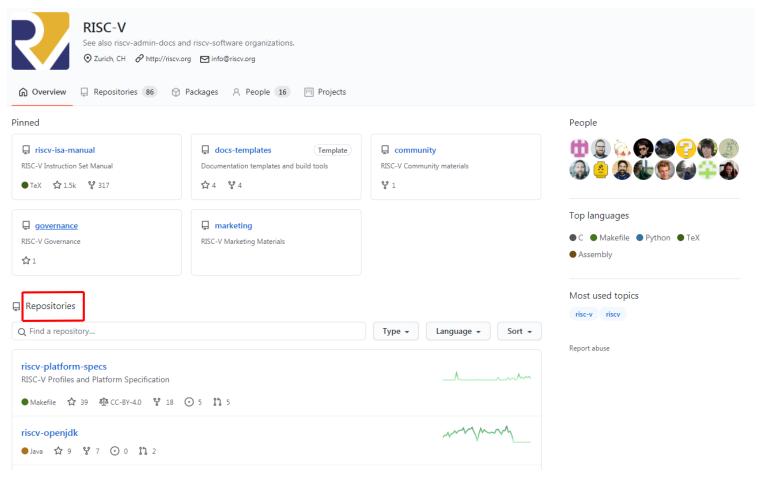






## RISC-V的软件工具

#### https://github.com/riscv



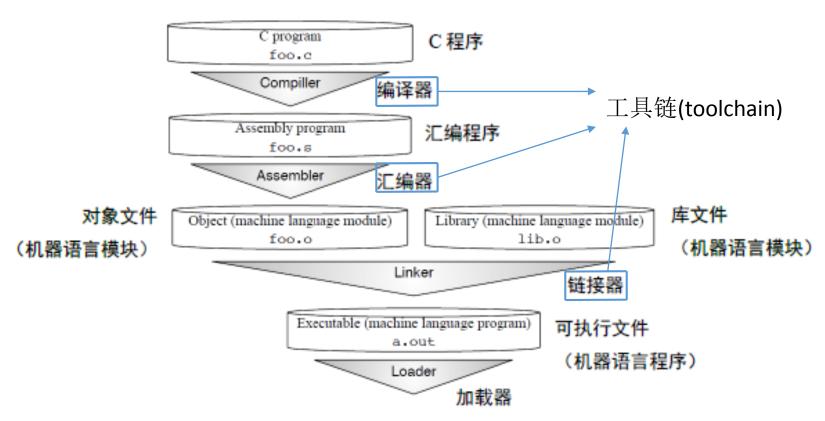
RISC-V相关的软件工具和文档都可以在RISC-V的官方GitHub账号上找到



### RISC-V工具链介绍



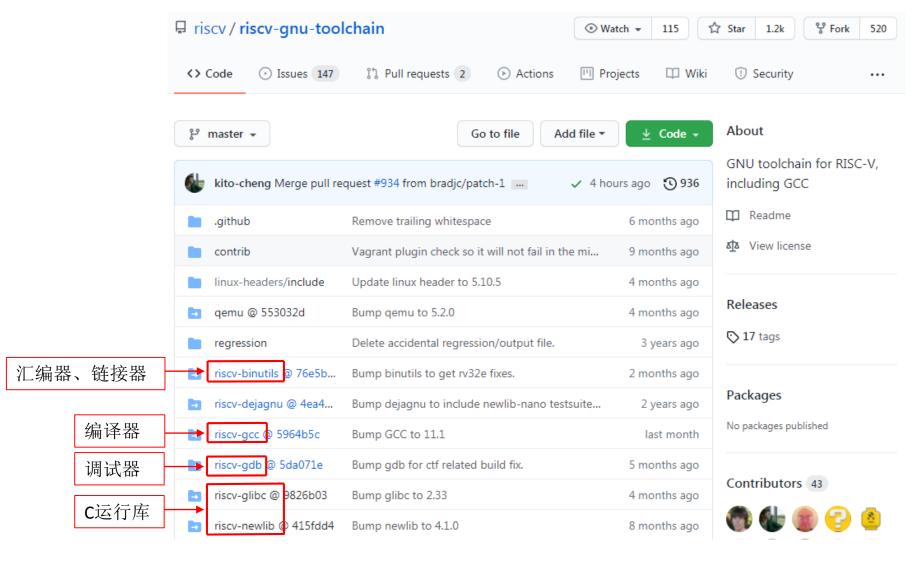
## 工具链的作用



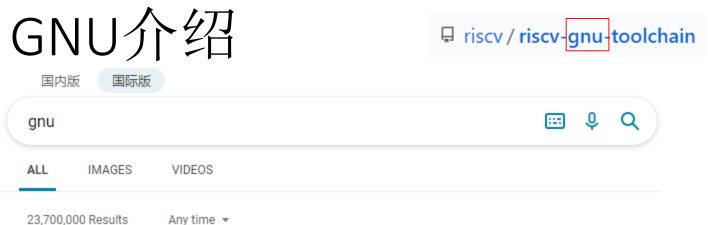
从C源代码翻译为可运行程序的步骤



### RISC-V的工具链







### The GNU Operating System and the Free Software Movement https://www.gnu.org •

Jul 21, 2021 · GNU is a Unix-like operating system. That means it is a collection of many programs: applications, libraries, developer tools, even games. The development of GNU, started in January 1984, is known as the GNU Project. Many of the programs in GNU are released under the auspices of the GNU Project; those we call GNU packages.

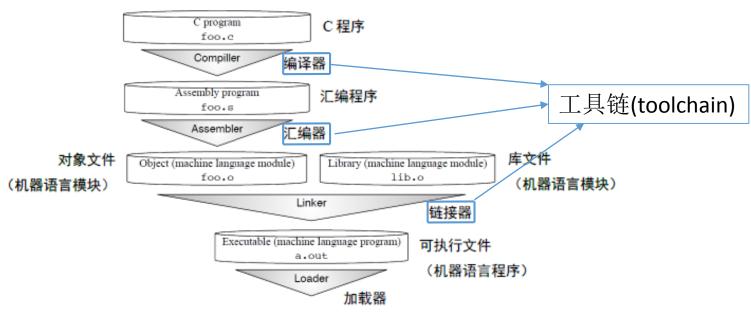
### GNU是什么?

GNU是一个<u>自由软件</u>操作系统—就是说,它尊重其使用者的自由。GNU操作系统包括GNU软件包(专门由GNU工程发布的程序)和由第三方发布的自由软件。GNU的开发使你能够使用电脑而无需安装可能会侵害你自由的软件。

我们建议安装<u>这些GNU版本</u>(更确切地说是,GNU/Linux发行版),它们完全是自由软件。<u>更多关于</u>GNU。



## GNU有哪些重要的软件工具?



#### GCC, the GNU Compiler Collection

The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Ada, Go, and D, as well as libraries for these languages (libstdc++,...). GCC was originally written as the compiler for the GNU operating system. The GNU system was developed to be 100% free software, free in the sense that it respects the user's freedom.

We strive to provide regular, high quality releases, which we want to work well on a variety of native and cross targets (including GNU/Linux), and encourage everyone to contribute changes or help testing GCC. Our sources are readily and freely available via Git and weekly snapshots.

https://gcc.gnu.org

#### **GNU Binutils**

The GNU Binutils are a collection of binary tools. The main ones are:

- Id the GNU linker.
- as the GNU assembler.

https://www.gnu.org/software/binutils



### GNU Compiler Collection(GCC)

- ·广义的GCC实质上是多个程序的集合:
  - GCC(GNU C Compiler)是编译工具,能够将C/C++语言编写的程序转换成处理器能够执行的二进制代码;
  - GCC既支持本地编译(即在一个平台上编译该平台运行的程序),也支持交叉编译(即在一个平台上编译供另外一个平台运行的程序);
  - Binutils(Binary Utilities)是一组二进制程序处理工具,包含汇编器as,链接器ld,反汇编器objdump,查看ELF文件信息的readelf,查看ELF文件大小的size等等;
  - C运行库,应用最为广泛的有glibc(GNU C Library),以及newlib(相比glibc更小,适合单片机系统使用)



- 命名方式: arch[-vendor][-os][-(gnu)eabi]-gcc
  - arm-none-eabi-gcc
  - arm-none-linux-gnueabi-gcc
  - arm-linux-gnueabi-gcc
  - arm-linux-gnueabihf-gcc
  - riscv64-unknown-linux-gnu-gcc
  - riscv32-unknown-linux-gnu-gcc
  - riscv64-unknown-elf-gcc
  - riscv32-unknown-elf-gcc

ARM architecture, no vendor, not target an operating system, complies with the ARM EABI

用于编译 ARM 架构的裸机系统(包括 ARM Linux 的 boot、kernel,不适用编译 Linux 应用 Application),一般适合 ARM7、Cortex-M 和 Cortex-R 内核的芯片使用,所以不支持那些跟操作系统关系密切的函数(比如fork函数),该工具链使用的是 newlib 库。



- 命名方式: arch[-vendor][-os][-(gnu)eabi]-gcc
  - arm-none-eabi-gcc
  - arm-none-linux-gnueabi-gcc
  - arm-linux-gnueabi-gcc
  - arm-linux-gnueabihf-gcc
  - riscv64-unknown-linux-gnu-gcc
  - riscv32-unknown-linux-gnu-gcc
  - riscv64-unknown-elf-gcc
  - riscv32-unknown-elf-gcc

ARM architecture, no vendor, creates binaries that run on the Linux operating system, and uses the GNU EABI

主要用于基于ARM架构的Linux系统,可用于编译 ARM 架构的 u-boot、Linux 内核、linux应用等,一般ARM9、ARM11、Cortex-A 内核,带有 Linux 操作系统的会用到,该工具链使用的是glibc库。



- 命名方式: arch[-vendor][-os][-(gnu)eabi]-gcc
  - arm-none-eabi-gcc
  - arm-none-linux-gnueabi-gcc
  - arm-linux-gnueabi-gcc
  - arm-linux-gnueabihf-gcc
  - riscv64-unknown-linux-gnu-gcc
  - riscv32-unknown-linux-gnu-gcc
  - riscv64-unknown-elf-gcc
  - riscv32-unknown-elf-gcc

两者区别在于选项-mfloat-abi的默认值不同,前者默认的选项是-mfloat-abi=softfp,也就是使用fpu执行浮点运算,但是传参数时使用的是通用整型寄存器,这样在处理中断时,只需要保存通用整型寄存器,优点是中断负荷小,绝点是参数需要转换成浮点形式再计算。后者的默认选项是-mfloat-abi=hard,即不仅使用fpu执行浮点运算,也使用浮点寄存器传递参数,这样参数就省去了转换的步骤,优点是性能好,缺点是中断负荷高。



- 命名方式: arch[-vendor][-os][-(gnu)eabi]-gcc
  - arm-none-eabi-gcc
  - arm-none-linux-gnueabi-gcc
  - arm-linux-gnueabi-gcc
  - arm-linux-gnueabihf-gcc
  - riscv64-unknown-linux-gnu-gcc
  - riscv32-unknown-linux-gnu-gcd
  - riscv64-unknown-elf-gcc
  - riscv32-unknown-elf-gcc

riscv64-unknown-linux-gnu-前缀表示该工具链是64位RISC-V架构的Linux版本工具链。这里的linux指的是该工具链使用glibc库。

同理,riscv32-unknown-linux-gnu-前缀是指32位RISC-V架构的Linux版本工具链。

前缀riscv64(还有riscv32的版本)与运行在64位或者32位电脑上毫无关系,此处的64和32是指如果没有通过-march和-mabi选项指定RISC-V架构的位宽,默认将会按照64位还是32位的RISC-V架构来编译程序。有关-march和-mabi选项的含义会在后面描述。



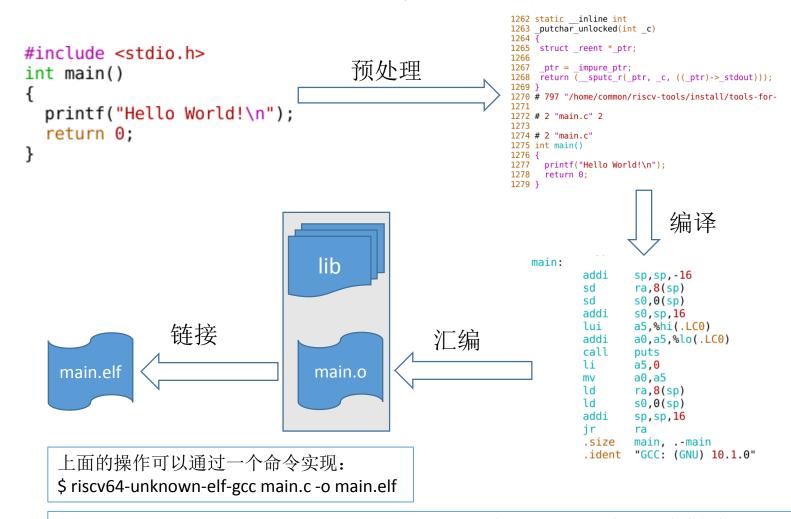
- 命名方式: arch[-vendor][-os][-(gnu)eabi]-gcc
  - arm-none-eabi-gcc
  - arm-none-linux-gnueabi-gcc
  - arm-linux-gnueabi-gcc
  - arm-linux-gnueabihf-gcc
  - riscv64-unknown-linux-gnu-gcc
  - riscv32-unknown-linux-gnu-gcc
  - riscv64-unknown-elf-gcc
  - riscv32-unknown-elf-gcc

以riscv64-unknown-elf-/riscv32unknown-elf为前缀表示该工具链为非 Linux(Non-linux)版本的工具链。

Non-Linux不是指当前版本工具链一定不能运行在Linux操作系统的电脑上,Non-Linux是指该GCC工具链会使用newlib作为C运行库。



### RISC-V工具链的简单使用



如果出现`riscv64-unknown-elf-gcc: command not found`的错误,需要将工具链路径加入PATH中: \$ export PATH=/opt/riscv:\$PATH



### RISC-V Binutils

- riscv64-unknown-elf-gdb
- riscv64-unknown-elf-as
- riscv64-unknown-elf-ld
- riscv64-unknown-elf-ar
- riscv64-unknown-elf-objdump
- riscv64-unknown-elf-readelf
- riscv64-unknown-elf-size
- riscv64-unknown-elf-objcopy

调试器

汇编器

链接器

用于打包静态库

反汇编器

查看有关ELF文件的信息

查看有关ELF文件大小信息

将ELF文件转成另外一种格式



## C运行库

为了解释C运行库,需要先回忆一下C语言标准。C语言标准主要由两部分组成:一部分描述C的语法,另一部分描述C标准库。C标准库定义了一组标准头文件,每个头文件中包含一些相关的函数、变量、类型声明和宏定义,譬如常见的printf函数便是一个C标准库函数,其原型定义在stdio头文件中。

C语言标准仅仅定义了C标准库函数原型,并没有提供实现。因此,C语言编译器通常需要一个C运行时库(C Run Time Libray,CRT)的支持。C运行时库又常简称为C运行库。与C语言类似,C++也定义了自己的标准,同时提供相关支持库,称为C++运行时库。



## C运行库

要在一个平台上支持C语言,不仅要实现C编译器,还要实现C标准库,这样的实现才能完全支持C标准。glibc(GNU C Library)是Linux下面C标准库的实现,其要点如下:

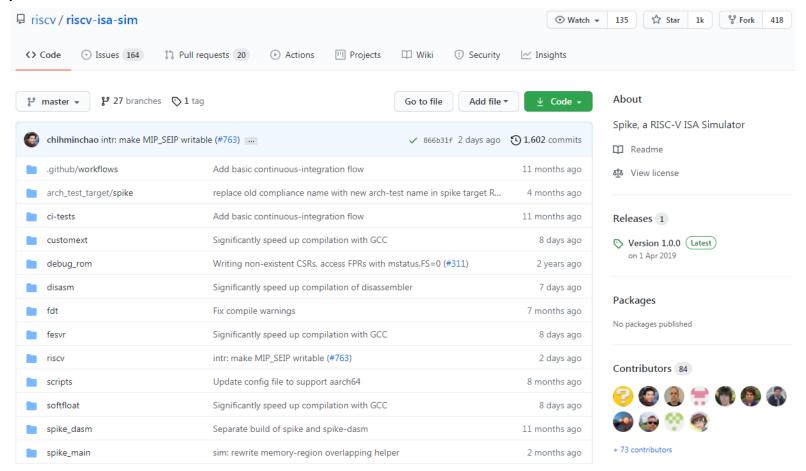
- glibc本身是GNU旗下的C标准库,后来逐渐成为了Linux的标准C库。glibc的主体分布在Linux系统的/lib与/usr/lib目录中,包括 libc 标准 C 函式库、libm数学函式库等等,都以.so做结尾;
- Linux系统通常将libc库作为操作系统的一部分,它被视为操作系统与用户程序的接口。譬如: glibc不仅实现标准C语言中的函数,还封装了操作系统提供的系统服务,即系统调用的封装;
- 对于C++语言, 常用的C++标准库为libstdc++;

newlib是一个面向嵌入式系统的C运行库。相对于glibc,newlib实现了大部分的功能函数,但体积却小很多。newlib独特的体系结构将功能实现与具体的操作系统分层,使之能够很好地进行配置以满足嵌入式系统的要求。由于专为嵌入式系统设计,newlib具有可移植性强、轻量级、速度快、功能完备等特点,已广泛应用于各种嵌入式系统中。



## spike介绍

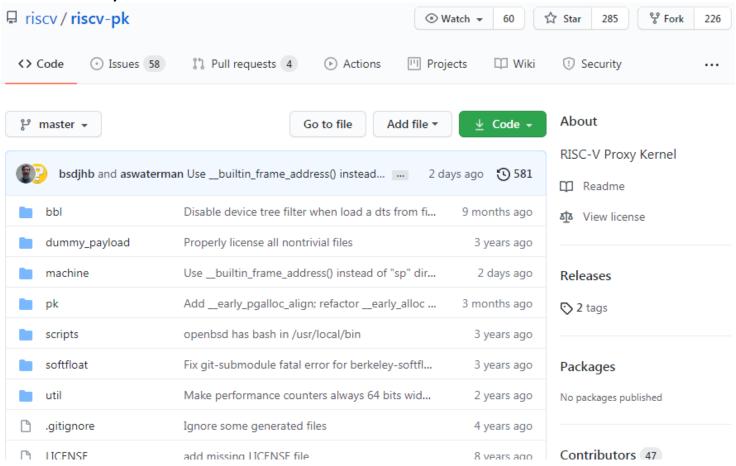
spike是一个RISC-V指令模拟器,它能够模拟一个或者多个RISC-V硬件线程的功能。





## pk介绍

RISC-V Proxy Kernel(RISC-V代理内核)用于给RISC-V执行文件提供执行环境。





## 结合spike和pk运行程序

- 编译helloword程序 \$ riscv64-unknown-elf-gcc main.c -o main.elf
- 将spike路径加入PATH环境变量中 \$ export PATH=/opt/riscv:\$PATH
- 将pk路径加入PATH环境变量中
   \$ export PATH=/opt/riscv/riscv64-unknown-elf/bin:\$PATH
- 运行spike和pk \$ spike pk main.elf



### Q & A



### Backup



### RISC-V工具链的构建步骤

- 获取源代码
  - \$ git clone https://github.com/riscv/riscv-gnu-toolchain.git
- 安装依赖文件
  - \$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev
  - \$ sudo apt-get install libmpfr-dev libgmp-dev gawk build-essential bison flex
  - \$ sudo apt-get install texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
- 开始构建
  - \$ cd riscv-gnu-toolchain
  - \$ mkdir build && cd build
  - \$ ../configure --prefix=/opt/riscv --enable-multilib
  - \$ make
  - \$ make install



## spike的构建步骤

- 获取源码
  - \$ git clone https://github.com/riscv/riscv.isa-sim.git
- 安装依赖文件
  - \$ sudo apt-get install device-tree-compile
- 开始构建
  - \$ cd riscv-isa-sim
  - \$ mkdir build
  - \$ cd build
  - \$ ../configure -prefix=/opt/riscv -enable-histogram
  - \$ make
  - \$ make install



## pk的构建步骤

- 获取源码
  - \$ git clone https://github.com/riscv/riscv-pk.git
- 开始构建
  - \$ cd riscv-pk
  - \$ mkdir build
  - \$ cd build
  - \$ ../configure -prefix=/opt/riscv -host=riscv64-unknown-elf
  - \$ make
  - \$ make install

构建pk之前需要先完成riscv-gnu-toolchain的安装

如果出现`riscv64-unknown-elf-gcc: command not found`的错误,需要将工具链路径加入PATH中: \$ export PATH=/opt/riscv:\$PATH



# 2 RISC-V汇编语言

潘志铭



### RISC-V寄存器



## RISC-V通用寄存器

Register									١.
Hard-wired zero 硬编码 0 x1 ra Return address 返回地址 No x2 sp Stack pointer 栈指针 Yes x3 gp Global pointer 全局指针 — x5 t0 Temporary/alternate link register im 寄存器 No /备用链接 No x8 s0/fp Saved register/frame pointer 保存寄存器 Yes /帧指针 x9 x10-11 a0-1 Function arguments/return values 函数参数 No /返回值 x12-17 a2-7 Function arguments 函数参数 No /返回值 x18-27 s2-11 Saved registers 保存寄存器 Yes x28-31 t3-6 Temporaries 临时寄存器 No f0-7 ft0-7 ft0-7 FP temporaries 浮点临时寄存器 No f8-9 fs0-1 FP saved registers 浮点临时寄存器 Yes	寄存器		接口名称		描述	L	<del>在调用中</del> 是	否保留?	Γ
x1raReturn address返回地址Nox2spStack pointer栈指针Yesx3gpGlobal pointer全局指针—x4tpThread pointer线程指针—x5t0Temporary/alternate link register limbar per NoAmount with a second per Nox8s0/fpSaved register/frame pointer保存寄存器 Yes /帧指针x9s1Saved register 保存寄存器Yesx10-11a0-1Function arguments/return values 函数参数No /返回值x12-17a2-7Function arguments 函数参数Nox18-27s2-11Saved registers保存寄存器Yesx28-31t3-6Temporaries临时寄存器Nof0-7ft0-7FP temporaries浮点临时寄存器Nof8-9fs0-1FP saved registers浮点临时寄存器No	Register	Т	ABI Nam	е	Description	]	Preserved a	cross call?	
x2spStack pointer栈指针Yesx3gpGlobal pointer全局指针—x4tpThread pointer线程指针—x5t0Temporary/alternate link register limbar Register Register limbar Register limbar Register Registe	x0	I	zere	$\overline{}$	Hard-wired zero 硬编码 0		_	_	
x3 gp tp Thread pointer 全局指针 — Thread pointer 线程指针 — Thread pointer 线程指针 — Thread pointer 线程指针 — Temporary/alternate link register Implies Pointer No / Saved register/frame pointer 保存寄存器 Yes / 帧指针 x9 s1 Saved register 保存寄存器 Yes x10—11 a0—1 Function arguments/return values 函数参数 No /返回值 x12—17 a2—7 Function arguments 函数参数 No /返回值 x18—27 s2—11 Saved registers 保存寄存器 Yes x28—31 t3—6 Temporaries 临时寄存器 No f0—7 ft0—7 ft0—7 fP temporaries 浮点临时寄存器 No fs0—1 FP saved registers 浮点临时寄存器 Yes	x1		ra		Return address 返回地址		N	0	
tp to Thread pointer 线程指针 — Temporary/alternate link register III 時寄存器 No /备用链接器 No x6-7 t1-2 Temporaries 临时寄存器 No x8 so/fp Saved register/frame pointer 保存寄存器 Yes /帧指针 x9 s1 Saved register 保存寄存器 Yes x10-11 a0-1 Function arguments/return values 函数参数 No /返回值 x12-17 a2-7 Function arguments 函数参数 No /返回值 x18-27 s2-11 Saved registers 保存寄存器 Yes x28-31 t3-6 Temporaries 临时寄存器 No f0-7 ft0-7 ft0-7 FP temporaries 浮点临时寄存器 No fs0-1 FP saved registers 浮点临时寄存器 Yes	x2		sp		Stack pointer 栈指针		Ye	2S	╫╴
x5	x3		gp		Global pointer 全局指针	ŀ		_	
x6-7 t1-2 Temporaries 临时寄存器 No saved register/frame pointer 保存寄存器 Yes /帧指针 x9 s1 Saved register 保存寄存器 Yes x10-11 a0-1 Function arguments/return values 函数参数 No /返回值 x12-17 a2-7 Function arguments 函数参数 No x18-27 s2-11 Saved registers 保存寄存器 Yes x28-31 t3-6 Temporaries 临时寄存器 No f0-7 ft0-7 ft0-7 FP temporaries 浮点临时寄存器 Yes Yes FP saved registers 浮点临时寄存器 Yes	x4		tp		Thread pointer 线程指针		_	_	
x8s0/fpSaved register/frame pointer保存寄存器 Yes /帧指针x9s1Saved register保存寄存器x10-11a0-1Function arguments/return values 函数参数No /返回值x12-17a2-7Function arguments 函数参数Nox18-27s2-11Saved registers保存寄存器Yesx28-31t3-6Temporaries临时寄存器Nof0-7ft0-7FP temporaries浮点临时寄存器Nof8-9fs0-1FP saved registers浮点保存寄存器Yes	x5		t0	4	Temporary/alternate link register #	Ç,	け寄存器 №	0/备用链接	寄
x9 s1 Saved register 保存寄存器 Yes x10−11 a0−1 Function arguments/return values 函数参数 No /返回值 x12−17 a2−7 Function arguments 函数参数 No /返回值 x18−27 s2−11 Saved registers 保存寄存器 Yes x28−31 t3−6 Temporaries 临时寄存器 No f0−7 ft0−7 ft0−7 FP temporaries 浮点临时寄存器 No f8−9 fs0−1 FP saved registers 浮点保存寄存器 Yes	x6-7		t1-2		Temporaries 临时寄存器		N	0	
x10-11a0-1Function arguments/return values 函数参数No /返回值x12-17a2-7Function arguments 函数参数Nox18-27s2-11Saved registers 保存寄存器Yesx28-31t3-6Temporaries 临时寄存器Nof0-7ft0-7FP temporaries 浮点临时寄存器Nof8-9fs0-1FP saved registers 浮点保存寄存器Yes	x8		s0/fp		Saved register/frame pointer \$	₹7	字寄存器 Ye	es /帧指针	
x12-17       a2-7       Function arguments 函数参数       No         x18-27       s2-11       Saved registers 保存寄存器       Yes         x28-31       t3-6       Temporaries 临时寄存器       No         f0-7       ft0-7       FP temporaries 浮点临时寄存器       No         f8-9       fs0-1       FP saved registers 浮点保存寄存器       Yes	x9		s1		Saved register 保存寄存器		Ye	es	┡
x18-27       s2-11       Saved registers       保存寄存器       Yes         x28-31       t3-6       Temporaries       临时寄存器       No         f0-7       ft0-7       FP temporaries       浮点临时寄存器       No         f8-9       fs0-1       FP saved registers       浮点保存寄存器       Yes	x10-11		a0-1		Function arguments/return values g	13	b参数 N	o /返回值	
x28-31     t3-6     Temporaries     临时寄存器     No       f0-7     ft0-7     FP temporaries     浮点临时寄存器     No       f8-9     fs0-1     FP saved registers     浮点保存寄存器     Yes	x12-17		a2-7		Function arguments 函数参数		N	o	
f0-7 ft0-7 FP temporaries 浮点临时寄存器 No f8-9 fs0-1 FP saved registers 浮点保存寄存器 Yes	x18-27		s2–11		Saved registers 保存寄存器		Ye	es	
f8-9 fs0-1 FP saved registers 浮点保存寄存器 Yes	x28-31		t3-6		Temporaries 临时寄存器		N	O	
	f0-7	Т	ft0-7		FP temporaries 浮点临时寄存器	1	N	0	
f10-11 fa0-1 FP arguments/return values 浮点参数/返回值 No	f8-9		fs0-1		FP saved registers 浮点保存寄存器		Ye	es	
	f10-11		fa0-1				/返回值 N	0	
f12-17 fa2-7 FP arguments 浮点参数 No	f12-17		fa2-7						
f18-27 fs2-11 FP saved registers 浮点保存寄存器 Yes	f18-27		fs2-11		FP saved registers 浮点保存寄存器	ŀ	Ye	es	
f28-31 ft8-11 FP temporaries 浮点临时寄存器 No	f28-31		ft8-11		FP temporaries 浮点临时寄存器	ŀ	N	0	J

RISC-V定义了32个XLEN-bit位宽的整型寄存器,和32个XLEN-bit位宽的浮点寄存器<sup>[1]</sup>

RISC-V各个寄存器的ABI别名

RISC-V各个寄存器的作用描述

在函数调用前后,寄存器的值是否需要保持不变

在RISC-V规范中,PC寄存器独立于通用寄存器

在RISC-V规范中,x0寄存器硬编码为0,读为0,写无效

- --- 上图来自《RISC-V手册》第42页
- --- [1] XLEN  $\in$  {32, 64}



## RISC-V CSR寄存器

Number	Privilege	Name	Description
			User Trap Setup
0x000	URW	ustatus	User status register.
0x004	URW	uie	User interrupt-enable register.
0x005	URW	utvec	User trap handler base address.
	•		User Trap Handling
0x040	URW	uscratch	Scratch register for user trap handlers.
0x041	URW	uepc	User exception program counter.
0x042	URW	ucause	User trap cause.
0x043	URW	utval	User bad address or instruction.
0x044	URW	uip	User interrupt pending.
		U	ser Floating-Point CSRs
0x001	URW	fflags	Floating-Point Accrued Exceptions.
0x002	URW	frm	Floating-Point Dynamic Rounding Mode.
0x003	URW	fcsr	Floating-Point Control and Status Register (frm + fflags).
			User Counter/Timers
0xC00	URO	cycle	Cycle counter for RDCYCLE instruction.
0xC01	URO	time	Timer for RDTIME instruction.
0xC02	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xC03	URO	hpmcounter3	Performance-monitoring counter.
0xC04	URO	hpmcounter4	Performance-monitoring counter.
		:	
0xC1F	URO	hpmcounter31	Performance-monitoring counter.
0xC80	URO	cycleh	Upper 32 bits of cycle, RV32I only.
0xC81	URO	timeh	Upper 32 bits of time, RV32I only.
0xC82	URO	instreth	Upper 32 bits of instret, RV32I only.
0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter3, RV32I only.
0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter4, RV32I only.
		:	1 /
0xC9F	URO	: hpmcounter31h	Upper 32 bits of hpmcounter31, RV32I only.



### RISC-V汇编指令

--- 以RV32I为例说明



## RISC-V基础整型指令

Category Name	Fmt	F	RV32I Base
Shifts Shift Left Logical	R	SLL	rd,rs1,rs2
Shift Left Log. Imm.	I	SLLI	rd,rsl,shamt
Shift Right Logical	R	SRL	rd,rs1,rs2
Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt
Shift Right Arithmetic	R	SRA	rd,rs1,rs2
Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt
Arithmetic ADD	R	ADD	rd,rs1,rs2
ADD Immediate	I	ADDI	rd,rsl,imm
SUBtract	R	SUB	rd,rs1,rs2
Load Upper Imm	U	LUI	rd,imm
Add Upper Imm to PC	U	AUIPC	rd,imm
Logical XOR	R	XOR	rd,rs1,rs2
XOR Immediate	I	XORI	rd,rsl,imm
OR	R	OR	rd,rs1,rs2
OR Immediate	I	ORI	rd,rsl,imm
AND	R	AND	rd,rs1,rs2
AND Immediate	I	ANDI	rd,rsl,imm
Compare Set <	R	SLT	rd,rs1,rs2
Set < Immediate	I	SLTI	rd,rs1,imm
Set < Unsigned	R	SLTU	rd,rs1,rs2
Set < Imm Unsigned	I	SLTIU	rd,rsl,imm
Branches Branch =	В	BEQ	rs1,rs2,imm
Branch ≠	В	BNE	rs1, rs2, imm
Branch <	В	BLT	rs1,rs2,imm
Branch ≥	В	BGE	rs1, rs2, imm
Branch < Unsigned	В	BLTU	rs1,rs2,imm
Branch ≥ Unsigned	В	BGEU	rs1,rs2,imm
Jump & Link J&L	J	JAL	rd,imm
Jump & Link Register	I	JALR	rd,rsl,imm
Synch Synch thread	I	FENCE	
Synch Instr & Data	I	FENCE.	.I
<b>Environment</b> CALL	I	ECALL	
BREAK	I	EBREAL	ζ

Base Inte			_	_			_			ference				
Category Name			RV32I Ba			+RV641		Categ			Fmt		V mnemo	onic
Shifts Shift Left Logical	R	SLL	rd,rsl,	rs2	SLLW TO	i,rsl,rs2		Trap	Mach-r	mode trap retur	n R	MRET		
Shift Left Log. Imm.	1	SLLI	rd,rs1,	shant	SLLIW ro	i,rsl,sham	t	Supe	rvisor-	mode trap return	n R	SHET		
Shift Right Logical	R	SEL	rd,rsl,	rs2	SRLW ro	d, rs1, rs2				Wait for Interrup		WFI		
Shift Right Log. Imm.	1	SELI	rd,rsl,	shant	SELIW FO	,rel,sham	t	MMU	Virtua	I Memory FENCI	E R	SPENC	E. VMA r	s1,1
Shift Right Arithmetic	R	SRA	rd,rsl,	rs2	SRAW TO	d, rs1, rs2		Ex	ample	s of the 60	RV Ps	eudoi	nstruct	ion
Shift Right Arith. Imm.	1	SRAI	rd,rs1,	shant	SRAIW ro	,rsl,sham	t	Brane	ch = 0 (	(BEQ re, x0, imm	) ]	BBQZ	rs,imm	
Arithmetic ADD	R	ADD	rd,rs1,	rs2		i,rsl,rs2		3	ump (us	ses JAL x0, imm	) ]	J ing		
ADD Immediate	1	ADDI	rd,rs1,	inn	ADDIW FO	i, rel, imm		MoN	le (uses	ADDI rd.rs.0	) R	HV rd	,rs	
SUBtract	R	SUB	rd.rsl.	rs2	SUBW ro	Lrsl,rs2		RETU	ım tuse	S JALR XO, O, TA	1	RET		
The second secon	U	LUI	rd,imm					Anna anna	-	The second second second second second	-	100000	D1/22	
Load Upper Imm	200							sed (		t) Instruction				
Add Upper Imm to PC Logical XOR	U	AUIPC		_		y Name	Fmt			RVC			equivale	
	R	XOR	rd,rsl,		Loads	Load Word and Word SP	CL	C.LW		d',rsl',imm	LW		csl',im	27.4
XOR Immediate	I	XORI	rd,rs1,		15.00000		CI	C.LWS		i, imm			p,imm*4	
OR Immediate	R	ORI	rd,rsl,			ad Word SP t Load Word	CT	C. FLW		i',rsl',imm	FLW		rsl',im	F.R
	R		rd,rs1,										p,imm*8	
AND Immediate	I	AND	rd,rsl,			Load Double	CT	C. PLD		d',ral',imm	FLD		rsl',im	
AND Immediate Compare Set <	R	SLT	rd,rsl,			Store Word	CS	C.FLD C.SW		d,imm	SW		p, imm*16 ,rs2', in	
Set < Immediate	I	SLTI	rd,rsl,			ore Word SP		C.SWS		81',r82',imm 82,imm	SW		sp,imm*	
Set < Immediate	R	SLTU	rd,rs1,			Store Word	CS	C.FSW		81',r82',imm	PSW		rs2',in	
Set < Imm Unsigned	ī		rd,rsl,			ore Word SP	CSS	C. PSW		2, imm	FSW		sp.imm*i	
Branches Branch =	B	880	rsl,rs2		0.0000000000000000000000000000000000000	Rore Double	CS	C. FSD		81',r82',imm	FSD		rs2', in	
Branch #	В	BNE	rsi,rs2	2100700		e Double SP	CSS	C. FSD		a2,imm	FSD		sp,imm*	
Branch <	В	BLT	rs1,rs2		Arithme		CR	C. ADD		rd,rsl	ADD		d,rsl	10
Branch ≥	В	BGE	rsl,rs2			Immediate	CI	C. ADD		rd.imm	ADDI		d, imm	
Branch < Unsigned	В	BLTU	rsl,rs2			P Imm * 16	CI			x0,imm	ADDI		p, imm*16	6
Branch ≥ Unsigned	В	BGEU	rs1, rs2			SP Imm * 4	CTW			rd', imn	ADDI		sp,imm*	
Jump & Link 38L	1	JAL	rd,imn	7.000		SUB	CR	C.SUB		rd,rsl	SUB		d,rsl	
Jump & Link Register	1	JALR	rd,ral,	1mn		AND	CR	C. AND		rd,rs1	AND		d.rsl	
Synch Synch thread	I	FENCE			ANE	Immediate	CI	C. AND		rd,imm	ANDI		d, imm	
Synch Instr & Data	T	FENCE	*		579355	OR	CR	C.OR		rd,rsl	OR		d, ral	
Environment CALL	i	ECALL				Xclusive OR	CR	C. KOR	81 .	rd,ral	AND		d,rsl	
BREAK	1	EBREAL	E.			MoVe	CR	C.NV		rd,rs1	ADD		sl.x0	
					Load	Immediate	CI	C.LI		rd, inn	ADDI		0, imm	
Control Status Regis	ter (	(CSR)			Load	Upper Imm	CI	C.LUI		rd, imm	LUI	rd, i		
Read/Write	1	CSRRW	rd, car	rsl	Shifts Sh	ift Left Imm	a	C.SLL	I.	rd,imm	SLLI		d, imm	
Read & Set Bit	1	CSRRS	rd, csr	rsl	Shift Rig	ht Ari. Imm.	CI	C. SRA	I	rd,imm	SRAI		d, imm	
Read & Clear Bit	1	CSRRC	rd, car			t Log. Imm.	CI	C.SRL		rd,imm	SELI	rd,r	d, imm	
Read/Write Imm	1	CSRRW	I rd, csr	, imm	Branche	s Branch=0	CB	C.BBQ	Z.	rsl',imm	BBQ		,x0,imm	
Read & Set Bit Imm	1		I rd, car		A 100 PM	Branch≠0	CB	C.BME	Z	ral', inm	BNE		,x0,imm	
Read & Clear Bit Imm	1	CSRRC	rd, csr	, imm	Jump	Jump	CJ	C.J		imn	JAL	x0,i	mm.	
						mp Register	CR	C.JR	-	rd,rsl	JALR	mo,r		
					Jump &		CI	C.JAL		imm	JAL	ra,i		
Loads Load Byte	1	LB	rd, rsl	A 12 12 1 1 1		ink Register	CR	C.JAL		rsl	JALR	ra,r	81,0	
Load Halfword	1	LH	rd, rsl	, inn	System	Env. BREAK	CI	C.EBB	EAK		EBREZ	NK.		
Load Byte Unsigned	1	LBU	rd, rsl			+RV641				al Compress				
Load Half Unsigned	1	LHU	rd, rs1		LWU FO	,rsl,imun		All RV	32C (e)	xcept C. JAE, 4 H	rord loa	ds, 4 m	ord strore	s) p
Load Word	1	LM	rd, ral	, imm	LD re	i, rsl, imm		1	ADD Wo	rd (C.ADOW)	Lo	ad Doub	eleword (	C.L
Stores Store Byte	S	SB	ral,ra	2,imm				ADD	Imm.	Word (C.ADDIW)	Load	Doubles	word SP (	C.L
Store Halfword	S	SB	rsl,rs	2,imm				SUI	Btract V	Word (c.susw)	Str	ore Dou	bleword (	(c.s
THOUSE LIBERTON OF	S	SW	rsl,rs		SD re	1,rs2,imm				and the same of the	Store	Double	word SP	(0.8
Store Word	32	-bit Inc	struction		ts		-		16	-bit (RVC) In				
Store Word		20	19 15	14 12	11	7 6	. 0	CR .	10 14 1	3 12 11 10	9 8 7	6. 5	4 1 2	1
Store Word 27 26 25 2			rsI	funct3		opeo		CI	from				ns2	0
Store Word  R 31 27 26 25 2 funct7	4 ES	12					(36)	148	funct3	imm rd/	and I		mm	0
Store Word    31   27   26   25   3	ES		rsl	funct3	rd	opeo					194	_		
Store Word	B	12	rsl rsl	funct3	limm 4:	0] opeo	de	css	funct3	lmm			ns2	0
Store Word  R 31 27 26 25 2 funct7 I [mm][11:0] S [mm][12:10:5]	IS IS	s2 s2	rsl		imm 4:1	0] opco 11 opco	de de	ciw	funct3 funct3	lmm	ım		ns2 rd*	0
Store Word  R 31 27 28 25 2	rs rs mm	s2 s2 [31:12]	rsl rsl rsl	funct3	imm[4:1 imm[4:1	0] opco 11] opco opco	de de de	CI W	funct3 funct3 funct3	imm imm	ım rel'	imm	nd' rd'	0
Store Word  R 31 27 28 25 2	rs rs mm	s2 s2	rsl rsl rsl	funct3	imm 4:1	0] opco 11 opco	de de de	ciw	funct3 funct3	lmm	ım	imm	ns2 rd*	



## RISC-V基础整型指令

Category Name	Fmt	F	RV32I Base
Shifts Shift Left Logical	R	SLL	rd,rs1,rs2
Shift Left Log. Imm.	I	SLLI	rd,rs1,shamt
Shift Right Logical	R	SRL	rd,rs1,rs2
Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt
Shift Right Arithmetic	R	SRA	rd,rs1,rs2
Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt
Arithmetic ADD	R	ADD	rd,rs1,rs2
ADD Immediate	I	ADDI	rd,rsl,imm
SUBtract	R	SUB	rd,rs1,rs2
Load Upper Imm	U	LUI	rd,imm
Add Upper Imm to PC	U	AUIPC	rd,imm
Logical XOR	R	XOR	rd,rs1,rs2
XOR Immediate	I	XORI	rd,rs1,imm
OR	R	OR	rd,rs1,rs2
OR Immediate	I	ORI	rd,rsl,imm
AND	R	AND	rd,rs1,rs2
AND Immediate	I	ANDI	rd,rsl,imm
Compare Set <	R	SLT	rd,rs1,rs2
Set < Immediate	I	SLTI	rd,rsl,imm
Set < Unsigned	R	SLTU	rd,rs1,rs2
Set < Imm Unsigned	I	SLTIU	rd,rsl,imm
Branches Branch =	В	BEQ	rs1,rs2,imm
Branch ≠	В	BNE	rs1,rs2,imm
Branch <	В	BLT	rs1,rs2,imm
Branch ≥	В	BGE	rs1,rs2,imm
Branch < Unsigned	В	BLTU	rs1,rs2,imm
Branch ≥ Unsigned	В	BGEU	rs1,rs2,imm
Jump & Link J&L	J	JAL	rd,imm
Jump & Link Register	I	JALR	rd,rsl,imm
Synch Synch thread	I	FENCE	
Synch Instr & Data	I	FENCE.	.I
<b>Environment</b> CALL	I	ECALL	
BREAK	I	EBREAL	ζ.

· RISC-V指令的一般格式:

op dst, src1, src2

- op = 操作符名称(operator)
- dst = 结果寄存器 (destination)
- src1 = 操作数1 (source 1)
- src2 = 操作数2 (source 2)
- RV32基础整型指令分类:
  - Shift: 移位指令
  - Arithmetic: 算术指令
  - Logical: 逻辑操作
  - Compare: 比较操作
  - Branches: 分支指令
  - Jump & Link: 跳转指令
  - Synch: 同步指令(不涉及)
  - Environment: 环境调用(不涉及)
- RV64I指令在RV32I指令基础上增加了部分移位和算术指令



## RISC-V移位指令

Instructions Name	RISC-V
Shift Left Logical	sll rd, rs1, rs2
Shift Left Logical immediate	slli rd, rs1, shamt
Shift Right Logical	srl rd, rs1, rs2
Shift Right Logical immediate	srli rd, rs1, shamt
Shift Right Arithmetic	sra rd, rs1, rs2
Shift Right Arithmetic immediate	srai rd, rs1, shamt

• Logical shift: 逻辑移位,高位填0

• Arithmetic shift: 算术移位,高位填符号位



## RISC-V算术指令

Instructions Name	RISC-V
Add	add rd, rs1, rs2
Add Immediate	addi rd, rs1, imm
Subtract	sub rd, rs1, rs2
Load Upper Immediate	lui rd, imm
Add Upper Immediate to PC	auipc rd, imm

### lui rd, immediate

x[rd] = sext(immediate[31:12] << 12)

高位立即数加载 (Load Upper Immediate). U-type, RV32I and RV64I.

将符号位扩展的 20 位立即数 immediate 左移 12 位,并将低 12 位置零,写入 x[rd]中。

### auipc rd, immediate

x[rd] = pc + sext(immediate[31:12] << 12)

*PC* 加立即数 (Add Upper Immediate to PC). U-type, RV32I and RV64I. 把符号位扩展的 20 位(左移 12 位)立即数加到 pc 上,结果写入 x[rd]。



## RISC-V逻辑操作指令

Instructions Name	RISC-V
XOR	xor rd, rs1, rs2
XOR Immediate	xori rd, rs1, imm
OR	or rd, rs1, rs2
OR Immediate	ori rd, rs1, imm
AND	and rd, rs1, rs2
AND Immediate	andi rd, rs1, imm



## RISC-V比较指令

Instructions Name	RISC-V
Set <	slt rd, rs1, rs2
Set < Immediate	slti rd, rs1, imm
Set < Unsigned	sltu rd, rs1, rs2
Set < Imm Unsigned	sltiu rd, rs1, imm

slt rd, rs1, rs2

$$x[rd] = (x[rs1] <_s x[rs2])$$

小于则置位(Set if Less Than). R-type, RV32I and RV64I.

比较 x[rs1]和 x[rs2]中的数,如果 x[rs1]更小,向 x[rd]写入 1,否则写入 0。

sltu rd, rs1, rs2

$$x[rd] = (x[rs1] <_{u} x[rs2])$$

无符号小于则置位(Set if Less Than, Unsigned). R-type, RV32I and RV64I.

比较 x[rs1]和 x[rs2], 比较时视为无符号数。如果 x[rs1]更小,向 x[rd]写入 1, 否则写入 0。



## RISC-V分支指令

Instructions Name	RISC-V
Branch =	beq rs1, rs2, imm
Branch ≠	bne rs1, rs2, imm
Branch <	blt rs1, rs2, imm
Branch ≧	bge rs1, rs2, imm
Branch < Unsigned	bltu rs1, rs2, imm
Branch ≥ Unsigned	bgeu rs1, rs2, imm

#### beq rs1, rs2, offset

if (rs1 == rs2) pc += sext(offset)

相等时分支 (Branch if Equal). B-type, RV32I and RV64I.

若寄存器 x[rs1]和寄存器 x[rs2]的值相等, 把 pc 的值设为当前值加上符号位扩展的偏移 offset。

### bltu rs1, rs2, offset

if  $(rs1 <_u rs2)$  pc += sext(offset)

无符号小于时分支 (Branch if Less Than, Unsigned). B-type, RV32I and RV64I.

若寄存器  $\mathbf{x}[rsI]$ 的值小于寄存器  $\mathbf{x}[rs2]$ 的值(均视为无符号数),把 pc 的值设为当前值加上符号位扩展的偏移 offset。



## RISC-V跳转指令

Instructions Name	RISC-V				
Jump and Link	jal rd, imm				
Jump and Link Register	jalr rd, rs1, imm				

jal rd, offset

x[rd] = pc+4; pc += sext(offset)

跳转并链接 (Jump and Link). J-type, RV32I and RV64I.

把下一条指令的地址(pc+4),然后把pc 设置为当前值加上符号位扩展的offset。rd 默认为x1。

jalr rd, offset(rs1)

t = pc+4;  $pc=(x[rs1]+sext(offset))&\sim1$ ; x[rd]=t

跳转并寄存器链接 (Jump and Link Register). I-type, RV32I and RV64I.

把 pc 设置为 x[rsI] + sign-extend(offset),把计算出的地址的最低有效位设为 0,并将原 pc+4 的值写入 f[rd]。rd 默认为 x1。

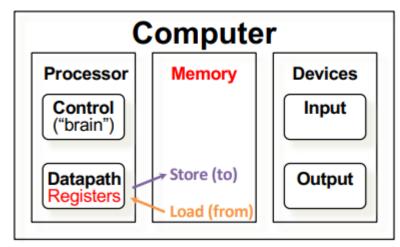


## RISC-V加载存储指令

Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd, rsl, imm
Load I	Byte Unsigned	1	LBU	rd, rs1, imm
	Half Unsigned	I	LHU	rd, rs1, imm
	Load Word	I	LW	rd, rs1, imm
Stores	Store Byte	S	SB	rs1,rs2,imm
Store Halfword		S	SH	rs1,rs2,imm
	Store Word	S	SW	rsl,rs2,imm

Load: 数据从存储器搬移到寄存器;

Store: 数据从寄存器搬移到存储器;



	O	per	י 🔀	4	<b>~!!</b>	$\supset$	-	V	Refe	erence	C	ard		1
Base Inte					I and R					/ Privilege				
Category Name	Fmt		RV32I Base		-	+RV641		Catego		Name	Fint		mnemo	nic .
Shifts Shift Left Logical	R	SLL	rd,rsl,r	52	SLLW ro	,rs1,rs2		Trap Ma	ech-mo	de trap return	R	MRET		
Shift Left Log. Imm.	1	SLLI	rd,rsl,s	hant	SLLIW rd	,rs1,sham	t.	Superv	isor-mo	de trap return	R	SIET		
Shift Right Logical	R	SEL	rd,rsl,r	82	SRLW re	,rs1,rs2		Interru	pt Wait	t for Interrupt	R	WFI		
Shift Right Log. Imm.	1	SELI	rd,rsl,s		SELIW FO	rs1,sham		MMU V	rirtual M	temory FENCE	R	SPENCE	.VMA rs	1.rs
	R	SRA	rd.rsl.r		700	rsl,rs2	~	STATE OF THE PERSON NAMED IN	Annual Property lies	of the 60 I	ALC: UNKNOWN			
Shift Right Arithmetic					1100 Park 1000									ons
Shift Right Arith. Imm.	1	SRAI	rd,rs1,s			,rsl,sham	τ			0 rs,x0,imm)		BEQZ r	s,imm	
Arithmetic ADD	R	ADD	rd,rsl,r	82	ADDW rd	,rsl,rs2				JAL x0, imm)	3	J imm		
ADD Immediate	1	ADDI	rd,rs1,i	mm	ADDIW re	,ral,imm		MoVe	(uses A	DDI rd.rs.0)	R	HV rd,	rs	
SUBtract	R	SUB	rd,rel,r	w2		rs1,rs2		DETurn	funes:	ALR x0,0,ra)	1	RET		
	17.1	28333				_			-	-	_	-	-	
Load Upper Imm	U	LUI	rd,imm		Opt	ional Con	pres	sed (10	5-bit)	Instruction	n Ext	ension	: RV32	C
Add Upper Imm to PC	U	AUIPC	rd, imn		Categor	y Name	Fmt	The second	RV	C		RISC-V	eguivalei	nt
ogical XOR	R	XOR	rd,rsl,r	n2	Loads	Load Word	CL	C.LN	rd'.	rsl', imm	LW	rd',r	sl',imm	+4
XOR Immediate	1	XORI	rd,rsl,i		10	ad Word SP	CI	C.LWSP	rd, i		LW		.imm*4	
OR	R	OR	rd,rsl,r			ad Word SP	CL	C. FLW		rsl', imm	PLW		al',imm	*0
OR Immediate	ī	ORI	rd,rsl,i			t Load Word	CI	C. FLWSP			FLM		.imm*8	
								1-11-11-1						
AND	R	AND	rd,rs1,r			oad Double	CL	C.FLD		ral', imm	FLD		sl',imm	
AND Immediate	I	ANDI	rd,rsl,i			d Double SP	CI	C.FLDSP			FLD		, imm*16	
ompare Set <	R	SLT	rd,rsl,r	82	Stores	Store Word	CS	C.SW	rel'	,rs2',imm	SW	ral',	rs2',im	n*4
Set < Immediate	1	SLTI	rd,rs1,i	mm	Str	ore Word SP	CSS	C.SWSP	rs2,	Smin	SW		p,imm*4	
Set < Unsigned	R	SLTU	rd.rsl.r	82	Roat	Store Word	CS	C.FSW	rsl'	,rs2',imm	PSW	ral',	rs2', im	n*8
Set < Imm Unsigned	1		rd,rsl,i			ore Word SP	CSS	C. PSWSP			FSW		p,imm*8	
Branches Branch =	В	BBO	rs1, rs2,		100000000000000000000000000000000000000	tore Double	CS	C.FSD		,rs2',imm	FSD		rs2', im	
		153527	5 5 5 6 6 7 7 7 7					7 007 000		and the second second				
Branch ≠	В	BME	rsi,rs2,			e Double SP	CSS	C. PSDSP			FSD		p,imm*1	6
Branch <	В	BLT	rs1, rs2,		Arithme		CR.	C. ADD		,rsl	ADD	rd, rd		
Branch ≥	В	BGE	rsl,rs2,	imm	ADD	Immediate	CI	C.ADDI	rd	,imm	ADDI	rd, rd	, inn	
Branch < Unsigned	В	BLTU	rsl,rs2,	inn	ADD S	P Imm * 16	CI	C. ADDII	6SP x0	, imm	ADDI	ap, ap	, imm*16	
Branch ≥ Unsigned	В	BGEU	rs1, rs2,	imm	ADD	SP Imm * 4	CIW	C.ADDI4	SPN rd	i', irun	ADDI	rd'.s	p,imm*4	
ump & Link 38L	1	JAL	rd,inn			SUB	CR	C.SUB		rsl	SUB	rd, rd		
Jump & Link Register	î	JALR	rd,ral,i	tento.		AND	CR	C. AND		l,ral	AND	rd, rd		
NAME AND ADDRESS OF THE OWNER, WHEN PERSON ADDRESS OF THE OWNER, WHEN PERSON AND ADDRESS OF THE OWNER, WHEN	-	PENCE	tu,tmi,i	inen.				C.ANDI						
nch Synch thread	1	- 11 Oct			AND	Immediate	CI	100000000000000000000000000000000000000		i,imm	ANDI	rd, rd		
Synch Instr & Data	1	FENCE.	,I			OR	CR	C.OR		i,rsl	OR	rd, rd		
vironment CALL	1	ECALL			e	Xclusive OR	CR	C. NOR	rd	i,ral	AND	rd, rd	,rsl	
BREAK	1	EBREAR	K.			MoVe	CR	C.NV	rd	rs1	ADD	rd, rs	1,x0	
and the second second second	W.O.				Load	Immediate	CI	C.LI	rd	inm.	ADDI	rd, x0	imm	
or trol Status Regis	ter (	CSR)			Load	Upper Imm	CI	C.LUI		,imm	LUI	rd, im		
Read/Write	I	CSRRM	rd, car,	ral		ift Left Imm	CI	C.SLLI		,imm	SLLI	rd, rd		_
Read & Set Bit	ī	CSRRS	rd.csr.			nt Ari. Imm.	CI	C. SRAI		,imm	SRAI	rd, rd		
Read & Clear Bit	1	CSRRC				t Log. Imm.	CI	C.SRLI						
			rd, car,			s Branch=0				i, irm	SELI	rd, rd		
Read/Write Imm	1		I rd, csr,		Branche			C.BEQZ		1',imm	BBQ		x0,imm	
Red & Set Bit Imm	1		rd, car,		CA YMPIN	Branch≠0	CB	C.BEEZ		l',imm	BNE	rsl',	x0,imm	
Rea 8. Clear Bit Imm	1	CSERCE	I rd, csr,	imm.	Jump	Jump	CJ	C.J	in	nán.	JAL	x0,im	m	
					Ju	mp Register	CR	C.JR	rd	,rsl	JALR	m0,rs	1.0	
					Jump &	Link J&L	CJ	C.JAL	in		JAL	ra, im		
oads Load Byte	1	LB	rd,rs1,	inn.		ink Register	CR	C.JALR	rs		JALR	ra, rs		
Load Halfword	500			77117		Env. BREAK	-	THE REAL PROPERTY.		-	Section Section	THE RESERVE AND ADDRESS OF THE PERSON.	4,0	_
	1	LH	rd, rs1,		aystein		CI	C.EBREA	_		EBREZ			_
Load Byte Unsigned	1	LBU	rd, rsl,			+RV64I				Compress				
Load Half Unsigned	1	LHU	rd, rs1,	imm	LWU re	,rsl,imun		All RV32	C (exce	pt C. JAL, 4 m	ord loa	ds, 4 wor	rd strores	) plus
Load Word	1	LM	rd, rsl,	imm	LD rd	,rsl,imn		ADI	D Word	(C.ADOW)	Los	nd Double	eword (c	.LD)
tores Store Byte	S	SB	ral,ra2	inn	100			ADD to	nm. Wor	nd (C.ADDIW)	Load	Doublew	ord SP /	LDS
Store Halfword	s	320	000000000	W. C. C. C. C.				100000000000000000000000000000000000000		The state of the s			leword (	
	_	SB	rs1,rs2	7.00	780 To 1800			SUBtr	act won	d (c.stew)				
Store Word	S	SW	rsl,rs2			1,rs2,imm							ord SP (	21808
25 32 25315	32		struction		its					it (RVC) In:				
31 27 26 25	24	20	19 15	14 12	11	7 6	0	CR15	14 13			6. 5		1 0
timers	ES	2	rs1	funct3		opeo		122	funct4	rd/i	st l	1	162	op
lmm 11:0			rsl	funct3		opeo		CI fu	met3   i	mm rd/s			ım	op
imm[11:5]	19	2	rs1	funct3			de		met3	lmm			s2	op
imm 12 10:5	13	2	rs1	funct3	lmm 4:1				met3	im	m	1	rd'	op
	imm]				rd	opeo			mct3	imm	rel'	limm	rd"	op
imm		1 11 19:1	12		nl	upeo		CL for	met3	lmm	ral*	lmm	rs2'	op
-			-			-		CS for	met3	offset	rst*	cel	Set	op
									met3		ump tar			op
								1 11	100.752			man file		



## RISC-V CSR指令

Read/Write I CSRRW rd,csr,rs1
Read & Set Bit I CSRRS rd,csr,rs1
Read & Clear Bit I CSRRC rd,csr,rs1
Read/Write Imm I CSRRWI rd,csr,imm
Read & Set Bit Imm I CSRRSI rd,csr,imm
Read & Clear Bit Imm I CSRRCI rd,csr,imm

CSR寄存器只能通过CSR指令访问

Base Inte					I and R				Reference RV Privilege			S	
	Fmt		V321 Ba			+RV641		Categor				nnemo	nic.
Shifts Shift Left Logical	R	SLL	rd,rs1,	rs2	SLLW ro	,rsl,rs2		Trap Ma	ch-mode trap retu	m R	MRET		
Shift Left Log. Imm.	1		rd,rel,		SLLIW rd	i,rsl,sham	rt.		or-mode trap retur		SIET		
Shift Right Logical	R	SEL	rd,rsl,			d, rs1, rs2			t Wait for Interru		WFI		
Shift Right Log. Imm.	1	SELI	rd,rsl,	shant	SELIW TO	,rel,sham	t.	MMU V	rtual Memory FENC	ER	SPENCE.	VMA rs	1, ra
Shift Right Arithmetic	R	SRA	rd,rs1,	rs2	SRAW ro	d,rsl,rs2		Exam	ples of the 60	RV Ps	eudoins	tructi	ons
Shift Right Arith. Imm.	1	SRAI	rd,rs1,	shant	SRAIW ro	i,rsl,sham	r.	Branch =	O (BEC re, xO, im	) J	BEQZ rs	, imm	11.00
rithmetic ADD	R	ADD	rd,rsl,	rs2	ADDW ro	i,rsi,rs2		Jump	(uses JAL x0, im	0 3	J imm		
ADD Immediate	1	ADDI	rd,rs1,	inn	ADDIW re	l, rel, imm		MoVe (	uses ADDI rd, rs, I	) R	HV rd, r	8	
SUBtract	R	SUB	rd,rsl,	rs2	SUBW re	Lrs1,rs2		RETurn /	uses JALR x0,0,r	0 1	RET		
Load Upper Imm	U	LUI	rd, imm	-1073	0.1			-		-		01/22	0
	U	-	rd,imn					sea (16	-bit) Instruction				
Add Upper Imm to PC ogical XOR	R	AUIPC	rd,rsl,	in a D	Loads	Load Word	CL	C.IN	rd',rsl',imm	LW	rd', rs		
XOR Immediate	I		rd.rsl.			ad Word SP	CI	C.LWSP	rd, imm	LW	rd, sp.		
OR Immediate	R	OR	rd,rsl,			ad Word SP		C. FLW	rd',rsl',imm	FLW	rd', rs		
OR Immediate	1	ORI	rd,rs1,			t Load Word	CI	C. FLWSP	rd.imm	FLW	rd, sp,		-0
AND	R	AND	rd,rs1,			Load Double	C	C.FLD	rd', ral', imm	FLM	rd', rs		*16
AND Immediate	I.		rd,rsl			d Double SP	CI	C.FLDSP	rd, inn	FLD	rd, sp,		
ompare Set <	R	SLT	rd,rsl,			Store Word	CS	C.SW	rs1',rs2',imm	SW	ral',ra	27 . Sm	m+4
Set < Immediate	ī		rd,rsl,			ore Word SP		C.SWSP	rs2,imm	SW	rsZ, sp,		
Set < Unsigned	R		rd.rsl.			Store Word		C.FSW	rs1',rs2',imm		ral',ra		
Set < Imm Unsigned	I		rd,rsl,			ore Word SP		C. PSWSP	rs2,imm	FSW	rs2.sp.		
ranches Branch =	В	BEO	rsl,rs			Rore Double	CS	C.FSD	rsl',rs2',imm	FSD	rsl',rs		
Branch #	В	BME	rsi.rs	1210-760		e Double SP		C. FSDSP	re2,imm	FSD	rs2,sp,		
Branch <	В	BLT	rel,re		Arithme		CR	C. ADD	rd,rsl	ADD	rd, rd,		0
Branch ≥	В	BOR	rsl,rs			Immediate		C. ADDI	rd,imm	ADDI			
Branch < Unsigned	В	ar-unu	rsl,rs			P Imm * 16			SP x0,imm	ADDI			
Branch ≥ Unsigned	В	BGEU	rsl,rs			SP Imm * 4			DN rd', imm	ADDI			
ump & Link 38L	1	JAL	rd,imm	· ranes	nee	SUB		C.SUB	rd,rsl	SUB	rd, rd,		
Jump & Link Register	í	JALR	rd,ral,	1mm		AND	CR	C. AND	rd,rsl	AND	rd, rd,		
ynch Synch thread	I	FENCE		-210	AND	Immediate	a	C.ANDI	rd, inm	ANDI			
Synch Instr & Data	1	FENCE.	т.		575555	OR	CR	C.OR	rd.rsl	OR	rd.rd.		
nvironment CALL	i	ECALL:	4.			Xclusive OR	CR	C. KOR	rd,ral	AND	rd,rd,		
BREAK	1	EBREAR				MoVe		C.NV	rd,rs1	ADD	rd, rsl,		
					Load	Immediate	CI	C.LI	rd, imm	ADDI			
Control Status Regist	ter (	CSR)			Load	Upper Imm	CI	C.LUI	rd, imm	LUI	rd, imm		
Read/Write	1	CSRRW	rd, car	rsl	Shifts Sh	ift Left Imm	a	C.SLLI	rd,imm	SLLI			
Read & Set Bit	1	CSRRS	rd, csi	rsl	Shift Righ	ht Ari. Imm.	CI	C.SRAI	rd, imm	SRAI	rd, rd,		
Read & Clear Bit	1	CSRRC	rd, car	,rsl		t Log. Imm.	CI	C.SRLI	rd,imm	SELI	rd, rd,	imm	
Read/Write Imm	1	CSRRWI	rd, csi	, imm	Branche	s Branch=0		C.BEQZ	rsl',imm	BBQ	ral',x		
Read & Set Bit Imm	1	CSERSI	rd, car	, imn		Branch≠0		C.BHEZ	ral',imm	BNE	rs1',x0	, imm	
Read & Clear Bit Imm	1	CSERCI	rd, csi	, imm	Jump	Jump	CJ	C.J	imm	JAL	x0,imm		
						mp Register	CR	C.JR	rd,rsl	JALR	m0, rs1,	0	
					Jump &		Cl	C.JAL	imm	JAL	ra,imm	121	
oads Load Byte	1	LB	rd,rs	i, imm		ink Register	CR	C.JALR	rsl	JALR	ra, rsl,	.0	
Load Halfword	1	LH	rd, rs	, inn	System	Env. BREAK	CI	C.EBREAR		EBRE	AK		
Load Byte Unsigned	1	LBU	rd, rs	, imm		+RV64I		Opt	ional Compres			RV64	4C
Load Half Unsigned	1	LHU	rd, rs	, imm	LWU re	,rel,imm			(except C. JAL, 4				
Load Word	1	LM	rd, rs	, imm	LD re	i, rsl, imm		ADD	Word (C.ADDW)	Lo	ad Doubley	word (c	LD)
tores Store Byte	S	SB	ral,r	2,imm	***			ADD Im	m. Word (C.ADDIW)	Load	Doublewon	d SP (c	c. Logs
Store Halfword	S	SB		2,imm				0.000	ct Word (c.stew)		ore Doubles	200 P.	
Store Word	s	SW		2,imm	SD re	1,rs2,imm			,		Doublewor		
Store Word	_	-		Forma		*************		_	16-bit (RVC) I				0.00000
31 27 26 25 2	4	20	19 15	14 12	11	7 6	0	10.0	4 13 12 11 10	2 8 7	6 b 4	1 2	1 0
funct7	ES	2	rsl	funct3		opec	de	CR		/nst	ns2		Op
imm 11:0			rsl	funct3	rd	opec			et3 imm rd	/rsl	lme		op
	19		rs1	funct3	limm 45				et3 lmm		ns2		op
imm[11:5]		9	rs1	funct3	lmm 4:1	11 opec				nm		rd'	op
imm[12]10:5]	13												
imm[12]30:5]	mm 3	31:12	107	1	rd	орес			ct3 imm	rel.	imm	nd	op
imm[12]30:5]	mm 3		2]	VI	rd rd	opec		oc fun	et3 imm et3 imm et3 offset	ral'	imm imm	rs2'	op op



## RISC-V伪指令

- 伪指令不是真实存在的RISC-V指令,它只是为了方便程序员理解
- 伪指令会被编译器翻译成真实的RISC-V指令
  - 伪指令move

例子: mv dst, reg1 基础指令: addi dst, reg1, 0

• 伪指令no operation

例子: nop

基础指令: addi x0, x0, 0

伪指令load address

例子: la dst, label

基础指令: auipc dst, <offset to label>

• 伪指令jump

例子: j offset

基础指令: jal x0, offset

• 更多伪指令可以看《RISC-V手册》第44、45页



## 一个简单的例子

```
# Fibonacci Sequence
main:
                                                       注释使用#符号
       add t0, x0, x0
       addi t1, x0, 1
            t3, n
       la
            t3, 0(t3)
       lw
fib:
      beq t3, x0, finish
      add
           t2, t1, t0
            t0, t1
      mv
           t1, t2
      mν
      addi t3, t3, -1
            fib
finish:
      addi a0, x0, 1
      addi a1, t0, 0
      ecall # print integer ecall
      addi a0, x0, 10
      ecall # terminate ecall
```



## 一个简单的例子

```
# Fibonacci Sequence
main: <
                                                      标签用于标记代码段
       add t0, x0, x0
       addi t1, x0, 1
            t3, n
       la
       lw
            t3_0(t3)
fib: 🔺
      beq
            t3, x0, finish
      add
            t2, t1, t0
            t0, t1
      mv
            t1, t2
      mν
      addi t3, t3,
finish:
      addi a0, x0, 1
      addi a1, t0, 0
      ecall # print integer ecall
      addi a0, x0, 10
      ecall # terminate ecall
```



## 一个简单的例子

### # Fibonacci Sequence

#### main:

add t0, x0, x0 addi t1, x0, 1 la t3, n lw t3, 0(t3)

#### fib:

beq t3, x0, finish add t2, t1, t0 mv t0, t1 mv t1, t2 addi t3, t3, -1 j fib

### finish:

addi a0, x0, 1 addi a1, t0, 0 ecall # print integer ecall addi a0, x0, 10 ecall # terminate ecall

### a[n] = a[n-1] + a[n-2], n>=2, a[0]=0, a[1]=1

```
t0=x0+x0 ===> t0=0;
t1=x0+1 ===> t1=1;
t3=address_of(n) ===> 此时t3指向n变量的地址;
t3=n ===> t3=n;
```

```
c如果t3为0, 说明已经将a[n] 计算出来, 可以结束循环t2=t1+t0===> 斐波那契计算公式t0=t1===> 更新t0t1=t2===> 更新t1t3=t3-1===> 计数器减1开始下一次循环
```

```
a0=x0+1 ===> a0=1
a1=t0+0 ===> a1=t0
调用printf函数
a0=x0+10 ===> a0=10
调用exit函数
```



### RISC-V汇编语法



## 汇编器指示符

Directive	Description
.text	Subsequent items are stored in the text section
.data	Subsequent items are stored in the data section
.bss	Subsequent items are stored in the bss section
.section .foo	Subsequent items are stored in the section name .foo
.align	Align to power of 2
.balign	byte align
.global sym	Declare that label sym is global
.string "str"	Store the string str in memory and null-terminate it
.type	Accepted for source compatibility
.equ	Constant definition
.macro	Begin macro definition \argname to substitute
.endm	end macro definition



## 汇编器指示符

Directive	Description
.byte b1,, bn	Store the n 8-bit quantities in successive bytes of memory
.half w1,, wn	Store the n 16-bit quantities in successive memory half-words
.word w1,, wn	Store the n 32-bit quantities in successive memory words
.dword w1,, w1	Store the n 64-bit quantities in successive memory doublewords
.float f1,, fn	Store the n single-precision floating-point numbers in successive memory words
.double d1,, dn	Store the n double-precision floating-point numbers in successive memory words



## 汇编器重定位表达式

Directive	Description	Instruction
%hi(symbol)	Absolute (HI20)	lui
%lo(symbol)	Absolute (LO12)	load, store, add
%pcrel_hi(symbol)	PC-relative (HI20)	auipc
%pcrel_lo(label)	PC-relative (LO12)	load, store, add
%tprel_hi(symbol)	TLS LE "Local Exec"	lui
%tprel_lo(symbol)	TLS LE "Local Exec"	load, store, add
%tprel_add(symbol)	TLS LE "Local Exec"	add
%tls_ie_pcrel_hi(symbol) *	TLS IE "Initial Exec" (HI20)	auipc
%tls_gd_pcrel_hi(symbol) *	TLS GD "Global Dynamic" (HI20)	auipc
%got_pcrel_hi(symbol) *	GOT PC-relative (HI20)	auipc



## 链接器的作用

- 符号解析(symbol resolution)
  - 目标文件定义和引用符号,每个符号对应于一个函数、一个全局变量或一个静态变量(即C语言中任何以 static属性声明的变量)。符号解析的目的是将每个 符号引用正好和一个符号定义关联起来。
- 重定位(relocation)
  - 编译器和汇编器生成从地址O开始的代码和数据节 (section)。链接器通过把每个符号定义与内存位置关联起来,从而重定位这些节,然后修改所有对这些符号的引用,是的它们指向这个内存位置。链接器使用汇编器产生的重定位条目(relocation entry)的详细信息,不加甄别地执行这样的重定位。



## 一个不简单的例子

```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

链接器以一组可重定位目标文件和命令行参数作为输入,生成一个完全链接的、可以加载和运行的可执行目标文件作为输出。输入的可重定位目标文件由各种不同的代码和数据节组成,每一节都是一个连续的字节序列。

```
.text
                              # Directive: enter text section
.align 2
                              # Directive: align code to 2^2 bytes
.global main
                              # Directive: declare global symbol main
                              # label for start of main
main:
                              # allocate stack frame
  addi sp. sp. -16
        ra, 12(sp)
                              # save return address
   SW
        a0, %hi(string1)
                              # compute address of
  lui
  addi a0, a0, %lo(string1) #
                                  string1
  lui a1, %hi(string2)
                              # compute address of
  addi al, al, %lo(string2) #
                                  string2
                              # call function printf
  call printf
         ra, 12(sp)
                              # restore return address
  lw
                              # deallocate stack frame
  addi sp, sp, 16
  li
         a0, 0
                              # load return value 0
   ret
                              # return
                              # Directive: enter read-only data section
.section .rodata
.balign 4
                              # Directive: align data section to 4 bytes
                              # label for first string
string1:
  .string "Hello, %s!\n"
                              # Directive: null-terminate string
                              # label for second string
string2:
  .string "world"
                              # Direcvive: null-terminate string
```

左图中有两个数据标签 (string1和string2)和两个代码标签(main和printf)需要重定位,由于在单个32位指令中很难指定一个32位的地址,RV32I的链接器通常需要为每个标签调整两条指令。



## 一个不简单的例子

```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

链接器以一组可重定位目标文件和命令行参数作为输入,生成一个完全链接的、可以加载和运行的可执行目标文件作为输出。输入的可重定位目标文件由各种不同的代码和数据节组成,每一节都是一个连续的字节序列。

### 00000000 <main>:

2c: 00008067

```
0: ff010113
               addi
                      sp, sp, -16
    00112623
                      ra, 12(sp)
               SW
    00000537
                      a0,0x0
               lui
    00050513
                      a0, a0
               mv
    000005b7
                      a1,0x0
10:
               lui
14: 00058593
                      a1, a1
               mv
               auipc ra,0x0
    00000097
    000080e7
               jalr
                      ra
20: 00c12083
                      ra, 12(sp)
               lw
    01010113
               addi
                      sp,sp,16
28:
    00000513
               li
                      a0,0
```

ret

左图是Hello World程序经过汇编后的重定位目标文件 hello.o,位置0x8到0x1c这6条指令的地址字段为0,将在后面由链接器填充。目标文件的符号表记录了链接器所需要的标签和地址。从左图可以看出:数据标签需要调整lui和addi,代码标签需要调整auipc和jalr.



## 一个不简单的例子

```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

链接器以一组可重定位目标文件和命令行参数作为输入,生成一个完全链接的、可以加载和运行的可执行目标文件作为输出。输入的可重定位目标文件由各种不同的代码和数据节组成,每一节都是一个连续的字节序列。

### 000101b0 <main>:

101d8: 00008067 ret

```
101b0: ff010113 addi sp,sp,-16
                     ra, 12(sp)
101b4: 00112623 sw
101b8: 00021537 lui
                     a0,0x21
101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
101c0: 000215b7 lui
                     a1,0x21
101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
101c8: 288000ef jal
                     ra,10450 <printf>
101cc: 00c12083 lw
                     ra, 12(sp)
101d0: 01010113 addi sp, sp, 16
101d4: 00000513 li
                     a0,0
```

左图是Hello World程序经过链接 后产生的a.out文件。与之前的重 定位目标文件hello.o相比,有两 处修改: (1) main函数代码被 赋予了一个新的地址字段; (2) 位置0x8到0x1c这6条指令的地址 字段被重新赋值。



### Q & A



# 3编译原理的基础知识

潘志铭



### 编译器的基础概念



## 编译器的作用

- 什么是编译器?
  - 编译器是一种工具,将一种语言编写的软件转换为另一种语言;
  - 编译器将某种语言编写的程序作为输入,产生一个等价的程序作为输出;
  - C语言属于典型的编译型语言
- 什么是解释器?
  - •解释器将一种可执行程序作为输入,对外输出该程序执行后的结果;
  - Python语言属于解释性语言



## 编译器的组成

- 前端
  - 预处理, 词法分析, 语法分析
- 中间表示
  - 抽象语法树,中间层代码形式,中间表示优化
- 后端
  - 目标机器描述,指令选择,指令调度,寄存器分配, 汇编代码生成





## 流行的编译器版本---GCC

- ·什么是GCC?
  - GCC 是 GNU 开发的程序语言编译器。它是一组在 GNU 通用公共许可证 (GPL) 和 GNU 宽松通用公共许可证 (LGPL) 下发布的免费软件。它是 GNU 和 Linux 系统的官方编译器,也是编译和创建其他 UNIX 操作系统的主要编译器。

### GCC, the GNU Compiler Collection

The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Ada, Go, and D, as well as libraries for these languages (libstdc++,...). GCC was originally written as the compiler for the GNU operating system. The GNU system was developed to be 100% free software, free in the sense that it respects the user's freedom.

We strive to provide regular, high quality releases, which we want to work well on a variety of native and cross targets (including GNU/Linux), and encourage everyone to contribute changes or help testing GCC. Our sources are readily and freely available via Git and weekly snapshots.

Major decisions about GCC are made by the steering committee, guided by the mission statement.



## 流行的编译器版本---LLVM

- ·什么是LLVM?
  - LLVM 包含一系列模块化的编译器组件和工具链。准确来说,它本身不是一个编译器,而是一个编译器框架,用于开发编译器的前端和后端。Clang 是一个支持 C/C++/Objective-C/Objective-C++的编译器,它基于LLVM 用 C++ 构建,并在 Apache 2.0 许可下发布。

### The **LLVM** Compiler Infrastructure

#### Site Map:

Overview
Features
Documentation
Command Guide
FAQ
Publications
LLVM Projects
Open Projects
LLVM Users
Bug Database
LLVM Logo
Blog
Meetings

#### LLVM Overview

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project.

LLVM began as a <u>research project</u> at the <u>University of Illinois</u>, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an <u>umbrella project consisting of a number of subprojects</u>.

#### Latest LLVM Release!

8 July 2021: LLVM 12.0.1 is now available for download! LLVM is publicly available under an open source <u>License</u>. Also, you might want to check out <u>the new features</u> in Git that will appear in the next LLVM release. If you want them early, download LLVM through anonymous Git.



### 编译原理的基础概念

--- 编译器前端



## 词法分析

- 词法单词
  - 词法分析器以字符流作为输入,生成一系列名字,关键字和标点符号,同时抛弃单词之间的空白符合注释
  - 词法单词是字符组成的序列,它可以看成是程序设计语言的文法单位,并且可以归为有限的几组单词类型。

```
ID
        foo n14 last
                                                             /* try again */
                                      注释
        73 0 00 515 082
NUM
                                                             #include<stdio.h>
                                      预处理命令
REAL
        66.1 .5 10. 1e67 5.5e-10
                                                             #define NUMS 5 , 6
                                      预处理命令
IF
        i f
                                                             NUMS
                                      12:
COMMA
                                      空格符、制表符和换行符
NOTEQ
LPAREN
RPAREN
```



• 词法单词

```
float match0(char *s) /* find a zero */
{if (!strncmp(s, "0.0", 3))
  return 0.;
}
```

FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN BANG ID(strncmp) LPAREN LBRACE IF LPAREN ID(s) COMMA STRING(0.0) COMMA NUM(3)RPAREN RPAREN RETURN REAL(0.0) SEMI RBRACE EOF



- 正则表达式
  - 为了得到一个简单可读性好的词法分析器。我们将用 正则表达式的形式语言来指明词法单词,用确定的有 限自动机来实现词法分析器,并用数学的方法将两者 联系起来。
  - 一种语言是字符串组成的集合,字符串时符号的有限序列。符号本身来自有限字母表。



- 正则表达式
  - 使用正则表示用有限的描述来指明无限的语言。每个正则表达式代表一个字符串的集合
  - 符号(symbol) 可选(alternation) 联结(concatenation) ε(epsilon) 重复(repetition)

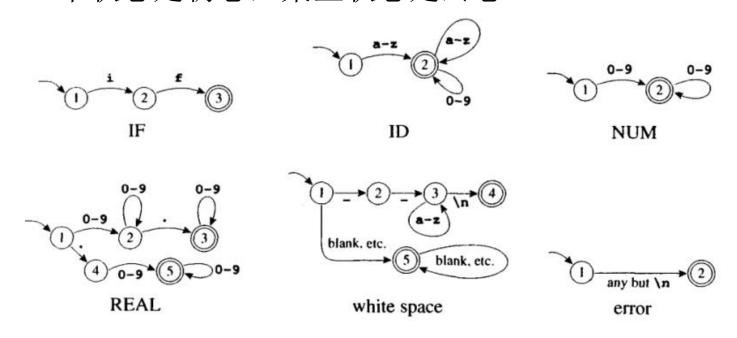
 $(0|1)^* \cdot 0$  由 2 的倍数组成的二进制数。  $b^*(abb^*)^*(ai\epsilon)$  由 a 和 b 组成,但 a 不连续出现的字符串。  $(a|b)^* aa(a|b)^*$  由 a 和 b 组成,且有连续出现的 a 的字符串。



- 正则表达式
  - 词法分析器使用下面两条规则消除二义性
  - 最长匹配
    - 初始输入子串中,取可与任何正则表达式匹配的那个最长的字符串作为下一个单词。例如: **if8是 一个标识符。**
  - 规则优先
    - 对一个特定的最长初始子串,第一个与之匹配的正则表达式决定了这个子串的单词类型。例如,**if是一个保留字**。

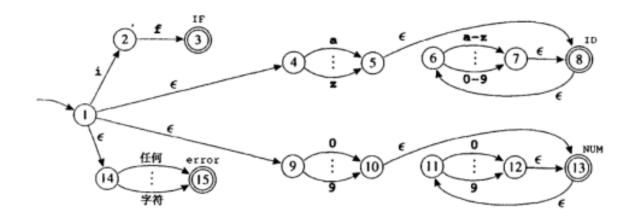


- 有限自动机
  - 有限自动机有一个有限状态集合和一些从一个状态通向另一个状态的边,每条边上标记有一个符号;其中一个状态是初态,某些状态是终态。





- 非确定有限自动机
  - 确定的有限自动机(DFA)不会有从同一状态触发的两条边标记有相同的符号。
  - 分确定的有限自动机(NFA)是一种需要对从一个状态 触发的多条标有相同符号的边进行选择的自动机。
  - 可以很容易将一个正则表达式转换成一个NFA





### 词法分析器

• Lex: 词法分析器的生成器

```
% {
/* C Declarations: */
#include "tokens.h"
                       /* definitions of IF, ID, NUM. ... */
#include "errormsg.h"
union {int ival; string sval; double fval;} yylval;
int charPos=1;
#define ADJ (EM_tokPos=charPos, charPos+=yyleng)
8}
/* Lex Definitions: */
digits [0-9]+
ક્ર ક્ર
/* Regular Expressions and Actions: */
if
                          {ADJ; return IF;}
[a-z][a-z0-9]*
                          {ADJ; yylval.sval=String(yytext);
                            return ID; }
{digits}
                        {ADJ; yylval.ival=atoi(yytext);
                            return NUM; }
({digits}"."[0-9]*)|([0-9]*"."{digits})
                                                {ADJ;
                            yylval.fval=atof(yytext);
                            return REAL; }
("--"[a-z]*"\n")|(" "|"\n"|"\t")+
                                       {ADJ;}
                          {ADJ; EM_error("illegal character");}
```



- 上下文无关文法
  - 有限自动机缺少递归的表示方法。
  - 上下文无关文法(context-free grammar)以说明的方式来定义语法。

```
1 S \rightarrow S; S

2 S \rightarrow id := E

3 S \rightarrow print (L)

4 E \rightarrow id

5 E \rightarrow num

6 E \rightarrow E + E

7 E \rightarrow (S, E)

8 L \rightarrow E

9 L \rightarrow L, E
```



- 上下文无关文法
  - 推导有限自动机从开始符号触发对其右边的每一个非 终结符,用非终结符对应的产生式中的任一右部来替 换它。

```
<u>s</u>
s; <u>s</u>
                                     最左推导:一种总是扩展最
\underline{S}; id := E
                                     左边非终结符的推导。
id := E; id := E
id := num ; id := E
id := num ; id := E + E
id := num ; id := \underline{E} + (S, E)
                                     最右推导:一种总是扩展最
id := num : id := id + (S, E)
                                     右边非终结符的推导。
id := num : id := id + (id := E, E)
id := num ; id := id + (id := E + E, \underline{E})
id := num ; id := id + (id := E + E , id)
id := num ; id := id + (id := num + E, id)
id := num; id := id + (id := num + num, id)
```



- 预测分析
  - 递归下降分析也称预测分析,适合于,每个子表达式的第一个终结符号能够为产生式的选择提供足够多的信息。
  - 有时使用语法分析器生成工具并不方便,预测分析器的有点就在于其算法简单,可以用它手工构造分析器。



### • 预测分析

}}

```
S \rightarrow \text{ if } E \text{ then } S \text{ else } S
S \rightarrow \text{ begin } S L
S \rightarrow \text{ print } E
L \rightarrow \text{ end}
L \rightarrow \text{ ; } S L
E \rightarrow \text{ num } = \text{ num}
```

```
构造预测分析表
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};
extern enum token getToken(void);
enum token tok:
                                                         2. 消除多重定义
void advance() {tok=getToken();}
void eat(enum token t) {if (tok==t) advance(); else error();}
void S(void) (switch(tok) {
                                                         3. 递归下降分析器
       case IF:
                  eat(IF); E(); eat(THEN); S();
                               eat (ELSE); S(); break;
       case BEGIN: eat(BEGIN); S(); L(); break;
       case PRINT: eat(PRINT); E(); break;
       default:
                  error():
void L(void) {switch(tok) {
       case END: eat(END); break;
       case SEMI: eat(SEMI); S(); L(); break;
       default:
                 error();
```



- 预测分析
- 1. 构造预测分析表
- 2. 消除多重定义
- 3. 递归下降分析器

$$S \rightarrow E$$
\$

 $T \rightarrow F T'$ 
 $E \rightarrow T E'$ 
 $T' \rightarrow *F T'$ 
 $E' \rightarrow + T E'$ 
 $E' \rightarrow - T E'$ 
 $T' \rightarrow /F T'$ 
 $F \rightarrow \text{num}$ 
 $F \rightarrow (E)$ 
 $F \rightarrow (E)$ 



- LR分析
  - LR(k): 从左至右分析、最右推导、超前查看k个单词

```
栈
                                                                                         动作
                      a := 7 ; b := c + (d := 5 + 6)
                                                                                     移进
1 ida
                                                                                    移进
                              7; b := c + (d := 5 + 6, d)
                                                                                    移进
1 id4 :=6
1 id4 :=6 num10
                                 ; b := c + (d := 5 + 6, d)
                                                                                    id_4 :=_6 E_{11}
                              ; b := c + (d := 5 + 6 , d)
                                                                                    妈约S \rightarrow id:=E
                                 ; b := c + (d := 5 + 6, d)
1 52
                                                                                    移进
1 52:3
                                   b := c + (d := 5 + 6, d)
                                                                                    移进
1 S2 :3 id4
                                                                                    移进
                                                                                    移进
1 S_2 : 3 id_4 := 6
1.52:3 id_4:=6 id_{20}
                                                                                    归约E \rightarrow id
_{1}S_{2}; _{3} id<sub>4</sub> := _{6}E_{11}
                                                                                    移进
+ S_2 : 3 id_4 := 6 E_{11} + 16
                                                                                    移进
1.52:3 \text{ id}_4 := 6 E_{11} + 16 (8)
                                                                                    移进
1.S_2:3 id_4:=6 E_{11}+16 (8 id_4)
                                                                                    移进
+ S_2 : 3 id_4 := 6 E_{11} + 16 (8 id_4 := 6)
                                                                                    移进
+S_2:3 id<sub>4</sub> := 6 E_{11} + 16 (g id<sub>4</sub> := 6 num<sub>10</sub>
                                                                                    1.S_2:3 id_4:=6 E_{11}+16 (8 id_4:=6 E_{11})
                                                              + 6 , d) $
                                                                                    移进
_{1}S_{2}:_{3}id_{4}:=_{6}E_{11}+_{16}(_{8}id_{4}:=_{6}E_{11}+_{16})
                                                                                    移进
1 S_2 : 3 id_4 := 6 E_{11} + 16 (8 id_4 := 6 E_{11} + 16 num_{10})
                                                                                    归约E \rightarrow num
_{1}S_{2}:_{3}id_{4}:=_{6}E_{11}+_{16}(_{8}id_{4}:=_{6}E_{11}+_{16}E_{17})
                                                                                    99E \rightarrow E + E
1.52:3 \text{ id}_4 := 6.E_{11} + 16.(8 \text{ id}_4 := 6.E_{11})
                                                                                    妈约 S \rightarrow id:=E
```



### • 分析器生成器

```
int yylex(void);
                                                      void yyerror(char *s) { EM_error(EM_tokPos, "%s", s); }
1 P \rightarrow L
                                                       %token ID WHILE BEGIN END DO IF THEN ELSE SEMI ASSIGN
2 S \rightarrow id := id
                                                       %start prog
                                      7 L \rightarrow S
S \rightarrow \text{while id do } S
                                                       ક્ર ક્ર
                                      8 L \rightarrow L; S
4 S \rightarrow \text{begin } L \text{ end}
5 S \rightarrow \text{if id then } S
                                                      prog: stmlist
6 S \rightarrow \text{if id then } S \text{ else } S
                                                       stm : ID ASSIGN ID
                                                            | WHILE ID DO stm
                                                            | BEGIN stmlist END
                                                            IF ID THEN stm
                                                            IF ID THEN stm ELSE stm
```

stmlist : stm

| stmlist SEMI stm

% {



### • 概述

- 语法正确的输入程序仍然可能包含严重的错误,导致 编译无法完成。为了检查这样的错误,编译器需要进 行更深层的检查,其中设计每条语句放到实际的上下 文中进行考虑。
- 为积累进一步转换需要的上下文知识,编译器必须开发出一些方法,从语法之外的视角来考察程序。



- 类型系统
  - 大多数程序设计语言都将一组性质关联到每个数据值, 这些性质的集合成为值的类型。与上下文无关语法相 比,利用类型系统可以在更精确的层次上规定程序的 行为。
  - 1. 确保运行时的安全性
  - 2. 提高表达力
  - 3. 生成更好的代码
  - 4. 类型检查



### • 属性语法

 属性语法的是用于上下文相关分析的一种形式化机制。 属性语法包括一个上下文无关语法,外加一组规定 了某些计算的规则。每个规则都通过其他属性的值 定义了一个值或属性。规则将属性关联到一个特定的 语法符号,出现在语法分析树中的每个语法符号实例 都有一个对应的属性实例。规则是功能性的,没有 蕴涵特定的求值次序,且唯一地定义了每个属性值。



### • 属性语法

ſ	Number	$\rightarrow$	Sign List	
	Sign	$\rightarrow$	+	$T = \{+, -, 0, 1\}$
$P = \begin{cases} 1 & \text{if } P = \\ 1 & \text{if } P = \end{cases} \end{cases} \end{cases} \end{cases} \end{cases}} \end{cases}$	List	 → 	List Bit	NT = {Number, Sign, List, Bit}
	Bit	1	0	$S = \{Number\}$

符号	属	性
Number	value	
Sign	negative	
List	position.	value
Bit	position,	value

	产生式	属性规则
1	Number → Sign List	List.position ← 0  if Sign.negative  then Number.value ← - List.value else Number.value ← List.value
2	Sign → +	Sign.negative ← false
3	$Sign \rightarrow -$	Sign.negative ← true
4	List → Bit	Bit.position ← List.position List.value ← Bit.value
5	$List_0 \rightarrow List_1$ Bit	$List_1$ . $position \leftarrow List_0$ . $position + 1$ $Bit$ . $position \leftarrow List_0$ . $position$ $List_0$ . $value \leftarrow List_1$ . $value + Bit$ . $value$
6	$Bit \rightarrow 0$	Bit.value ← 0
7	$Bit \rightarrow 1$	Bit. value ← 2Bit. position



### • 属性语法

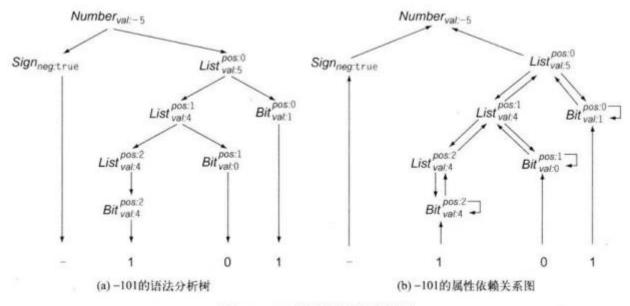


图4-6 -101的属性化语法分析树



### Q & A



# 4 RISC-V工具链的开发

潘志铭



## 为什么要进行工具链开发?

- RISC-V指令集的特点是什么?
  - 开源; 模块化; 可扩展性强

- 什么情况下需要设计自定义指令?
  - 特定应用场景:如PS5游戏加速指令
- 硬件实现自定义指令后,软件如何使用?
  - · 修改GNU工具链,增加对自定义指令的支持



## 如何进行工具链开发?

- 1. 描述自定义指令
- 2. 确定指令码
- 3. 为binutils增加自定义指令
- 4. 为gcc增加自定义指令
- 5. 在spike上增加自定义指令
- 6. 重新构建toolchain和spike
- 7. 测试toolchain和spike



### 描述自定义指令

·增加一条乘加指令mac,具体描述如下:

汇编指令	mac rd, rs1, rs2
软件模型	rd = rd + rs1 * rs2
例子	mac a2, a0, a1



• 依据 RISC-V SPEC, mac 属于 R-Type 指令

31	27	26	25	24		20	19	15	14	12	11	7	6	0	
	funct7				rs2		rs1		fun	ct3	1	rd	opc	ode	R-type
	in	nm[	11:0	)]			rs1		fun	ct3	1	d	opc	ode	I-type
i	mm[11:5	5]			rs2		rs1		fun	ct3	imn	n[4:0]	opc	ode	S-type
in	m[12 10	:5]			rs2		rs1		fun	ct3	imm[	4:1[11]	opc	ode	B-type
	imm[31:12]								1	rd	opc	ode	U-type		
	imm[20 10:1 11 19:12]								1	d	opc	ode	J-type		

B-type U-type J-type

31	25 24	20 19	15 14 12 11	7 6	0
funct7	rs2	rs1	funct3	rd	OPCODE
7	5	5	3	5	7



### • 确定OPCODE

RISC-V SPEC规定了四种自定义指令的OPCODE,分别是7'b000\_1011, 7'b010\_1011, 7'b101\_1011, 7'b111\_1011, 这里我们选用CUSTOM0编码 7'b000\_1011

inst[4:2	000	001	010	011	100	101	110	111
inst[6:5	Π							(> 32b)
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/rv128	$\geq 80b$

31	25 2	4	20 19		15 14	12 11	;	7 6	0	
funct	7	rs2	I	rs1	func	t3	rd	I	OPCODE	
7		5		5	3		5		7	



### • 确定funct3

为了进一步规范自定义指令码格式,我们可以约定funct3的编码格式: 3'b010表示只使用了RS1寄存器,3'b011表示只使用了RS1和RS2 寄存器,3'b100表示只使用了RD寄存器,3'b110表示只使用RD和RS1寄存器,而3'b111则表示RD、RS1和RS2寄存器都用到了,所以这里的funct3应该是3'b111.

funct3	rd	rs1	rs2	Symbol
3'b000	0	0	0	CUSTOMX
3'b001	0	0	1	保留
3'b010	0	1	0	CUSTOMX_RS1
3'b011	0	1	1	CUSTOMX_RS1_RS2
3'b100	1	0	0	CUSTOMX_RD
3'b101	1	0	1	保留
3'b110	1	1	0	CUSTOMX_RD_RS1
3'b111	1	1	1	CUSTOMX_RD_RS1_RS2



• 最终确定的指令码

由于funct7没有特别的约束,我们这里选定为7'b101\_0111.

31 2	25 24	20 19	15 14 12 11	7 6	0
funct7	rs2	rs1	funct3	rd	OPCODE
7'b1010111 7	L RS2		3'b111 3		7'b001011 7



## 为binutils增加自定义指令

• 修改riscv-opc.c文件和riscv-opc.h文件:

```
diff --qit a/include/opcode/riscv-opc.h b/include/opcode/riscv-opc.h
index 7bdc7e4..c68d1c7 100644
--- a/include/opcode/riscv-opc.h
+++ b/include/opcode/riscv-opc.h
@@ -644.6 +644.9 @@
#define MATCH_VGHASH_V 0x4a0fa057
 #define MASK_VGHASH_V 0xfe0ff07f
+#define MATCH_MAC 0xae00700b
+#define MASK_MAC 0xfe00707f
/* Temporary Load/store encoding info
MOP load
 00 unit-stride VLE<EEW>, VLE<EEW>FF, VL<nf>RE<EEW> (nf = 1, 2, 4, 8)
diff --git a/opcodes/riscv-opc.c b/opcodes/riscv-opc.c
index 43714eb..26a3db2 100644
--- a/opcodes/riscv-opc.c
+++ b/opcodes/riscv-opc.c
@@ -897.6 +897.7 @@ const struct riscv_opcode riscv_opcodes[] =
{"divuw".
             64, INSN_CLASS_M, "d,s,t", MATCH_DIVUW, MASK_DIVUW,
match_opcode, 0 },
              64, INSN_CLASS_M, "d,s,t", MATCH_REMW, MASK_REMW,
{"remw",
match_opcode, 0 },
{"remuw",
             64, INSN_CLASS_M, "d,s,t", MATCH_REMUW, MASK_REMUW,
match_opcode, 0 },
            INSN_CLASS_M, "d,s,t", MATCH_MAC, MASK_MAC, match_opcode,
+{"mac",
0 },
/* Half-precision floating-point instruction subset */
{"flh",
               0, INSN_CLASS_F_AND_ZFH, "D,o(s)", MATCH_FLH, MASK_FLH,
match_opcode, INSN_DREF|INSN_2_BYTE },
```

汇编器需要定义一些宏来确定指令编码,其中有两个宏是一定要定义的,分别是 MATCH 和 MASK,其中MATCH宏定义用来识别指令码,MASK宏定义用来获取指令码,在生成指令码时,汇编器会这样使用这些宏定义:

 $((insn \land MATCH) \& MASK) == 0.$ 



# 为gcc增加自定义指令

### • 修改riscv.md文件:

```
diff --git a/gcc/config/riscv/riscv.md b/gcc/config/riscv/riscv.md
index 8cfac79..c1b72ee 100644
--- a/gcc/config/riscv/riscv.md
+++ b/gcc/config/riscv/riscv.md
@@ -750,6 +750,25 @@
  [(set_attr "type" "imul")
   (set_attr "mode" "SI")])
+(define_insn "macsi3"
+ [(set (match_operand:SI
                                    0 "register_operand" "=r")
        (plus: SI (mult:SI (match_operand:SI 2 "register_operand" " r")
                           (match_operand:SI 3 "register_operand" " r"))
                   (match_operand: SI 1 "register_operand" "0")))]
  "TARGET_MUL"
  "mac\t%0,%2,%3"
+ [(set_attr "type" "imul")
   (set_attr "mode" "SI")])
+(define_insn "macdi3"
+ [(set (match_operand:DI
                                   0 "register_operand" "=r")
        (plus: DI (mult:DI (match_operand:DI 2 "register_operand" " r")
                           (match_operand:DI 3 "register_operand" " r"))
                   (match_operand: DI 1 "register_operand" "0")))]
  "TARGET_MUL && TARGET_64BIT"
  "mac\t%0,%2,%3"
+ [(set_attr "type" "imul")
  (set_attr "mode" "DI")])
;;
```

为了让 GCC 识别 mac 指令,我们需要在 GCC 后端添加指令模板,由于 mac 指令可以描述成乘、加两种指令的混合体,所以我们可以直接使用 GCC 中自带的 SPN 来描述指令模板(对于 AES、SHA 这种加密指令,是无法直接使用 GCC 中自带的 SPN 来描述的,只能用 intrinsic,然后通过 intrinsic 与 GCC 后端的对应指令模板做匹配),经过修改后的 riscv.md 文件如左图所示。



# 在spike上增加自定义指令

• 修改encoding.h文件

```
diff --git a/riscv/encoding.h b/riscv/encoding.h
index 9cbb271..42391d2 100644
--- a/riscv/encoding.h
+++ b/riscv/encoding.h
@@ -1844,6 +1844,10 @@
#define MASK VL4R V Oxfff0707f
#define MATCH_VL8R_V 0x1e807007
#define MASK VL8R V Oxfff0707f
+#define MATCH MAC 0xae00700b
+#define MASK MAC 0xfe00707f
#define CSR FFLAGS 0x1
#define CSR_FRM 0x2
#define CSR_FCSR 0x3
@@ -2921.6 +2925.7 @@ DECLARE_INSN(vl1r_v, MATCH_VL1r_v, MASK_VL1r_v)
DECLARE_INSN(v12r_v, MATCH_VL2R_V, MASK_VL2R_V)
DECLARE_INSN(v14r_v, MATCH_VL4R_V, MASK_VL4R_V)
DECLARE_INSN(v]8r_v, MATCH_VL8R_V, MASK_VL8R_V)
+DECLARE_INSN(mac, MATCH_MAC, MASK_MAC)
#endif
#ifdef DECLARE_CSR
DECLARE_CSR(fflags, CSR_FFLAGS)
```

spike 是一款 RISC-V 指令模拟器,我们可 以在 spike 上增加自 定义指令,运行包含 自定 义指令 mac 的应 用程序,从而验证修 改后的 GNU 工具链是 否能够正常工作。



# 在spike上增加自定义指令

- 增加指令描述:
  - 在riscv/insns目录下新建文件mac.h,内容如下:

```
require_extension('M');
WRITE_RD(sext_xlen(RD + RS1 * RS2));
```

• 修改riscv/riscv.mk.in文件:



# 重新构建toolchain和spike

### • 重新构建toolchain

\$ cd riscv-gnu-toolchain

\$ mkdir build && cd build

\$ ../configure --prefix=/opt/riscv --enable-multilib

\$ make && make install

### • 重新构建spike

\$ cd riscv-isa-sim

\$ mkdir build && cd build

\$ ../configure -prefix=/opt/riscv -enable-histogram

\$ make && make install



# 测试toolchain和spike

• 编写测试文件main.c

```
1 #include <stdio.h>
2
3 int32_t op1 = 6;
4 int32_t op2 = 7;
5 int32_t res = 1;
6
7 int main(void)
8 {
9  res += op1 * op2;
10  printf("res = %d\n", res);
11
12  return 0;
13 }
```



# 测试toolchain和spike

### •测试步骤

```
$ riscv64-unknown-elf-gcc -O2 main.c -o main.elf
```

\$ riscv64-unknown-elf-objdump -D main.elf > main.asm

\$ spike pk main.elf

```
000000000000100b0 <main>:
                 7501a603
                                                    a2,1872(gp) # 1f588 <op1>
   100b0:
                                           lw
   100b4:
                 7481a783
                                           LW
                                                    a5,1864(gp) # 1f580 <res>
                                                    a3,1868(qp) # 1f584 <op2>
   100b8:
                 74c1a683
                                           lw
   100bc:
                 6571
                                           lui
                                                    a0,0x1c
   100be:
                 1141
                                                    sp, sp, -16
                                           addi
   100c0:
                 aed6778b
                                                    a5, a2, a3
                                           mac
   100c4:
                 65050513
                                                    a0,a0,1616 # 1c650 < clzdi2+6
                                           addi
   100c8:
                                                    ra, 8(sp)
                                           sd
                 e406
   100ca:
                 0007859b
                                           sext.w
                                                    a1.a5
   100ce:
                 74f1a423
                                                    a5,1864(gp) # 1f580 <res>
                                           SW
   100d2:
                 200000ef
                                           ial
                                                    ra, 102d2 <printf>
   100d6:
                 60a2
                                           ld
                                                    ra, 8(sp)
   100d8:
                 4501
                                           li
                                                    a0,0
   100da:
                 0141
                                           addi
                                                    sp, sp, 16
   100dc:
                 8082
                                           ret
```



### Q & A