# 为什么需要zqh\_riscv

riscv指令集是近几年兴起的开源处理器指令集，在IC领域掀起了一股多年未见的开源风暴，以往谈到处理器往往想到的是桌面通用处理器x86架构，或者在移动通信与嵌入式应用领域大红大紫的ARM，或许还有些人了解一些目前不太流行的power pc、mips架构。然而无论是x86、ARM还是其他的架构目前都是非开源的指令集，若要开发出一款优秀的芯片，这些处理器IP授权是绕不开的，而且芯片设计的巨大投资成本与高风险特性会使很多中小公司“望芯却步”，芯片牌桌上的玩家从来几乎都是资金与技术实力雄厚的的大公司。而一般的中小公司若想开发属于自己的的芯片，是非常困难的，其中有资金的原因、人才的原因，但更重要的是技术门槛太高，芯片设计所需的关键IP与EDA工具都是高度垄断的，尤其是IP授权更是集中在少数公司手中，普通公司即使买来了IP，但要整合成一个完整可用的芯片系统还是需要大量的工作要做，里面的坑是一个接一个的，一不留神就会导致流片失败，回来的芯片成了石头。

芯片领域一直期待着能有一个类似软件界的linux系统的那样的开放平台，然而由于各种因素以及芯片特定的的硬件设计的属性，这样的期待一直没有出现，当然早些年也有一些相对比较成功的开源芯片架构，知名度比较高的就是opencore架构，opencore是一个开源指令集的处理器，有官方的维护网站，有不少开发者在上面开发了很多IP，这些IP包含了处理器core、ddr控制器、总线、usb、以太网控制器等等，然而这些零碎的代码并没有形成一个有机的整体，都是一些孤零零的代码，很多project都没有使用说明文档，有些都无法成功的run起来。opencore官网打开进去，里面的项目已经都多年没有更新了，关注者越来越少，其实opencore没有起来的最大原因，我个人认为还是因为没有一个强有力的开发生态，没有一个强力的个人/组织去不遗余力的去做大开发生态，开发者流失严重。

riscv开源指令集是2015年左右逐渐的进入公众视野的，riscv跟他的前辈mips可谓是师出同门，他们都是精简指令集架构，都出自加州伯克利大学，鉴于前辈mips由于各种原因而日薄西山，riscv就成功的接过了它的衣钵并大胆的对指令集做了永久开源，它从一开始就被视为芯片设计界的linux，IC设计领域可谓是久旱逢甘霖，桌面/服务器领域苦Intel久已，移动/嵌入式领域苦ARM久已。riscv的官方基金会已经迁往瑞士，以防止被单个国家所限制与控制，这对于那些担忧被芯片技术卡脖子的国家或公司是一个很大的鼓励，甚至印度已经将riscv指令集定为国家指令集，我们中国理应开放的拥抱riscv，目前国内已经有不少公司都对rsicv表现出了极大的兴趣，有巨头通信、互联网公司，也有创业公司，有的公司已经开发出了成功芯片产品。

riscv方向目前的领头羊公司无疑是SiFive公司，SiFive公司可以说就是riscv的创始人创立的，对riscv指令集架构有着重要的影响力，他们开发出了多款基于riscv指令集的通用处理器IP，而且成功的流片了，可以跑linux系统，有开发板供给开发者使用。github上有他们的开源项目Rocket--一款基于riscv的开源代码处理器IP，该项目的关注度非常高，实际上很多其他公司开发的基于riscv的芯片都使用了这款开源处理器的代码，或者至少是参考了Rocket的设计思路。SiFive公司主要提供的是riscv处理器IP，其他公司要使用的话需要他们的商业授权，当然单独的处理器IP并不能组成一款芯片，往往还需要购买很多其他公司的各类商业IP，最终有机组合起来才有可能设计出一款可用的riscv架构的芯片。

鉴于最近几年美国对中国的芯片卡脖子问题，国内的很多高科技公司都在储备芯片设计能力，华为的海思无疑是中国目前最顶尖的芯片设计公司，还有像百度、阿里巴巴等互联网公司也都在积极涉足芯片领域。要问目前中国国内芯片设计方向最火的是哪个方向？毫无疑问的是AI(人工智能)方向，大公司也好，小公司也罢，都在往AI芯片里冲，很是热闹。不知道大家有没有感觉到这种场景有点似曾相识？是不是很有中国特色？是的，没错，发现一个潜在的热点，大家都一窝蜂的往里冲，都害怕被落下，华为有昇腾，百度有百度大脑，阿里巴巴有含光，寒武纪也有，oppo/vivio也正在投入AI芯片。说实话我个人认为真的没必要这么多公司去做这些重复的开发，而且不同公司的技术还是独家保密的，这是不是会有很大的资源浪费呢？我个人认为是的。况且芯片热点趋势也不一定永远是AI，未来的IOT市场是一个多元的广阔市场，理应有更多的人/公司去关注。然而这么多公司去设计不同的芯片，重复的造轮子肯定会造成资源浪费，由于有高昂的技术成本，很多中小公司都没有动力去做自己的定制特色芯片。

目前的芯片市场，桌面通用计算领域与移动计算领域已经有x86与ARM两强在统治了，新势力想要挤进去，如果没有颠覆性的创新几乎是不可能的了。AI芯片是Nvidia的天下，各大互联网公司大部分都是依靠GPU在做AI计算，当然也有定制的ASIC AI芯片，但无疑在灵活性与生态方面都不是Nvida的对手，终端AI芯片可以说是手机厂商的天下，高通、苹果的手机芯片无疑是最合适的集成AI终端芯片的地方，其他中小芯片公司开发的五花八门的AI芯片，我个人是不太看好的，因为AI计算的核心还是在数据端、软件端，芯片的功能其实是非常单一的，一个掌握大量训练数据的公司，它如果愿意，设计出一款AI加速芯片其实并不太难，这无疑会使很多专门设计AI芯片但不掌握海量数据的纯芯片公司处于不利的地位。

我个人认为，中国目前非常需要一个开放的芯片设计平台来整合各个零散的芯片开发技术，一个有统一标准且开源的芯片开发平台会大大降低芯片设计的门槛，非常有利于众多的中小公司来实现他们的定制芯片，技术都是相通的，大家都贡献出来并没有什么不好，蛋糕做大了，中国整体的芯片实力也就上去了。

摩尔定律已经逐渐的接近物理极限，在硅片工艺上能压榨出来的性能也越来越少了，未来的芯片形态一定往定制化、专用化方向发展，而且开发芯片的门槛会越来越低，就像现在的PCB电路设计一样简单，流片的成本也会随着工艺的进步逐渐降低，未来的芯片市场一定是一个百花齐放的形态。如果有一个开放的类似linux系统那样的平台，无疑会大大推进芯片设计往平民化方向发展。至少在目前全球科技民族主义之风盛行的时代，中国的内部是迫切的需要这样一个开放平台的。

而这也是为什么需要zqh\_riscv的原因，zqh\_riscv的最终目标就是要建一个中国的开放芯片设计平台，整合目前的各种主流IP，基于开源riscv指令集来构建一个灵活、通用、易用的芯片开发平台。如果这个目标能实现，在未来的IOT时代一定会有更多的有活力的中小公司敢于去定制自己的芯片，开发出自己的特色产品，甚至于也能让个人也承担得起芯片设计的成本与风险。

# zqh\_riscv整体介绍

zqh\_riscv是一套开源SOC开发平台，核心部分包含处理器core、cache、片内互联总线、中断控制器、memory控制器、片内总线slave接口、片内总线master接口、片内总线device、片外总线device、时钟复位控制器、debug控制器。还包含了SOC功能验证/仿真相关的脚本程序和测试用例。除了可以运行电路仿真，平台还提供了ASIC综合脚本，可以对生成的电路做逻辑综合。支持在FPGA上的原型仿真验证。

处理器core选择开源指令集的riscv架构，zqh\_riscv SOC的目标应用平台是各种IOT设备/嵌入式设备，处理器core不采用通用计算处理器的微架构，因此core不会集成MMU/TLB、多核cache一致性等典型AP处理器的功能。core的微架构常常选择类似于Rocket项目的结构，力求在能满足计算要求的前提下尽可能占用最低的面积与功耗。

片内互联总线选择的是tilelink总线，tilelink是开源的总线标准，它跟riscv指令集一样出自加州伯克利大学，跟riscv core的搭配最合适不过了。tilelink没有ARM的AMBA总线的名气大，但它简洁高效的结构比AXI/AHB等ARM总线更适合IOT芯片。但鉴于目前各种商业IP提供的大部分都是AMBA总线的接口，zqh\_riscv也提供了tilelink接口与AMBA总线接口的转换。

作为一个集成了处理器core的SOC系统，中断控制器自必不可少，zqh\_riscv平台提供了通用的本地中断控制器与平台中断控制器。本地中断控制器与特定的处理器core紧耦合。平台中断控制器是所有外设的中断控制中枢，负责把特定的外设中断请求送给特定的处理器core的外部中断引脚上。

zqh\_riscv平台集成了目前各种主流的外设IP，例如UART、SPI、I2C、GPIO、PWM、JTAG、USB、Ethernet MAC、DDR、DMA引擎、时钟与复位控制器(CRG)、调试debug控制器等。除了模拟电路相关的功能，大部分数字电路相关的IP都做了整合集成，IP的接口统一为tilelink，IP既可以作为子模块集成进zqh\_riscv系统，也可以单独使用并集成到任意地方。

zqh\_riscv平台还提供了一套仿真脚本，可以运行仿真测试用例。综合脚本实现ASIC电路综合。可以在FPGA上跑原型仿真，zqh\_riscv内会自动替换部分FPGA相关的cell。

zqh\_riscv平台的实现语言以python为主，硬件描述代码使用的是PHGL，PHGL可以构建高度参数化的模块电路。



图表 2‑1 zqh\_riscv系统硬件结构图

zqh\_riscv的完整硬件系统如上图所述，zqh\_riscv处理器外挂tilelink master与slave接口，memory bus与IO bus分别由独立的tileink master控制。fbus slave接口用来提供外部访问的接口，例如带master接口的外设访问ITIM/DTIM memory。mem bus上挂接onchip sram、DDR3控制器、SPI XIP Flash控制器。mmio bus上挂接IO属性的配置模块与外设，支持各种主流外设: I2C、SPI、UART、PWM、USB等。目前还不支持原生ADC与DAC，模拟电路相关部分暂时无原生IP提供，但是后续规划中会随着模拟电路部分的完善而加入，print\_monitor是一个仿真打印device，用来打印软件输出的debug信息。时钟产生模块由于通常有PLL等模拟电路，目前没有原生IP，但有后续规划加入。支持jtag debug接口，可以调试软件。

zqh\_riscv平台提供的不仅仅是芯片硬件平台，还包含与之配套的软硬件调试脚本、测试用例、test benth、逻辑综合脚本等必不可少的部分。



图表 2‑2 zqh\_riscv完整组件框图

全芯片的测试仿真需要test benth，test benth中提供各种标准接口的仿真模型，例如DDR、UART、I2C、SPI、eth GMII、USB host/device等。提供一整套测试用例，配合软件代码可以测试芯片系统的各个组成模块。

# memory地址映射

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Start | End | Attribute | Description |  |
| 0x0000\_0000 | 0x0000\_0fff | RWXC | Debug |  |
| 0x0001\_0000 | 0x0001\_ffff | RXC | Bootrom |  |
| 0x0100\_0000 | 0x010f\_ffff | RW | ITIM IO access | ITIM IO address mapping |
| 0x0110\_0000 | 0x011f\_ffff | RW | L1I cache controler |  |
| 0x0120\_0000 | 0x012f\_ffff | RW | DTIM IO access | DTIM IO address mapping |
| 0x0130\_0000 | 0x013f\_ffff | RW | L1D cache controler |  |
| 0x0200\_0000 | 0x0200\_ffff | RW | CLINT |  |
| 0x3000\_0000 | 0x300f\_ffff | RW | DMA |  |
| 0x0c00\_0000 | 0x0fff\_ffff | RW | PLIC |  |
| 0x1000\_0000 | 0x1000\_0fff | RW | CRG\_CTRL |  |
| 0x1001\_2000 | 0x1001\_2fff | RW | GPIO0 | manage UART0 PWM0 I2C0 |
| 0x1001\_3000 | 0x1001\_3fff | RW | UART0 |  |
| 0x1001\_4000 | 0x1001\_4fff | RW | SPI0 |  |
| 0x1001\_5000 | 0x1001\_5fff | RW | PWM0 |  |
| 0x1001\_6000 | 0x1001\_6fff | RW | I2C0 |  |
| 0x1001\_f000 | 0x1001\_ffff | RW | GPIO1 | manage ETH\_MAC\_PHY USB\_HOST\_PHY USB\_DEVICE\_PHY |
| 0x1002\_0000 | 0x1002\_ffff | RW | ETH\_MAC |  |
| 0x1003\_0000 | 0x1003\_ffff | RW | DDR\_MC |  |
| 0x1004\_0000 | 0x1004\_ffff | RW | USB\_HOST |  |
| 0x1005\_0000 | 0x1005\_ffff | RW | USB\_DEVICE |  |
| 0x1f00\_0000 | 0x1fff\_ffff | RW | Print monitor | Simulation only |
| 0x2000\_0000 | 0x2fff\_ffff | RXC | SPI flash XIP |  |
| 0x4000\_0000 | 0x4fff\_ffff | RWXC | On chip SRAM |  |
| 0x5000\_0000 | 0x5fff\_ffff |  | Reserved |  |
| 0x6000\_0000 | 0x601f\_ffff | RWX | ITIM memory access | ITIM core access mapping |
| 0x6020\_0000 | 0x603f\_ffff | RWX | DTIM memory access | DTIM core access mapping |
| 0x8000\_0000 | 0xffff\_ffff | RWXC | DDR memory |  |

# zqh\_riscv各子系统介绍

## 时钟与复位(CRG)



图表 4‑1 时钟与复位(CRG)模块

clock\_ref是系统参考时钟，来自芯片外部输入，经过各个子系统的pll之后生成各种所需的时钟。

reset\_por是上电复位信号，来自芯片外部输入，crg\_ctrl的配置寄存器控制各个子系统的复位是否有效，复位信号是高电平有效。

crg\_ctrl是CRG控制信号，配置值是软件可修改的，用来在boot阶段控制各个子系统的时钟与复位信号的时序关系。

## zqh\_riscv core/wrapper

### zqh\_riscv r1 core



图表 4‑2 zqh\_riscv r1 core流水线

zqh\_riscv r1 core的流水线结构参考了成熟的开源riscv项目Rocket，是6级主流水线结构。取指单元IFU内部是2级流水线，IFU取出的指令在预解码(inst asm)之前先入队指令缓冲buffer(inst queue)，inst queue是可以bypass的，如果queue为空且后级没有反压则直接bypass IFU的指令，否则会暂存进内部寄存器等待。inst asm模块是指令预解码模块，主要对compress压缩指令做展开，如果32位指令跨了非4字节对齐边界，inst asm模块会等待指令拼接完整之后再输出，有可能会延迟1个周期输出。实际上整个流水线是变长的，如果inst queue bypass且inst asm模块没有指令拼接，流水线是最少的6级的，否则可能是7级或8级的。

IFU取指的来源可以是L1I$(一级指令cache)，也可以是ITIM(指令TIM)，L1I$与ITIM都是2级流水线的，L1I$ hit的延时跟ITIM是一样的。如果是uncacheable的指令，指令数据将直接从总线返回。

LSU的访问目标可以是L1D$(一级数据cache)，也可以是DTIM(数据TIM)，L1D$与DTIM都是2级流水线的，L1D$ hit的延时跟DTIM是一样的。如果是uncacheable的访问，数据将直接从总线返回。

ITIM与DTIM不仅可以被core访问，也可以通过IO地址空间被外部设备访问，例如DMA引擎可以读写ITIM/DTIM的数据。

分支预测是可选的，内部细分为3个单元：BTB(branch target buffer) BHT(branch history table) RAS(return address stack)。如果集成分支预测功能的话，BTB是必选的，BHT与RAS是可选的。

浮点处理单元FPU(float point unit)是可选的功能。

### zqh\_riscv e1 core



图表 4‑3 zqh\_riscv e1 core流水线

zqh\_riscv e1 core是针对超低功耗设计的，进一步精简了流水线设计，原r1 core的ID EX MEM流水级合并为一个EX级，即e1是3级流水结构的。对于只需要ALU运算的指令是3级流水。需要memory访问且访问的是DTM的指令，EX级需要延迟一个/两个周期，DTIM的访问延时可以是1或2个周期。

IFU、inst queue、inst asm、inst decode等前端模块跟r1 core的保持一致，LSU取消了l1 cache，只保留了DTIM，DTIM地址空间之外的load/store都需要从外部总线访问。

### zqh\_riscv\_core wrapper



图表 4‑4 zqh\_riscv\_core wrapper

zqh\_riscv\_core wrapper是对core、IFU、LSU、FPU、中断、in/out bus做的一层封装，使得整个riscv core对外呈现为一个相对独立的IP。对外的互连总线是tilelink总线，有一个master接口与一个slave接口，master接口向外部总线发送memory/device请求，slave接口用来给外设访问内部的ITIM/DTIM。

## zqh\_riscv tile



图表 4‑5 zqh\_riscv tile结构

zqh\_riscv tile整合了zqh\_riscv core wrapper、CLINT、PLIC、debug、DMA、system bus等模块，这些功能模块都是与core紧密耦合的，所以集成到一个子系统内部，core的数量可以是1-4个。

对外呈现出如下接口：

1. mem bus master，该master接口用来访问memory属性的load/store请求。
2. mmio bus master，该master接口用来访问device属性的load/store请求。
3. front bus slave，该slave接口用来访问tile内部的资源。
4. device中断线输入，所有的片内外设与片外中断都连接到内部的PLIC。
5. debug DMI接口，与JTAG DTM模块通过DMI接口连接，用来debug芯片。

## zqh\_riscv片上互连总线

zqh\_riscv采用的是tilelink互连总线，tilelink是一款免费开源的片上总线协议，可以说是为riscv量身定做的，当然可以用到其他非riscv架构的芯片上。

详细的tilelink规格文档可以参考: <https://www.sifive.com/documentation>

zqh\_riscv系统面向的并不是高性能计算的场景，所有并不包含复杂的cache一致性协议，所以没有用到TL-C，使用到了TL-UH与TL-UL。



图表 4‑6 片上互连总线树状图

## Core Local Interruptor (CLINT)

CLINT负责产生软件中断请求与timer中断请求。软件中断请求经常使用在多核通信的场景，例如core0可以写core0-3的软件中断寄存器来触发相应的core的中断。timer中断负责产生每个core的定时心跳中断。

## Platform-Level Interrupt Controller(PLIC)

芯片片内各个device的中断请求在PLIC汇聚，PLIC内部寄存器控制各个中断源的mask与优先级，汇聚之后的中断请求送入riscv core的meip中断引脚。

|  |  |  |
| --- | --- | --- |
| IRQ ID | Peripheral | Description |
| 0 |  |  |
| 1 |  |  |
| 2 |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

图表 4‑7 中断ID与设备映射关系

## DMA engine

支持的特性：

1. 4个独立的通道
2. 读写粒度可配2的n次方格式: 1 2 4 8……256
3. 支持严格order保序或者乱序

## Universal Asynchronous Receiver/Transmitter (UART)

支持的特性:

1. 发送parity校验码生成有/无，奇校验/偶校验
2. 接收parity校验码检查有/无，奇校验/偶校验
3. 发送支持1/2 stop bit
4. 8深度发送数据fifo，fifo支持flush操作
5. 8深度接收数据fifo，fifo支持flush操作
6. 16倍波特率接收过采样
7. 状态上报：错误检测(接收start/stop错误、接收数据校验错误)，发送fifo数据个数，接收fifo数据个数

## Serial Peripheral Interface (SPI)

支持的特性:

1. sck时钟的极性与相位可配
2. cs与sck的相对延时可配
3. 数据big endian/small endian可配
4. 传输模式支持normal mode(single mode)、dual mode、quad mode
5. 8深度发送数据fifo，fifo支持flush操作
6. 8深度接收数据fifo，fifo支持flush操作
7. 支持SPI flash读写指令
8. 支持XIP

## Inter－Integrated Circuit(I2C)

支持的特性:

1. 工作在master模式
2. 支持multi master下工作
3. 支持slave stretch SCL
4. 支持SCL时钟占空比调节配置
5. 支持data hold/sample的时序调节配置

## Pulse-Width Modulation (PWM)

支持的特性：

1. always模式/one shot模式
2. 4路PWM输出
3. 分频范围为2\*\*0 – 2\*\*15
4. 改变比较值时支持Deglitch去毛刺
5. 支持生成左对齐/右对齐PWM输出
6. 支持中心对齐PWM输出
7. 支持4路PWM的仲裁捆绑
8. 支持counter模式/timer模式

## General Purpose Input/Output Controller (GPIO)

支持的特性:

1. 两个GPIO group(GPIO0 GPIO1)
2. 每个GPIO group管理32个IO PAD
3. GPI0管理UART0 PWM0 I2C0的IO PAD
4. GPIO1管理ETH\_MAC\_PHY USB\_HOST\_PHY USB\_DEVICE\_PHY的IO PAD
5. 每个GPIO的IO pad可以配置为普通软件IO或者硬件功能IO

## Universal Serial Bus(USB) controler & PHY

controler支持的特性：

1. USB1.1 FULL speed/LOWspeed
2. UTMI+接口
3. host模式device模式可配
4. host模式64 byte tx fifo与64 byte rx fifo
5. host模式SOF帧的发送
6. device模式下支持4个endpoint。每个endpoint有各自独立的tx fifo与rx fifo，fifo的深度都是64 byte

PHY支持的特性:

1. 支持直接寄存器控制DP/DM信号线
2. 支持EOP时序参数的配置
3. line state的实时上报
4. connection状态的实时上报

## Ethernet MAC GMII controler

支持的特性：

1. PHY接口支持MII/GMII
2. 数据速率支持10M/100M/1000M
3. 半双工模式/全双工模式
4. 发送数据CRC的硬件计算并自动插入
5. 发送短包PAD数据字节的自动插入
6. 接收CRC校验
7. 接收数据包超短/超长检测
8. 支持source MAC地址的自动插入
9. 支持发送/接收控制帧
10. 支持接收并硬件处理流控帧
11. 支持接收广播数据帧的地址hash过滤
12. 4KB的发送数据buffer
13. 4KB的接收数据buffer
14. 8深度的发送BD(Buffer Descripter) fifo
15. 8深度的接收BD fifo
16. 8深度的tx CPL(complete) fifo
17. 8深度的rx CPL(complete) fifo
18. SMI接口支持

## DDR3 memory controler & PHY

控制器支持的特性：

1. DDR3模式
2. 与DDR PHY的接口为DFI接口
3. 支持4个X8 PHY或者2个X16 PHY的模式
4. 支持DDR不同bank读写命令的乱序执行
5. 最多支持4个outstanding读写命令
6. 写操作支持full data write/partitial write
7. 支持读写的burst长度为1-2K byte
8. 支持自动刷新频率的配置
9. ODT支持
10. 支持DDR PHY MR寄存器的配置
11. 错误检测与状态上报
12. 所有DDR相关时序参数可配
13. 不支持tranning

DDR PHY支持的特性

1. DDR3模式
2. DFI接口
3. 支持X8/X16模式
4. ODT支持
5. DLL延迟参数可配

## JTAG Debug Interface

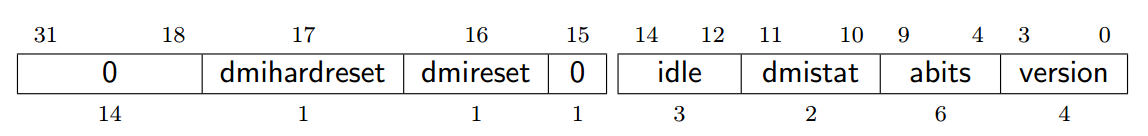
参考《riscv-debug-release.pdf》

|  |  |  |
| --- | --- | --- |
| Address | Name | Description |
| 0x00 | BYPASS | JTAG recommends this encoding |
| 0x01 | IDCODE | JTAG recommends this encoding |
| 0x10 | DTM Control and Status (dtmcs) | For Debugging |
| 0x11 | Debug Module Interface Access (dmi) | For Debugging |
| 0x12 | Reserved (BYPASS) |  |
| 0x13 | Reserved (BYPASS) |  |
| 0x14 | Reserved (BYPASS) |  |
| 0x15 | Reserved (BYPASS) |  |
| 0x16 | Reserved (BYPASS) |  |
| 0x17 | Reserved (BYPASS) |  |
| 0x1f | BYPASS | JTAG requires this encoding |

图表 4‑8 JTAG DTM TAP Registers

### DTM Control and Status (dtmcs, at 0x10)

The size of this register will remain constant in future versions so that a debugger can always determine the version of the DTM.



图表 4‑9 dtmcs寄存器

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Description | Access | Reset |
| dmihardreset | Writing 1 to this bit does a hard reset of the DTM,causing the DTM to forget about any outstanding DMI transactions. In general this should only be used when the Debugger has reason to expect that the outstanding DMI transaction will never complete (e.g. a reset condition caused an inflight DMI transaction to be cancelled). | W1 | - |
| dmireset | Writing 1 to this bit clears the sticky error state and allows the DTM to retry or complete the previous transaction. | W1 | - |
| idle | This is a hint to the debugger of the minimum number of cycles a debugger should spend in RunTest/Idle after every DMI scan to avoid a ‘busy’ return code (dmistat of 3). A debugger must still check dmistat when necessary.  0: It is not necessary to enter Run-Test/Idle at all.  1: Enter Run-Test/Idle and leave it immediately.  2: Enter Run-Test/Idle and stay there for 1 cycle before leaving. And so on. | R | Preset |
| dmistat | 0: No error.  1: Reserved. Interpret the same as 2.  2: An operation failed (resulted in op of 2).  3: An operation was attempted while a DMI access was still in progress (resulted in op of 3). | R | 0 |
| abits | The size of address in dmi. | R | Preset |
| version | 0: Version described in spec version 0.11.  1: Version described in spec version  0.13.  15: Version not described in any available version of this spec. | R | 1 |

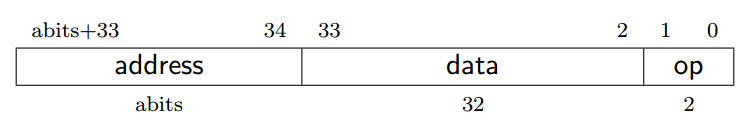
图表 4‑10 dtmcs寄存器域段描述

### Debug Module Interface Access (dmi, at 0x11)

This register allows access to the Debug Module Interface (DMI).

In Update-DR, the DTM starts the operation specified in op unless the current status reported in op is sticky.

In Capture-DR, the DTM updates data with the result from that operation, updating op if the current op isn’t sticky.



图表 4‑11 dmi寄存器

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Description | Access | Reset |
| address | Address used for DMI access. In Update-DR this value is used to access the DM over the DMI. | R/W | 0 |
| data | The data to send to the DM over the DMI during Update-DR, and the data returned from the DM as a result of the previous operation. | R/W | 0 |
| op | When the debugger writes this field, it has the following meaning: 0: Ignore data and address. (nop)  Don’t send anything over the DMI during Update-DR. This operation should never result in a busy or error response. The address and data reported in the following Capture-DR are unde fined.  1: Read from address. (read)  2: Write data to address. (write)  3: Reserved. When the debugger reads this field, it means the following:  0: The previous operation completed successfully.  1: Reserved.  2: A previous operation failed. The data scanned into dmi in this access will be ignored. This status is sticky and can be cleared by writing dmireset in dtmcs. This indicates that the DM itself responded with an error. There are no specified cases in which the DM would respond with an error, and DMI is not required to support returning errors.  3: An operation was attempted while a DMI request is still in progress. The data scanned into dmi in this access will be ignored. This status is sticky and can be cleared by writing dmireset in dtmcs. If a debugger sees this status, it needs to give the target more TCK edges between Update DR and Capture-DR. The simplest way to do that is to add extra transitions in Run-Test/Idle. | R/W | 0 |

图表 4‑12 dmi寄存器域段描述

# zqh\_riscv test harness

zqh\_riscv test harness是testbench的顶层，包含的功能如下:

1. 例化SOC全芯片(DUT)
2. 例化芯片接口功能的仿真模型并跟DUT连接



图表 5‑1 zqh\_riscv\_test\_harness结构

# zqh\_riscv testbench

验证环境是用纯SV语言编写，之所以没有使用类似UVM/VMM的验证方法学，主要考虑到如下几点原因:

1. 增强环境的可移植性，只要是支持SV语言仿真的都可以无差别的运行。而UVM/VMM环境不是每款仿真器都支持的，有些底层的特性甚至是跟仿真软件深度耦合的，不同的仿真器的运行结果可能有差异。
2. 尽量用简洁的语法，UVM/VMM是比较臃肿的，内部的各种运行机制颇为复杂，不可避免的会导致环境臃肿。
3. 最终的交付件是RTL代码，验证环境不是最终要交付的产品。再高级的方法学搭建的验证环境，即使在最初看起来很优雅，但是到验证的后期都避免不了变得”千疮百孔”。存SV搭建的环境无论是修改维护还是debug都会高效很多。
4. 方便沟通交流，RTL设计人员往往对验证方法学的东西不甚了解，过多的验证方法学无疑会增加设计人员跟验证人员沟通的障碍。



图表 6‑1 验证环境目录结构

common目录下是公共资源与工作组件。vips目录下是各种仿真组件，例如以太网PHY模型、DDR模型、JTAG BFM模型、SPI flash芯片等。zqh\_vbase\_lib下存放验证环境中的一些基本class，例如log打印等。

scripts目录下是各种脚本程序，主要是python语言编写的。

zqh\_riscv目录是验证的主体目录，下面有env, sim, tests, th四个子目录。env目录下是环境代码，例如宏定义、基本数据类、DUT初始化等相关的公共代码。sim目录是跑仿真时的目录，通过makefile运行测试用例，生成log与波形文件。tests目录是测试用例目录，下面有2个子目录，zqh\_riscv\_sw目录里存放软件代码，以c语言为主，zqh\_riscv\_tests下存放的是SV测试用例。th目录下是验证环境的顶层，例化验证环境组件与DUT module，实现环境与DUT的连通。

# zqh\_riscv test cases

## ISA测试用例

这些都是riscv指令集相关的测试case，来自于riscv工具链里自带的标准测试程序。



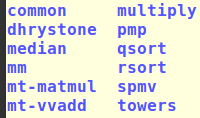


图表 7‑1 ISA测试用例

## benchmarks标准测试用例

来自于riscv工具链里自带的标准测试程序。



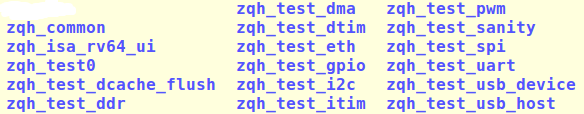


图表 7‑2 benchmarks测试用例

## zqh\_riscv模块功能测试用例

这些测试用例对应于芯片的各个内部子系统模块。





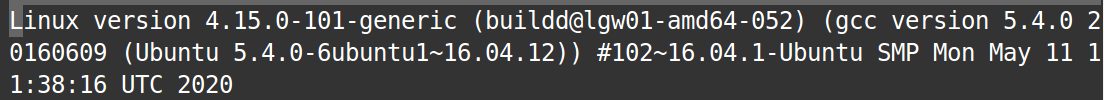
图表 7‑3 zqh\_riscv模块功能测试用例

# zqh\_riscv simulation

## 仿真环境准备

### 操作系统

ubuntu 16.04(64位)

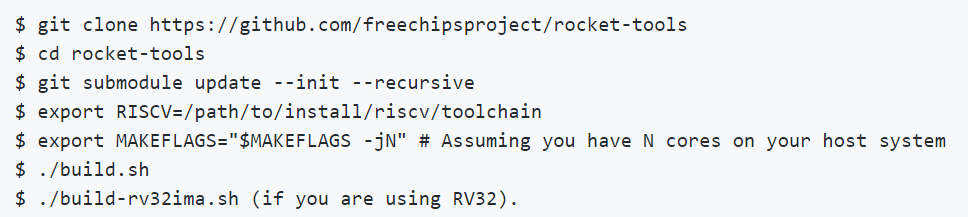


系统需要支持python3.0以上版本

系统gcc version 5.4.0

### riscv交叉编译工具链安装

github下载最新代码并安装:



### EDA软件

RTL仿真: vcs 2016.06(64位)/iverilog

波形查看: verdi 2015.09-SP1-1/gtkwave

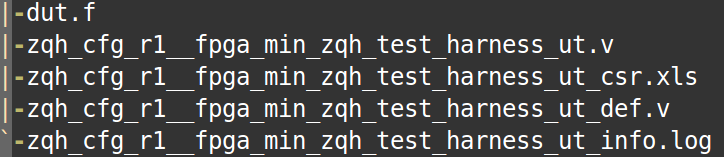
## PHGL to verilog RTL

PHGL编写的代码在仿真之前需要转换成verilog代码。

在sim目录下运行如下命令：

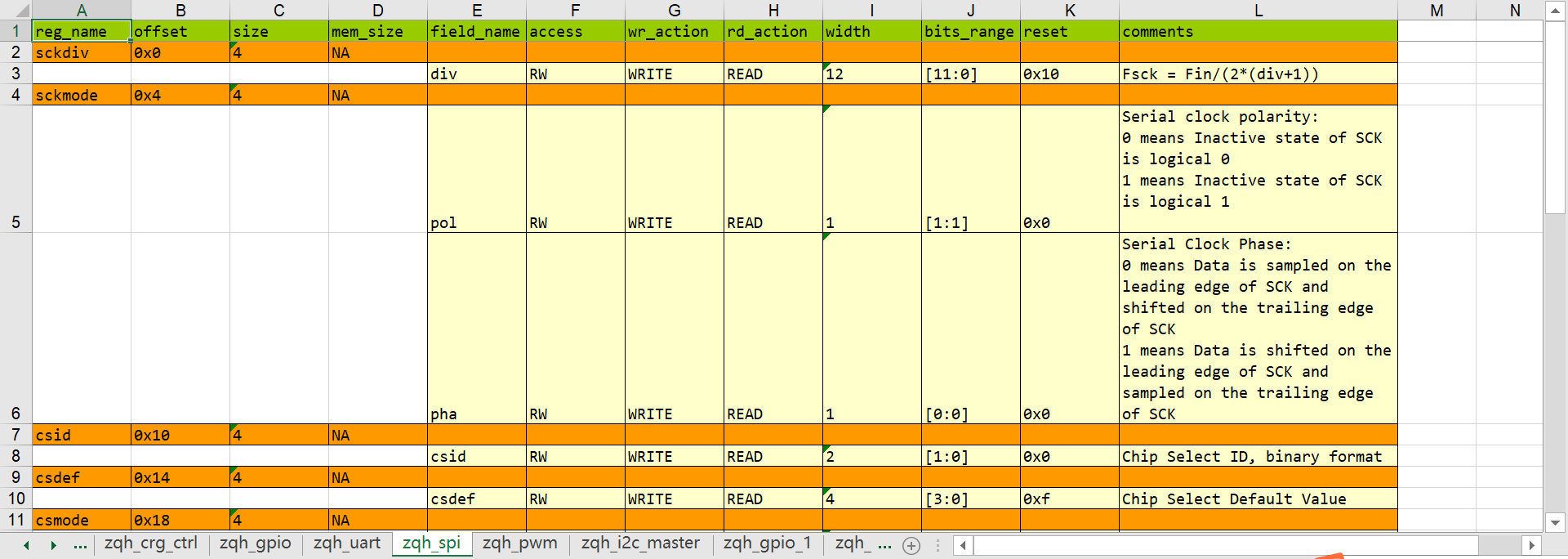


等命令执行完毕，可以在当前的py2rtl目录下看到生成的几个文件:



dut.f是RTL文件列表，内部是两个.v文件的路径。xxx.v与xxx\_def.v是verilog文件，xxx.v是module代码，xxx\_def.v是宏定义。

xxx\_csr.xls是csr module的配置寄存器描述表格，可以用Excel打开，每个module的csr存放在不同的sheet里，内容如下:



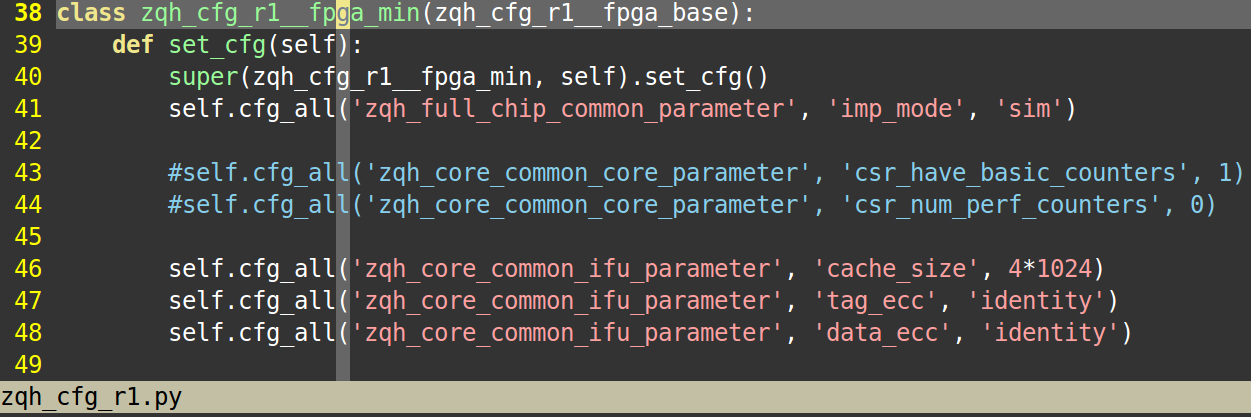
info.log是PHGL在运行时的log信息，用于debug PHGL语法错误或者提取模块信息。

上述make gen\_rtl命令用的默认配置参数，如果要用其他的配置参数，需要加上gen\_cfg参数：



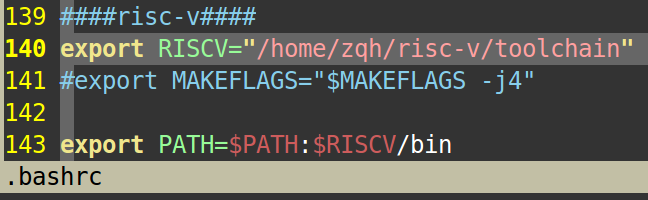
gen\_cfg参数用于指定配置参数，配置参数是一个python class。





## riscv软件编译

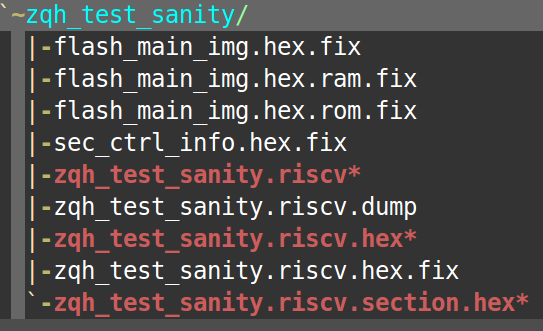
编译之前需要先安装riscv gcc交叉编译工具链。并设置$RISCV环境变量到系统PATH路径:



例如，如果要编译zqh\_test\_sanity用例:



如果没有编译错误，会在log目录下生成如下文件:



xxx.riscv.hex是仿真环境需要加载的程序代码。flash\_main\_xxx3个文件是FPGA原型仿真时需要放入SPI flash中的主程序代码。sec\_ctrl\_info.hex是主程序代码的段描述信息，FPGA仿真时需要跟随主程序代码一同写入SPI flash。

## 运行仿真

运行zqh\_test\_sanity用例:



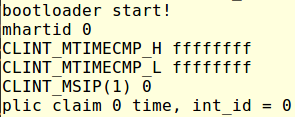
该命令会编译SV验证环境，在sim目录下生成simv可执行文件。编译完成之后自动运行生成的simv可执行文件。



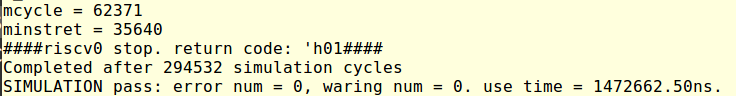
默认的仿真软件是vcs，如果需要改为iverilog，需要命令行指定”simulator=iverilog”:



仿真开始之后，可以在屏幕上看到如下的软件打印信息:



仿真结束后会看到如下打印信息:



mcycle是软件运行的时钟周期数，minstret是执行的指令个数。return code为1表示用例执行成功，其他code值表示用例执行发现错误，需要debug定位。

编译log:



仿真log:



## make all命令

make all命令用来顺序执行上述的三个命令: PHGL to rtl, 软件编译，环境编译/仿真。不用再单独的执行上面3个命令。



## ISA用例仿真

ISA用例仿真时make命令需要传入如下3个参数来指定运行哪个用例:





## 观察波形

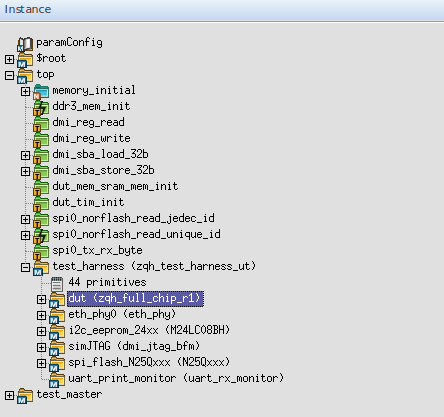
仿真默认生成的是fsdb格式的波形文件，需要用verdi打开：

波形文件:

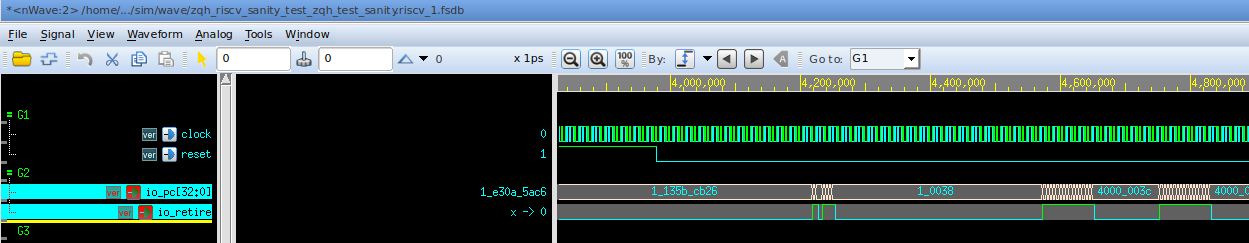


打开verdi:





打开fsdb文件:



verdi rc文件:

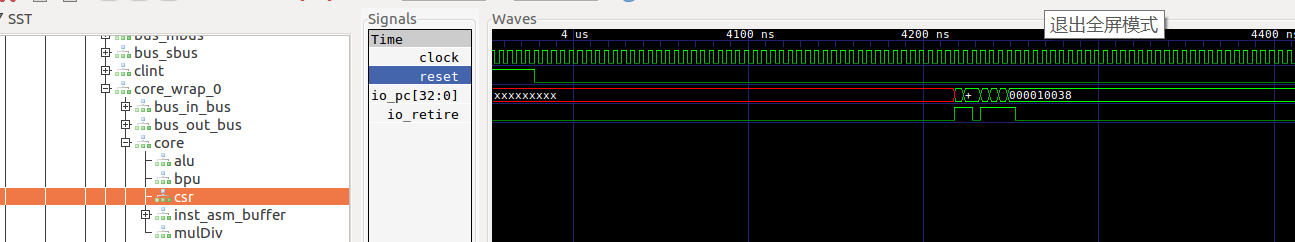


如要要生成vcd波形文件，运行仿真时加如下参数:



用gtkwave打开vcd波形:





# zqh\_riscv synthesize

# zqh\_riscv FPGA verification

## 软硬件环境准备

### FPGA原型开发板

ALINX XILINX A7 FPGA 黑金开发板(Artix-7 100T AX7102)

### 综合工具

Vivado 2019.1

### Arduino UNO R3开发板

用途1: 用来烧写存放riscv代码的SPI flash。

用途2: zqh\_riscv JTAG debug。

### SPI flash模块

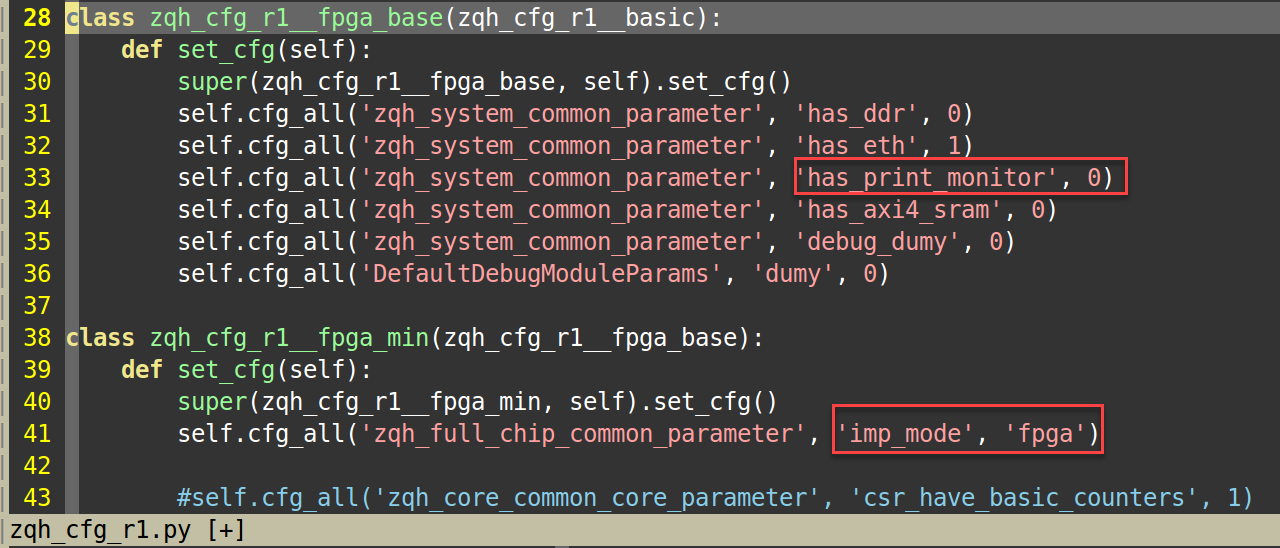
模块颗粒为W25Q128，用来存放riscv的软件代码。

### windows PC机

需安装python3运行环境。作为Arduino UNO R3的上位host主机，通过usb to uart与Arduino板通信。

## RTL生成

### 修改默认的PHGL配置参数



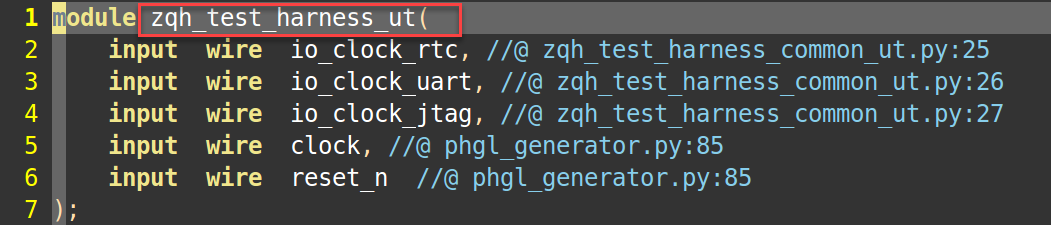
有两个参数需要修改:

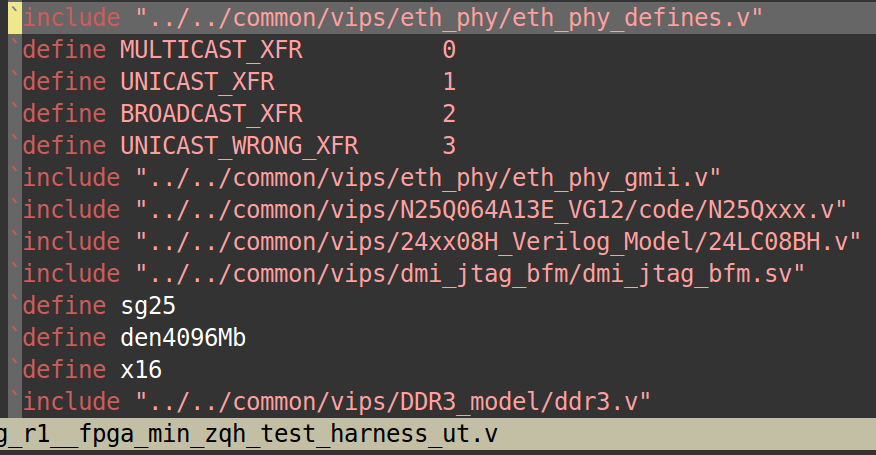
has\_print\_monitor改为0。表示不例化仿真用的print\_monitor模块，该模块只能用在仿真环境。

imp\_mode改为fpga。表示是fpga的实现方式，PLL、分频器等时钟模块将会例化为fpga的资源IP。

### 执行make gen\_rtl命令

RTL相关代码会生成到py2rtl目录下。因为生成的RTL里包含了zqh\_test\_harness\_ut顶层等非可综合的模块，vivado综合之前需要删掉这些代码。





## vivado综合

### 解压并打开project模板



project里已经完成了必要的配置，约束文件已经写好。

### 覆盖project的RTL文件

用上一步生成并修改过的.v文件覆盖projcet下sources\_1目录下的.v文件

### implementation & generate bitstream

生成的bitstream写入FPGA开发板的配置flash

## SPI flash模块刷入riscv启动代码

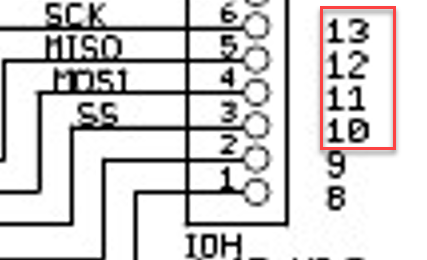
### arduino UNO开发板刷入spi\_flash\_prog固件程序

arduino UNO通过AVR单片机的硬件SPI接口对SPI flash模块编程。



arduino UNO开发板的SPI接口连接SPI flash模块，注意SPI flash模块的供电电压是3.3v的，arduino的5v IO电压要转换成3.3v的才能跟SPI flash模块连接。

arduino UNO开发板SPI接口:



### riscv bootloader固件hex文件生成





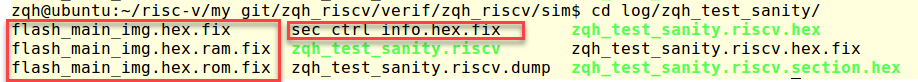
把生成的bootloader.hex.fix文件copy到PC机的spi\_flash\_prog\_host.py脚本所在的目录下。



### riscv主程序固件hex文件生成



把生成的镜像文件copy到PC机的spi\_flash\_prog\_host.py脚本所在的目录下。



默认情况下，上述命令生成的hex镜像文件的程序跳转入口是在zqh\_riscv的片内SRAM。即bootloader会将程序镜像加载到片内SRAM里执行。

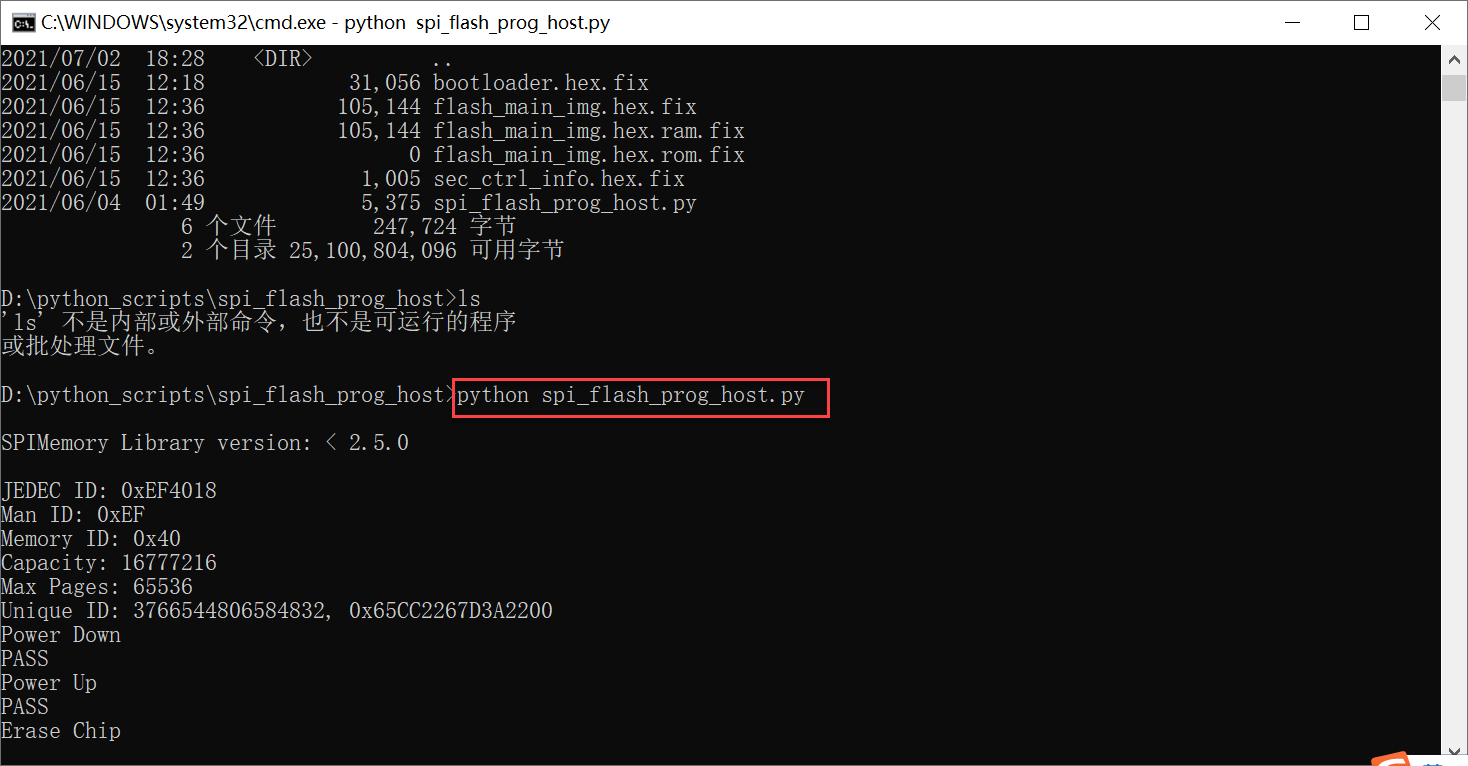
### 命令行启动spi\_flash\_prog\_host.py脚本程序

PC机通过usb转uart跟arduino UNO开发板连接。

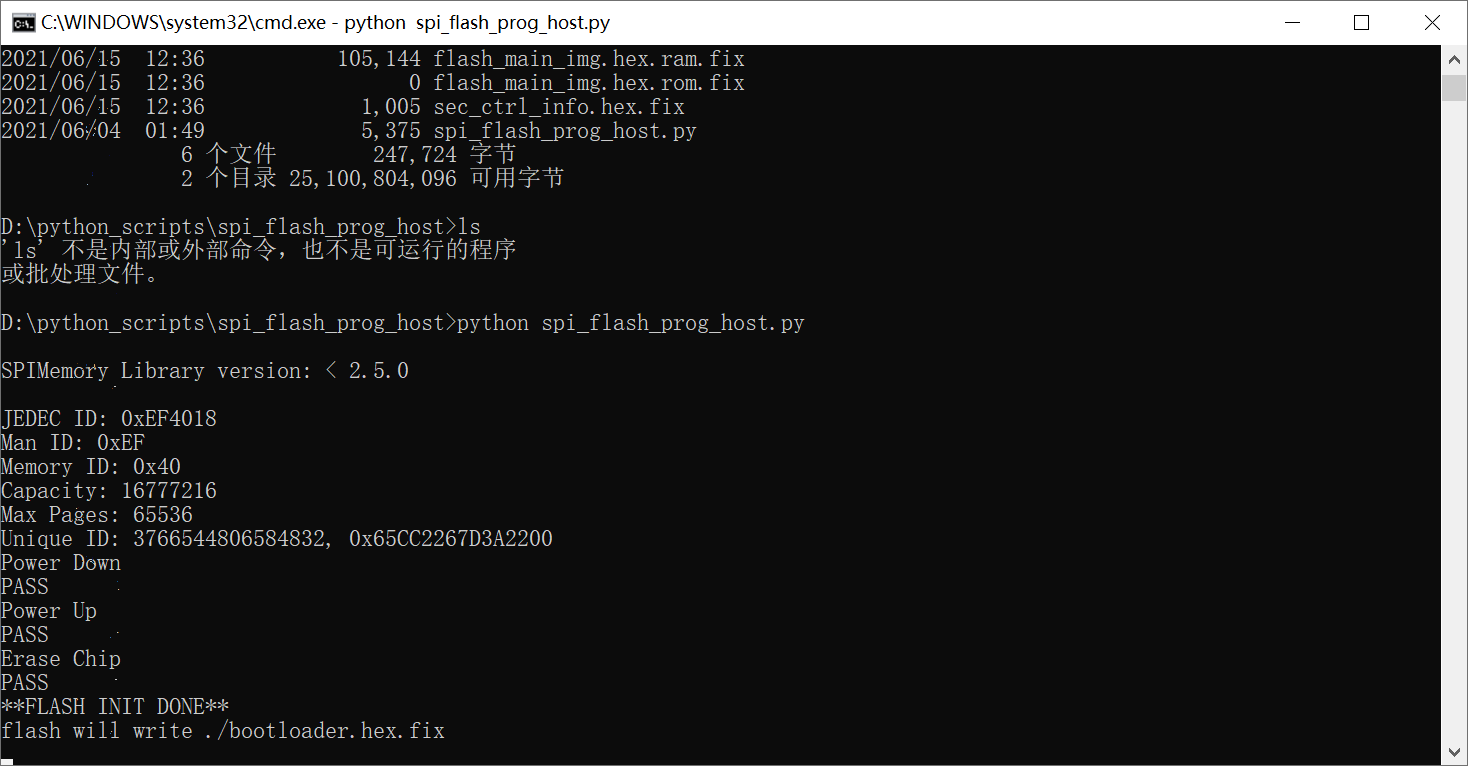


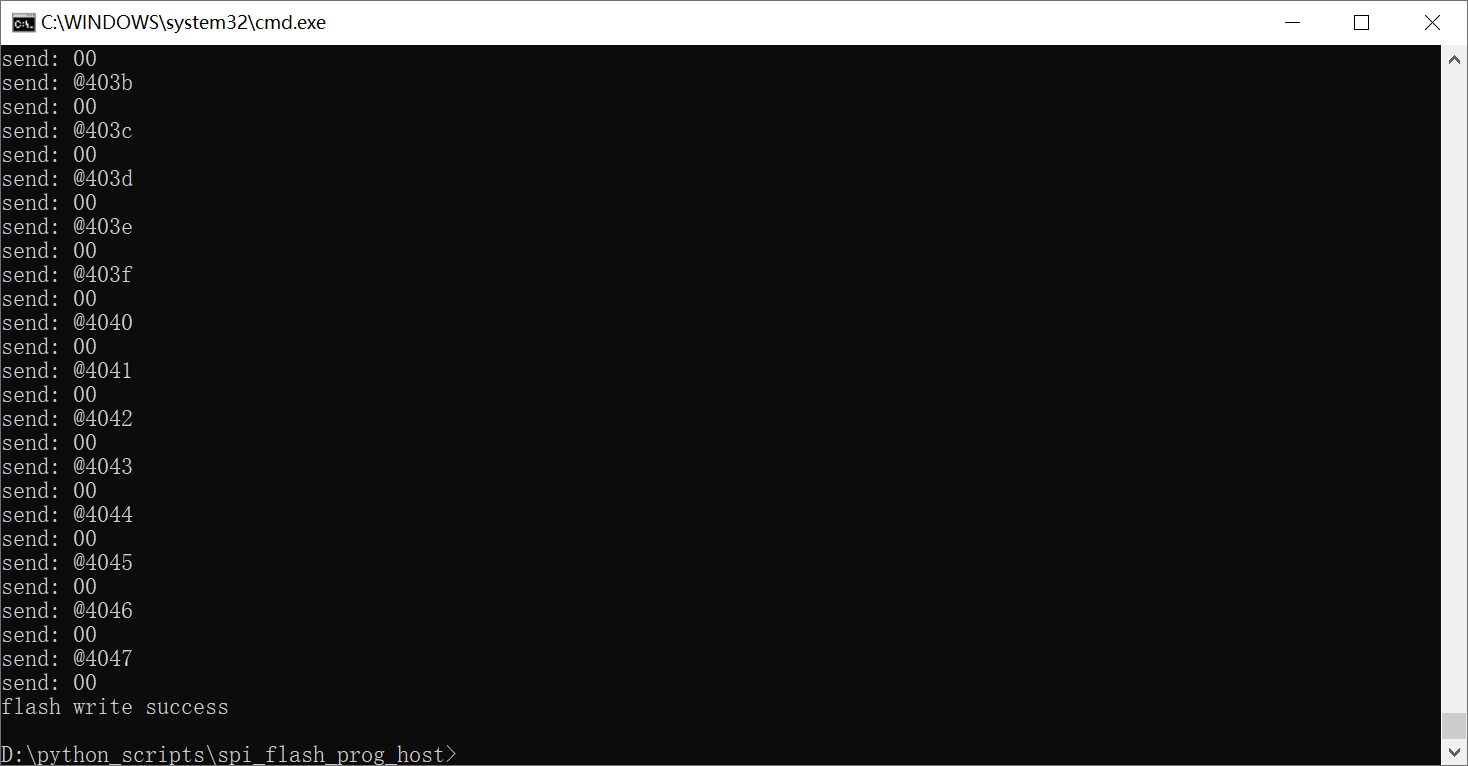
### SPI flash模块程序烧写

host主机程序跟arduino UNO连接成功之后会在cmd命令行终端显示如下信息:



flash刷写打印信息:

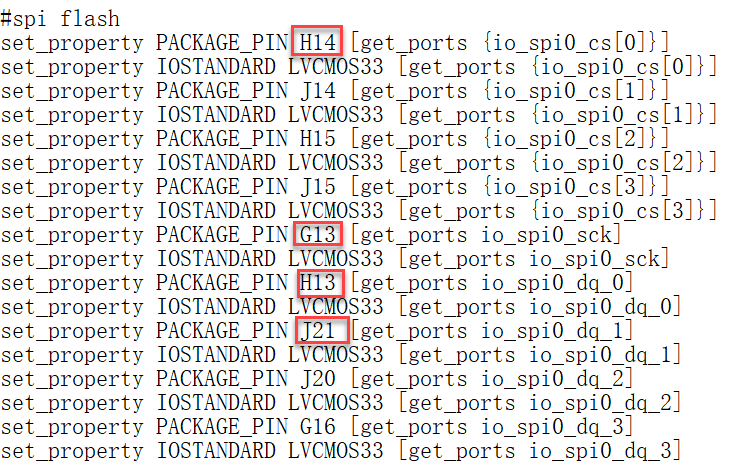




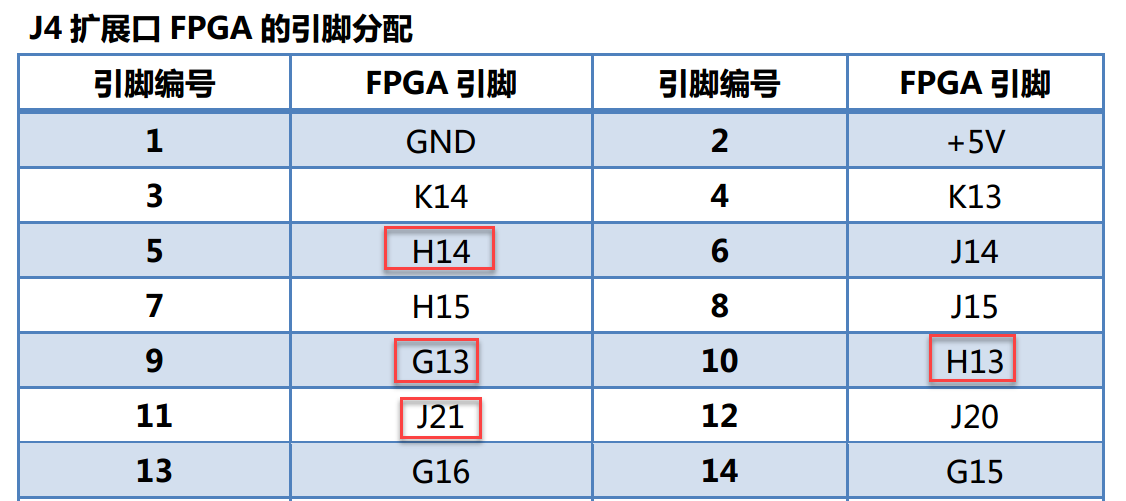
## FPGA开发板上电启动zqh\_riscv

### zqh\_riscv SPI0接口连接SPI flash模块

spi0端口(io\_spi0\_dq\_0为MOSI，io\_spi0\_dq\_1为MISO):



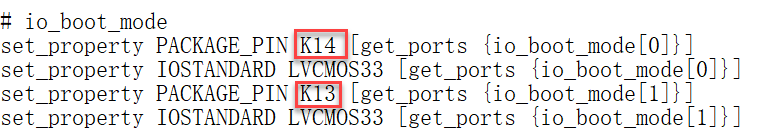
上述端口在FPGA开发板的J4扩展口上:



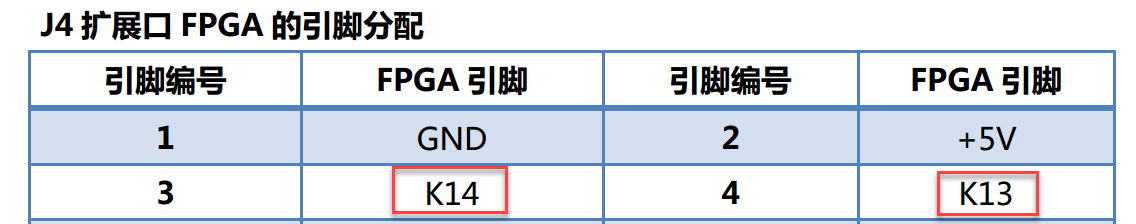
### zqh\_riscv boot\_mode端口IO都接地

boot\_mode=0表示软件从SPI flash XIP启动

boot\_mode端口:

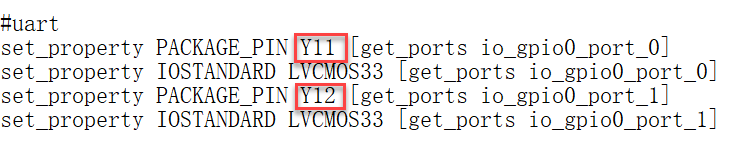


上述端口在FPGA开发板的J4扩展口上:

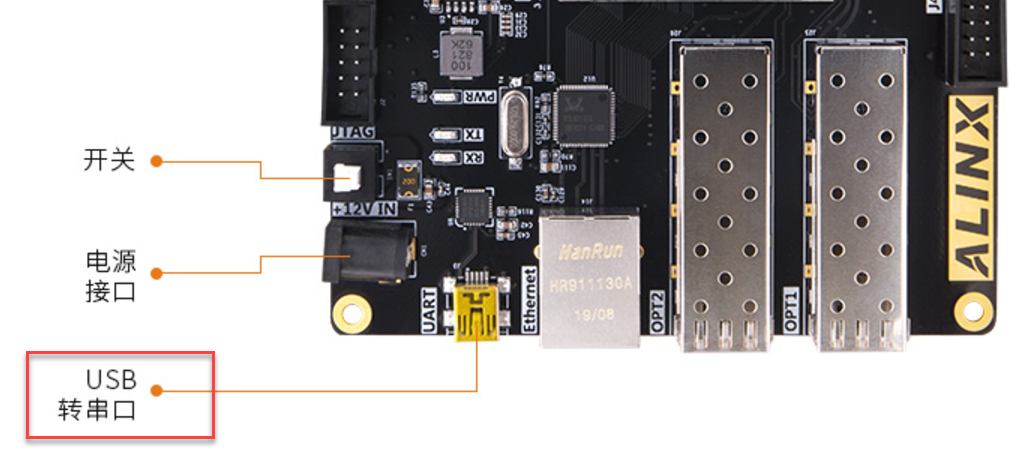


### zqh\_riscv的uart0接口连接PC host

uart0端口(io\_gpio0\_port\_0为tx, io\_gpio0\_port\_1为rx):



上述端口正好是FPGA开发板的usb转串口接口，直接用usb mini线连接PC host即可。

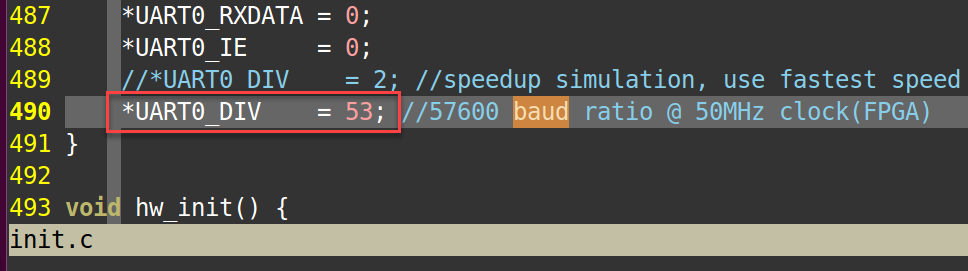


PC host里打开串口终端窗口，波特率设置为57600(zqh\_riscv主时钟为50MHz时)。

如果要修改波特率，则需要修改bloatloader代码，重新编译并烧写SPI flash。

波特率修改:

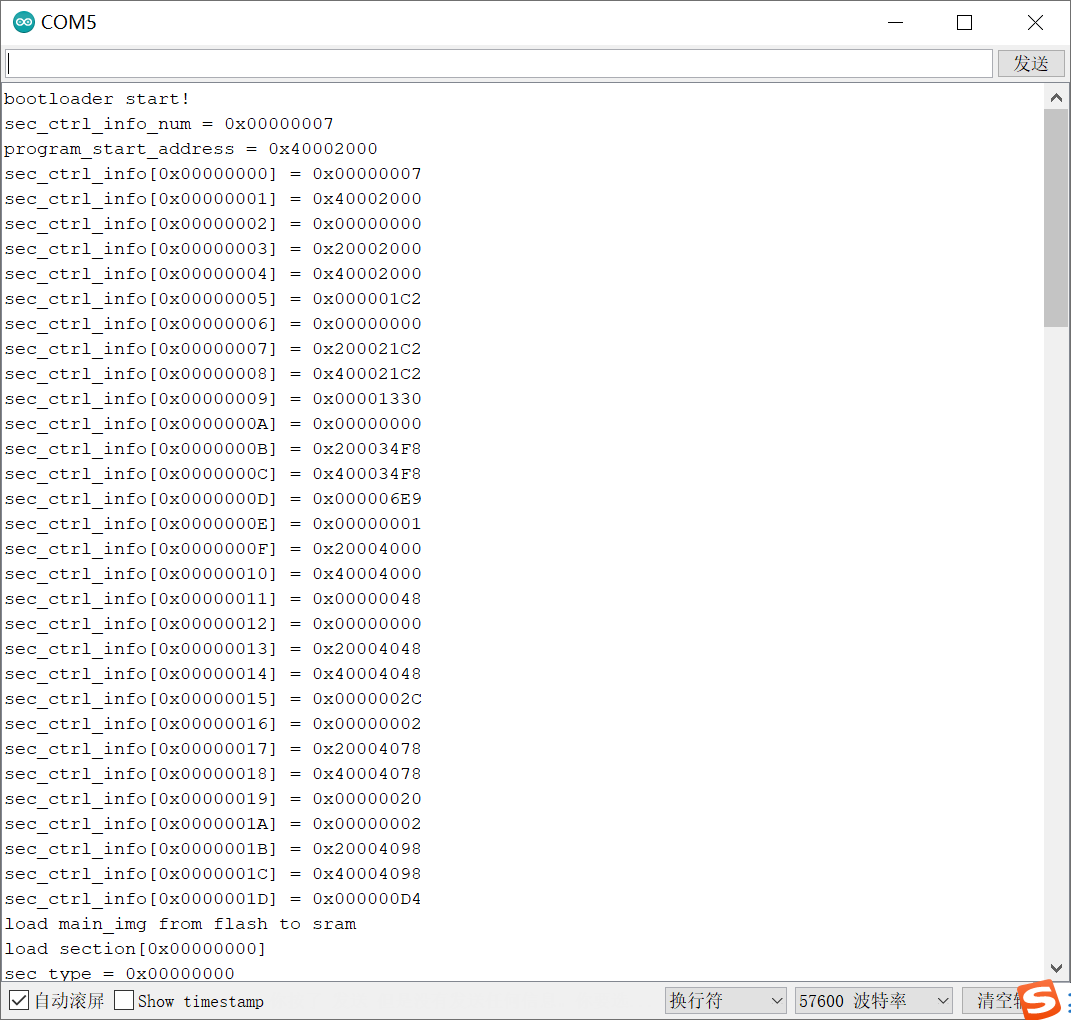




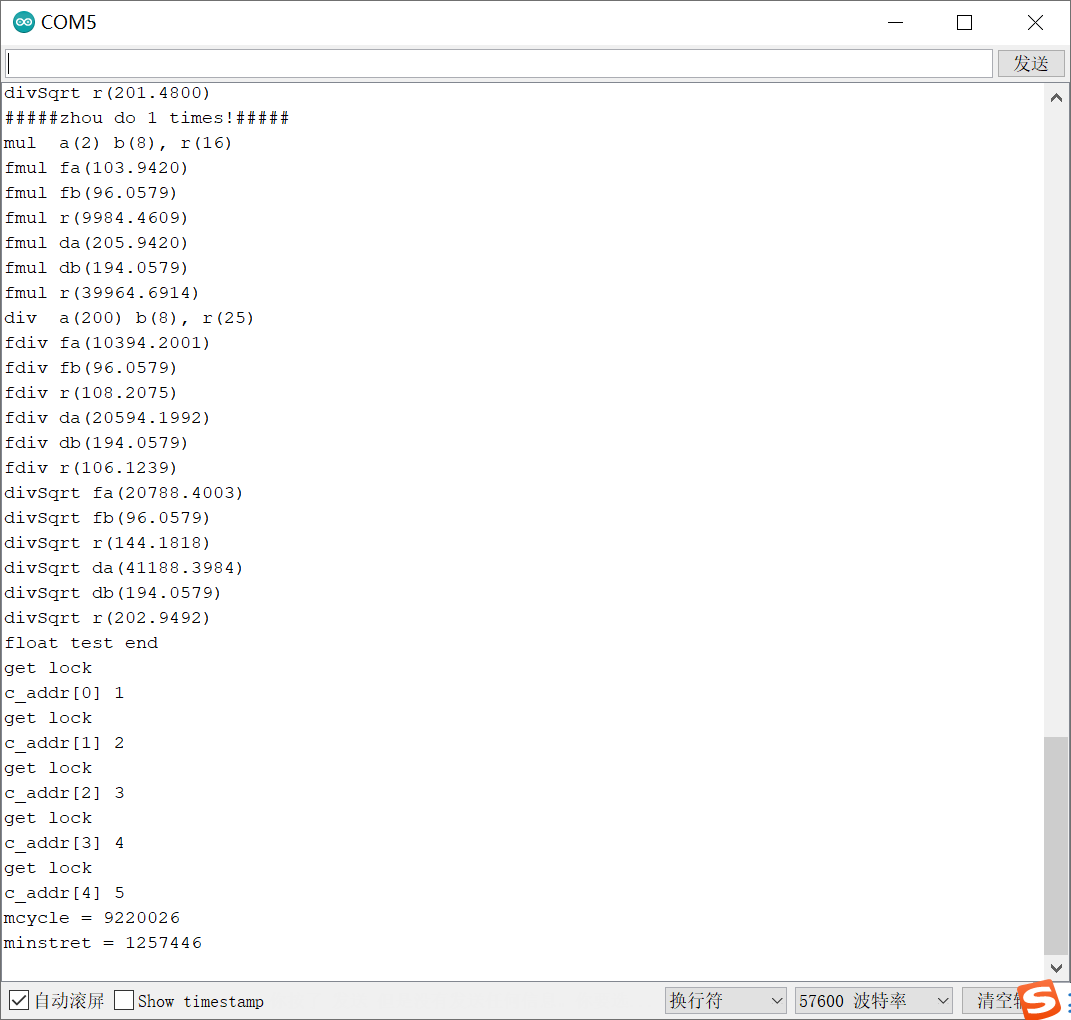
baudrate = Fin/(16\*(div+1)) //Fin取50MHz

### zqh\_riscv上电启动

上电之后，如果一切正常会在PC host的串口终端里看到如下bootloader的打印信息:



主程序打印如下信息:



## JTAG debug

通过JTAG接口可以debug芯片内部资源，可以使riscv core进入debug状态调试软件。还可以通过(SBA)system access bus接口通过tilelink总线访问片内任意地址空间。详细的debug命令请参考《riscv-debug-release.pdf》。

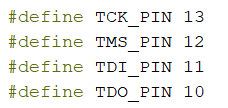
### arduino UNO开发板刷入riscv\_jtag\_debug固件程序



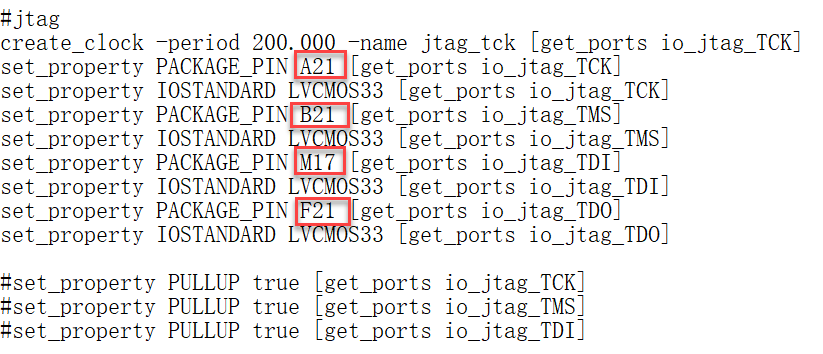
arduino UNO通过IO软件模拟JTAG时序debug zqh\_riscv.

线缆连接arduino UNO的JTAG IO与zqh\_riscv的JTAG端口，注意FPGA的IO电压是3.3v的，arduino的5v IO电平需要转换成3.3v的。

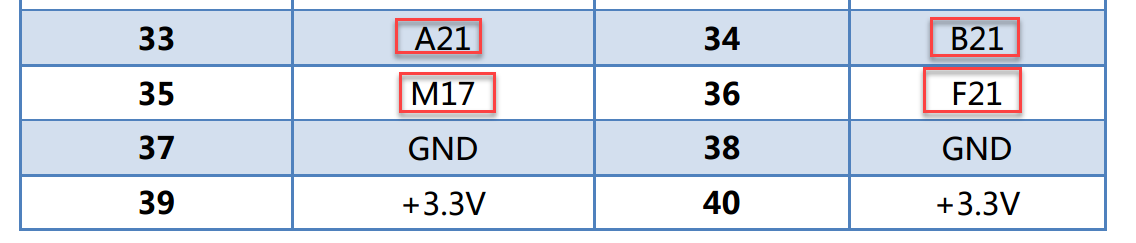
arduino UNO的JTAG IO宽口号:



zqh\_riscv FPGA JTAG接口:



在FPGA开发板的J4扩展口上:



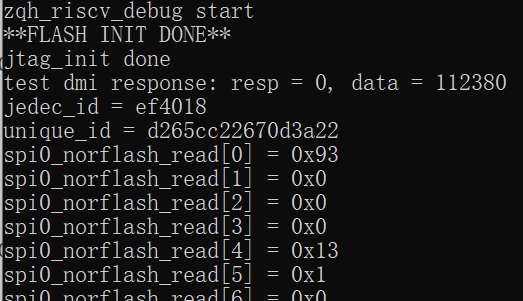
### 命令行启动riscv\_jtag\_debug\_host.py脚本程序

riscv\_jtag\_debug\_host.py通过usb转串口跟arduino UNO通信，发送debug命令给arduino UNO。arduino UNO解析收到的主机debug命令并转换成JTAG接口时序来控制zqh\_riscv。



注意，需要先上电FPGA原型板再运行riscv\_jtag\_debug\_host.py。

如果一切正常，将会在命令行看到如下打印信息:



通过JTAG接口读取SPI flash的jedec\_id与uniquue\_id，接着读取SPI flash的前64字节的数据。

也可以通过JTAG对SPI flash烧写程序，但由于是arduino UNO软件模拟的JTAG时序，烧写速度会比较慢。