OS2ATC 2019 – CPU Tutorial

RISC-V开源处理器及Chisel硬件敏捷开发语言入门

张　科、余子濠、陈欲晓

2019年12月15日

RV教程群

CRVA联盟

中国科学院计算技术研究所
Institute of Computing Technology, Chinese Academy of Sciences

鹏城实验室
Peng Cheng Laboratory

中国开放指令生态(RISC-V)联盟
China RISC-V Alliance

提 纲

9:00-9:15 **1** 开源芯片的缘起

9:15-10:15 **2** RISC-V开源芯片

10:30-11:40 **3** Chisel硬件语言

11:40-11:50 **4** 敏捷开发的愿景

Break Time

本教程涉及的主要概念与术语

结构

- **计算机系统结构/体系结构**：Computer Architecture
- **指令集架构**：Instruction Set Architecture (ISA)
  - 复杂指令集计算机：Complex Instruction Set Computer (CISC)
  - **精简指令集计算机**：Reduced Instruction Set Computer (RISC)
    - 美国加州大学伯克利分校的第五代精简指令集架构：**RISC-V**

芯片

- **集成电路/芯片**：Integrated Circuit (IC) / Chip
  - **芯片设计**：IC design / Chip design，本文所提到的芯片设计有时专指设计、有时则包含设计与验证
  - **芯片验证**：IC verification，本文所提到的芯片验证均指硅前验证，而不包括芯片制造完成后的硅后验证
  - 芯片制造：IC manufacturing，有时专指晶圆制造，有时还包括芯片晶圆加工生产后的芯片封装和测试
- 数字电路芯片：Digital IC
- **中央处理器**/微处理器/处理器核：Central Processing Unit (**CPU**) / microprocessor / core
  - 面向不同新兴领域的处理器加速芯片：例如，图形处理器(GPU)、张量处理器(TPU)及其他xPU
- 处理器微架构：Microarchitecture (μarch)

方法

- **开源**: Open Source
- **敏捷开发**: Agile Development
- **软硬件协同设计**: Hardware-software Co-design
- 现场可编程门阵列芯片: Field-Programmable Gate Array (**FPGA**)，**硬件描述语言 (Verilog/Chisel)**，**EDA**
- 云计算: **Cloud** Computing
- 异构计算: Heterogeneous Computing

提 纲

1 开源芯片的缘起

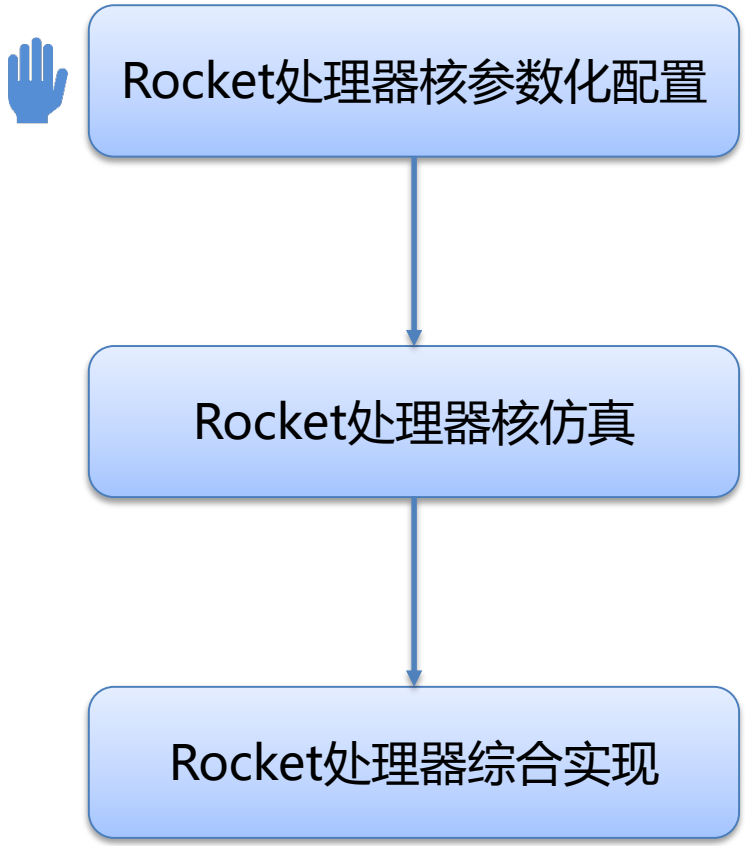
2 RISC-V开源芯片

3 Chisel硬件语言

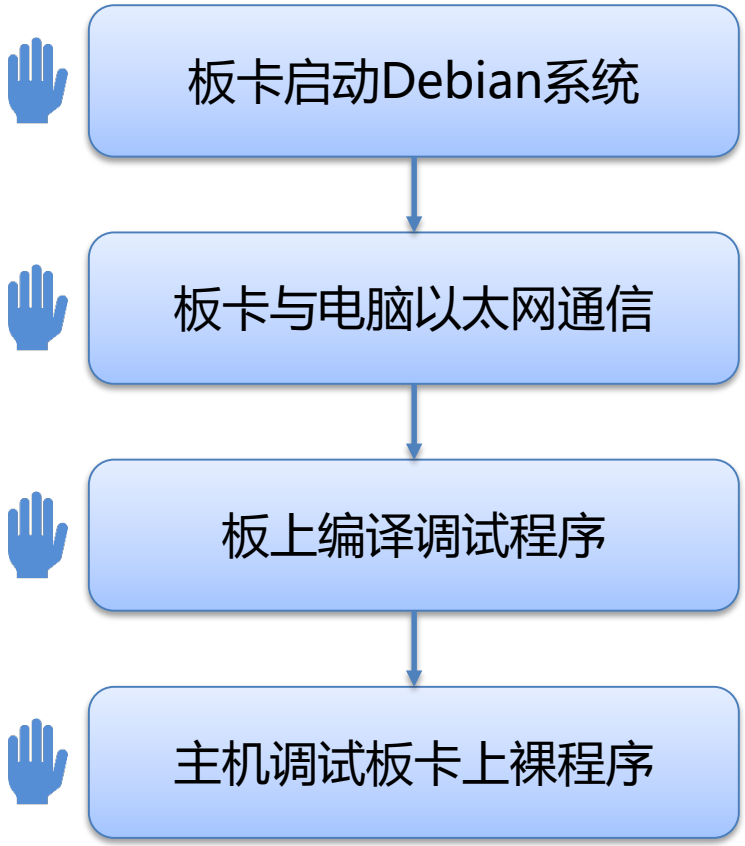
4 敏捷开发的愿景

第一部分实验内容

硬件实验



软件实验



工具准备

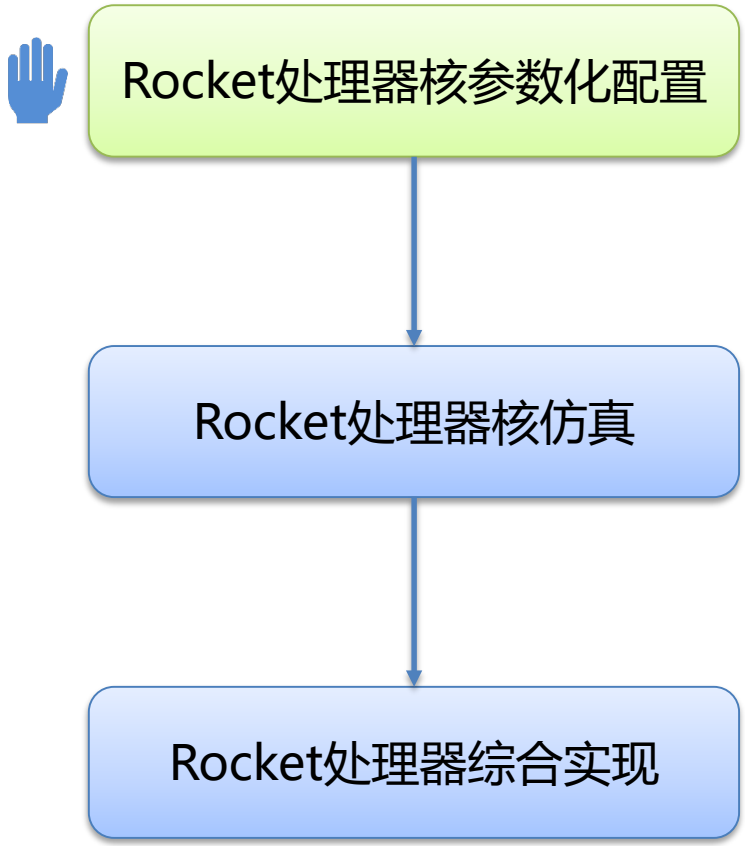
- 系统：Ubuntu 18.04.3 LTS

- 分发的SD卡第一分区包含以下文件或目录：
  - toolchain.tar.gz：已编译的RISC-V GNU工具链+OpenOCD
  - openocd：包含OpenOCD启动脚本
  - baremetal：包含一个已编译的直接运行于RISC-V硬件平台上的裸程序

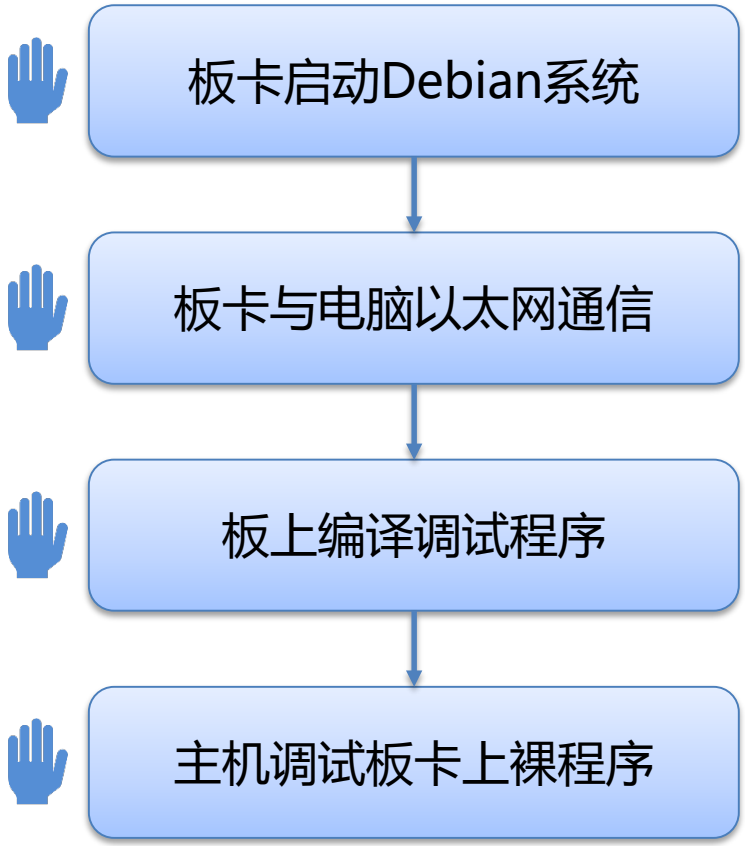
- 将以上文件复制到本机，安装需要的软件并解压工具：
- **$ sudo apt-get install git scala make device-tree-compiler python**
- **$ sudo apt-get install libusb-0.1-4 libftdi1 minicom**
- **$ sudo mkdir /opt/riscv-serve**
- **$ sudo tar xzvf toolchain.tar.gz -C /opt/riscv-serve**

第一部分实验内容

硬件实验



软件实验



RISC-V指令集

Base	Version	Status
RVWMO	2.0	**Ratified**
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*
Extension	Version	Status
Zifencei	**2.0**	**Ratified**
Zicsr	**2.0**	**Ratified**
M	**2.0**	**Ratified**
A	*2.0*	Frozen
F	**2.2**	**Ratified**
D	**2.2**	**Ratified**
Q	**2.2**	**Ratified**
C	**2.0**	**Ratified**
Ztso	*0.1*	*Frozen*
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.2*	*Draft*
V	*0.7*	*Draft*
N	*1.1*	*Draft*
Zam	*0.1*	*Draft*

基本与扩展指令集

Subset	Name	Implies
Base ISA		
Integer	I	
Reduced Integer	E	
Standard Unprivileged Extensions		
Integer Multiplication and Division	M	
Atomics	A	
Single-Precision Floating-Point	F	Zicsr
Double-Precision Floating-Point	D	F
Control and Status Register Access	Zicsr	
Instruction-Fetch Fence	Zifencei	
General	G	IMADZifencei
Quad-Precision Floating-Point	Q	D
Decimal Floating-Point	L	
16-bit Compressed Instructions	C	
Bit Manipulation	B	
Dynamic Languages	J	
Transactional Memory	T	
Packed-SIMD Extensions	P	
Vector Extensions	V	
User-Level Interrupts	N	
Misaligned Atomics	Zam	A
Total Store Ordering	Ztso	

部分扩展指令集代号含义

RISC-V处理器开源实现

- 主要的RISC-V处理器开源实现
  - 节选自https://riscv.org/risc-v-cores/

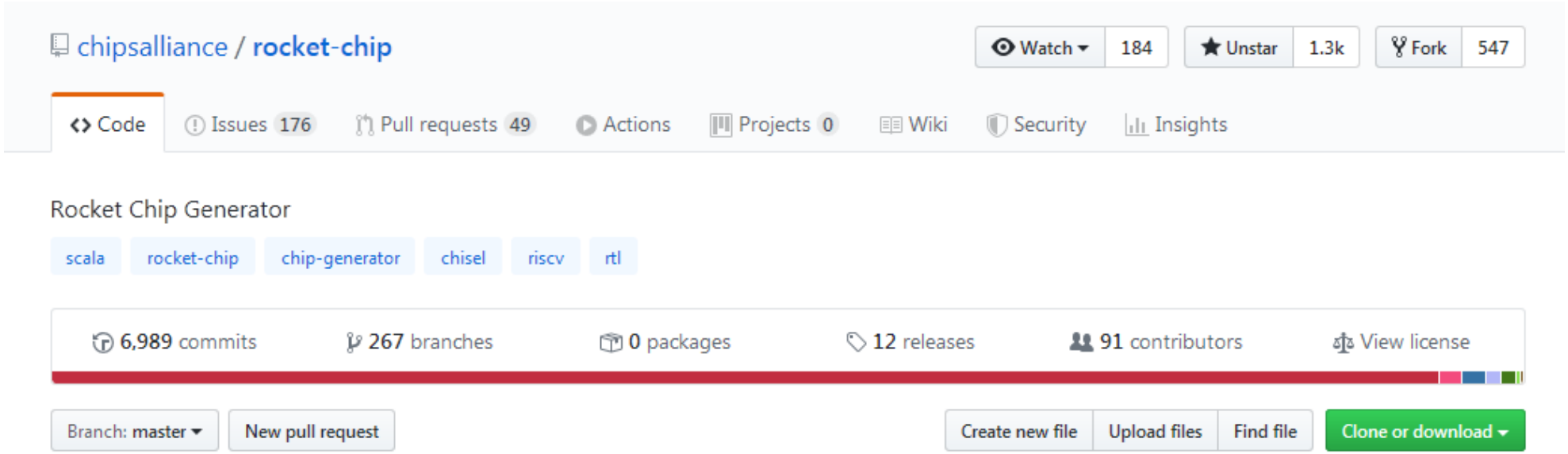
名称	ISA	项目网址	开发语言
PicoRV32	RV32I/E[MC]	https://github.com/cliffordwolf/picorv32	Verilog
OPenV/mriscv	RV32I	https://github.com/onchipuis/mriscv	Verilog
Hummingbird E200	RV32IMAC	https://github.com/SI-RISCV/e200_opensource	Verilog
RI5CY	RV32IMC	https://github.com/pulp-platform/riscv	SystemVerilog
Zero-riscy	RV32IMC	https://github.com/lowRISC/ibex	SystemVerilog
Rocket	RV32G/RV64G	https://github.com/chipsalliance/rocket-chip	Chisel

Rocket Chip

- Rocket Chip是UC Berkeley开发的参数化RISC-V核生成器，具有以下特点：
  - 采用Chisel语言编写
  - 包含MMU、Cache、浮点单元等组件，为运行Linux发行版操作系统提供支持
  - 可配置参数，指定核心数量及Cache、TLB等部件的容量，对功能单元进行裁剪，以适应不同需求
  - 设计结构层次清晰，为增加自定义指令和加速器提供框架
  - 经过流片验证

实验：Rocket处理器核参数化配置 - 获取代码

- 克隆Rocket-Chip代码仓库：
- `$ git clone https://github.com/chipsalliance/rocket-chip.git`
- `$ cd rocket-chip`
- `$ git checkout v1.2.1`
- `$ git submodule update --init --recursive .`



补充：硬件描述语言（HDL）

- HDL：描述电子硬件系统的结构、行为、数据流的语言
- Verilog（1984年）
  - 最常用的HDL之一，广泛用于业界
  - 语法近似于C，易于学习
  - 可以在多个层次描述设计，提供模块化抽象和封装功能
- SystemVerilog（2002年）
  - 建立在Verilog基础之上，将设计语言与验证语言相结合
  - 比Verilog抽象层次更高，设计建模能力更强
- Chisel（～2010年）
  - 基于Scala的HDL
  - 将面向对象编程和函数式编程思想融入硬件设计，代码更加简洁，参数化能力更强
  - 可将设计转化为Verilog等低层次描述
  - Jonathan Bachrach, et al. “Chisel: Constructing Hardware in a Scala Embedded Language”, IEEE DAC Design Automation Conference 2012

实验：Rocket处理器核参数化配置 - 更改参数

修改src/main/scala/system/Configs.scala，文件最后位置：

```
class BaseFPGAConfig extends Config(new BaseConfig)
class DefaultFPGAConfig extends Config(new WithNSmallCores(1) ++ new BaseFPGAConfig)
```

- 将**WithNSmallCores(1)**改为**WithNSmallCores(2)**
- 使得处理器具有**两个小规模配置的RISC-V核**

* 关于WithNSmallCores对应的详细配置，请参考src/main/scala/subsystem/Configs.scala

实验：Rocket处理器核参数化配置 - 编译代码

- 编译Rocket-Chip并生成Verilog代码（**第一次运行时间较长**）
- **$ export RISCV=/opt/riscv-serve**
- **$ cd vsim**
- **$ make verilog CONFIG=DefaultFPGAConfig**（若第一次运行遇到错误请重试）
- 完成后在vsim/generated-src目录下生成后缀为.v的Verilog源文件

- (1) 过程中可以看到生成的设备树，其中描述了两个CPU（右图）
- (2) 打开生成的Verilog 代码文件，搜索“module ExampleRocketSystem”，在该模块内找到RocketTile（Rocket核心的封装）的两次实例化，说明生成的代码中具有两个核

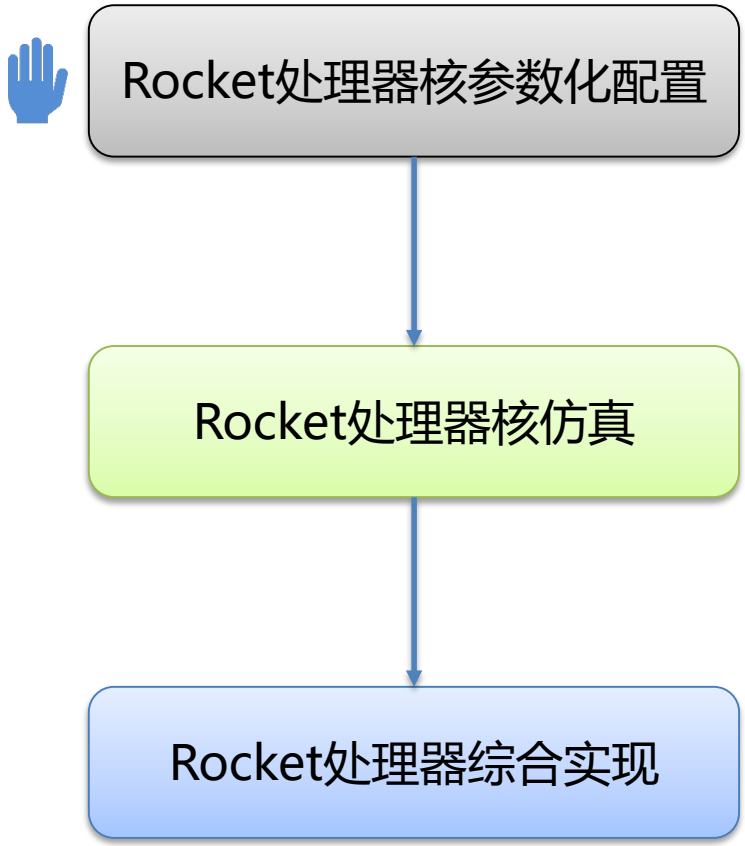
```
191394    RocketTile tile ( // @[RocketSubsystem.scala 42:28:
191395      .clock(tile_clock),
191396      .reset(tile_reset),
```

```
191454    RocketTile tile_1 ( // @[RocketSubsystem.scala 42:28
191455      .clock(tile_1_clock),
191456      .reset(tile_1_reset),
```

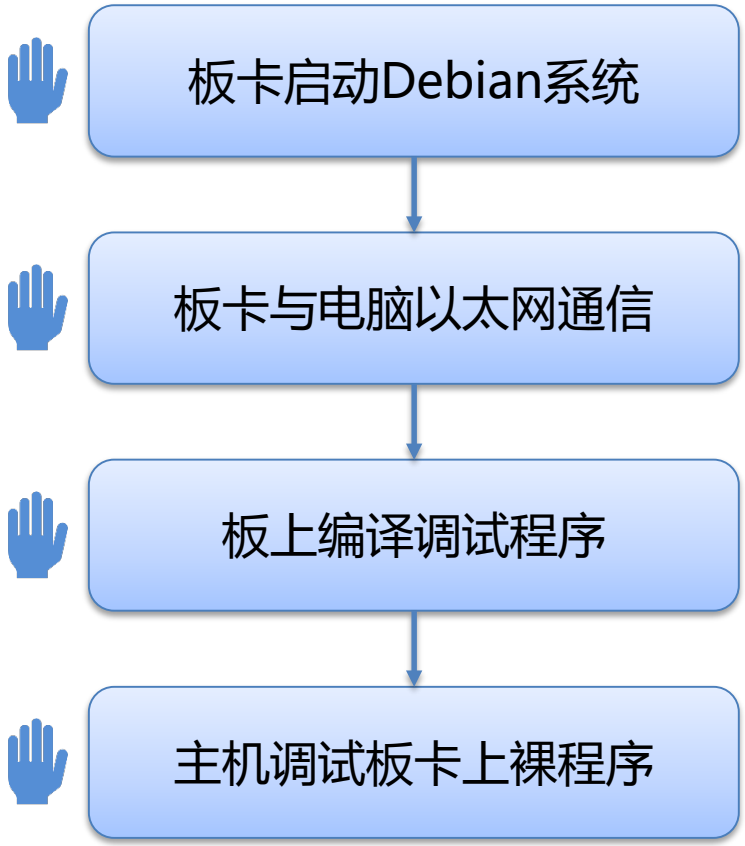
```
L6: cpu@0 {
        clock-frequency = <0>;
        compatible = "sifive,rocket0", "riscv";
        d-cache-block-size = <64>;
        d-cache-sets = <64>;
        d-cache-size = <4096>;
        device_type = "cpu";
        hardware-exec-breakpoint-count = <1>;
        i-cache-block-size = <64>;
        i-cache-sets = <64>;
        i-cache-size = <4096>;
        next-level-cache = <&L11>;
        reg = <0x0>;
        riscv,isa = "rv64imac";
        riscv,pmpregions = <8>;
        status = "okay";
        timebase-frequency = <1000000>;
        L4: interrupt-controller {
                #interrupt-cells = <1>;
                compatible = "riscv,cpu-intc";
                interrupt-controller;
        };
};
L9: cpu@1 {
        clock-frequency = <0>;
        compatible = "sifive,rocket0", "riscv";
        d-cache-block-size = <64>;
        d-cache-sets = <64>;
        d-cache-size = <4096>;
        device_type = "cpu";
        hardware-exec-breakpoint-count = <1>;
        i-cache-block-size = <64>;
        i-cache-sets = <64>;
        i-cache-size = <4096>;
        next-level-cache = <&L11>;
        reg = <0x1>;
        riscv,isa = "rv64imac";
        riscv,pmpregions = <8>;
        status = "okay";
        timebase-frequency = <1000000>;
        L7: interrupt-controller {
                #interrupt-cells = <1>;
                compatible = "riscv,cpu-intc";
                interrupt-controller;
        };
```

第一部分实验内容

硬件实验



软件实验



Rocket处理器核仿真

- 编译安装rocket-tools
- **$ git clone https://github.com/chipsalliance/rocket-tools.git**
- **$ cd rocket-tools**
- **$ git submodule update --init --recursive**
- **$ export RISCV=/opt/riscv**
- **$ sudo ./build.sh**

- 安装verilator（Verilog仿真器）
- **$ sudo apt install verilator**

- 在rocket-chip仓库根目录下执行
- **$ export RISCV=/opt/riscv**
- **$ cd emulator**
- **$ make**
- **$ make run-asm-tests**
- **$ make run-bmark-tests**
- 上述命令运行指令测试和benchmark测试，并在output目录下生成仿真结果

- 详细步骤参考rocket-chip的README.md

运行benchmark效果

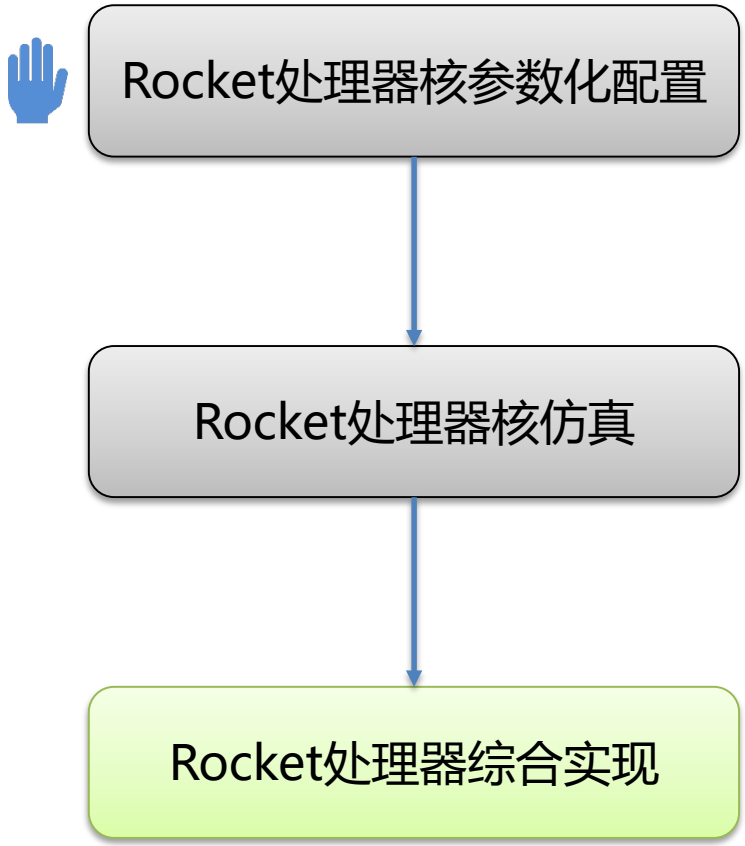
```
Microseconds for one run through Dhrystone: 491
matmul(cid, nc, 16, input1_data, input2_data, results_data); barDhrystones per
econd:                        2035
rier(nc): 41074 cycles, 10.0 cycles/iter, 1.5 CPI
mcycle = 245708
minstret = 198773

  [ PASSED ] output/mm.riscv.out         Completed after 763866 cycles
  [ PASSED ] output/spmv.riscv.out       Completed after 895314 cycles
  [ PASSED ] output/mt-vvadd.riscv.out   Completed after 723434 cycles
  [ PASSED ] output/median.riscv.out     Completed after 162330 cycles
  [ PASSED ] output/multiply.riscv.out   Completed after 169990 cycles
  [ PASSED ] output/qsort.riscv.out      Completed after 725558 cycles
  [ PASSED ] output/towers.riscv.out     Completed after 120534 cycles
  [ PASSED ] output/vvadd.riscv.out      Completed after 169614 cycles
  [ PASSED ] output/dhrystone.riscv.out          Completed after 471314 cycles
  [ PASSED ] output/mt-matmul.riscv.out          Completed after 275798 cycles
(10:28:59)free@free-ThinkPad-T450:~/workspace/rocket-chip/emulator$
```

仿真过程生成的trace

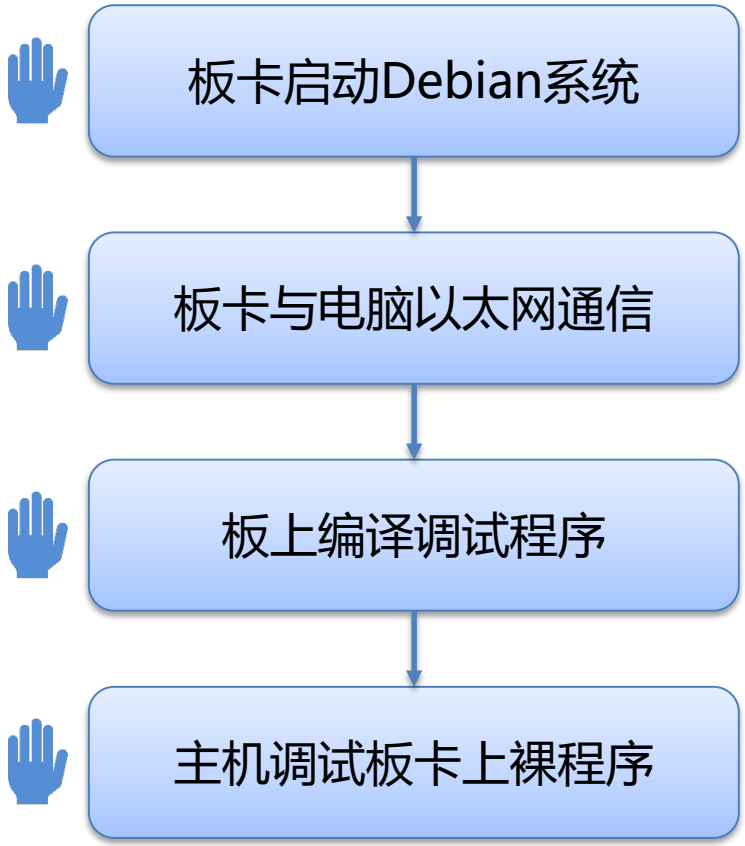
```
inst=[fea791e3] bne     a5, a0, pc - 30
inst=[0009c783] lbu     a5, 0(s3)
inst=[00002b05] c.addiw s6, 1
inst=[0ffb7b13] andi    s6, s6, 255
inst=[0767e163] bltu    a5, s6, pc + 98
inst=[0767e163] bltu    a5, s6, pc + 98
inst=[0767e163] bltu    a5, s6, pc + 98
inst=[0767e163] bltu    a5, s6, pc + 98
inst=[04300593] li      a1, 67
inst=[0000855a] c.mv    a0, s6
inst=[c58ff0ef] jal     pc - 0xba8
inst=[c58ff0ef] jal     pc - 0xba8
inst=[c58ff0ef] jal     pc - 0xba8
inst=[0ff57513] andi    a0, a0, 255
inst=[0ff5f593] andi    a1, a1, 255
inst=[00b50463] beq     a0, a1, pc + 8
inst=[00004501] c.li    a0, 0
inst=[00008082] ret
inst=[f4c42783] lw      a5, -180(s0)
inst=[00002501] c.addiw a0, 0
inst=[fea791e3] bne     a5, a0, pc - 30
inst=[fea791e3] bne     a5, a0, pc - 30
inst=[0009c783] lbu     a5, 0(s3)
inst=[00002b05] c.addiw s6, 1
inst=[0ffb7b13] andi    s6, s6, 255
inst=[0767e163] bltu    a5, s6, pc + 98
inst=[0767e163] bltu    a5, s6, pc + 98
inst=[0767e163] bltu    a5, s6, pc + 98
inst=[0767e163] bltu    a5, s6, pc + 98
inst=[0767e163] bltu    a5, s6, pc + 98
inst=[f4442783] lw      a5, -188(s0)
inst=[f4842b03] lw      s6, -184(s0)
inst=[00002a85] c.addiw s5, 1
inst=[03b78dbb] mulw    s11, a5, s11
inst=[f4440513] addi    a0, s0, -188
inst=[f4440513] addi    a0, s0, -188
inst=[f4440513] addi    a0, s0, -188
inst=[f4440513] addi    a0, s0, -188
inst=[036dc73b] divw    a4, s11, s6
inst=[036dc73b] divw    a4, s11, s6
inst=[036dc73b] divw    a4, s11, s6
inst=[036dc73b] divw    a4, s11, s6
inst=[036dc73b] divw    a4, s11, s6
inst=[036dc73b] divw    a4, s11, s6
inst=[f2e43c23] sd      a4, -200(s0)
inst=[f4e42223] sw      a4, -188(s0)
inst=[cd4ff0ef] jal     pc - 0xb2c
inst=[00001717] auipc   a4, 0x1
inst=[74d74703] lbu     a4, 1869(a4)
inst=[04100793] li      a5, 65
inst=[04100793] li      a5, 65
inst=[00f70363] beq     a4, a5, pc + 6
inst=[0000411c] c.lw    a5, 0(a0)
```

第一部分实验内容

硬件实验

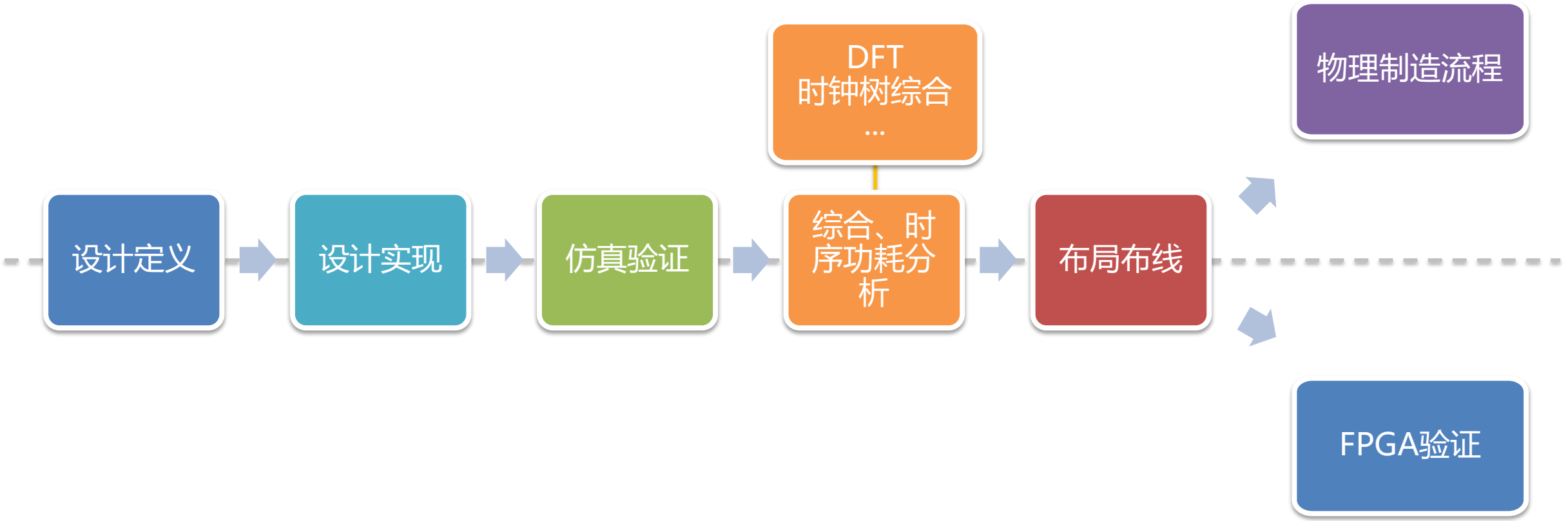


软件实验



让RISC-V处理器真正运行起来

芯片设计

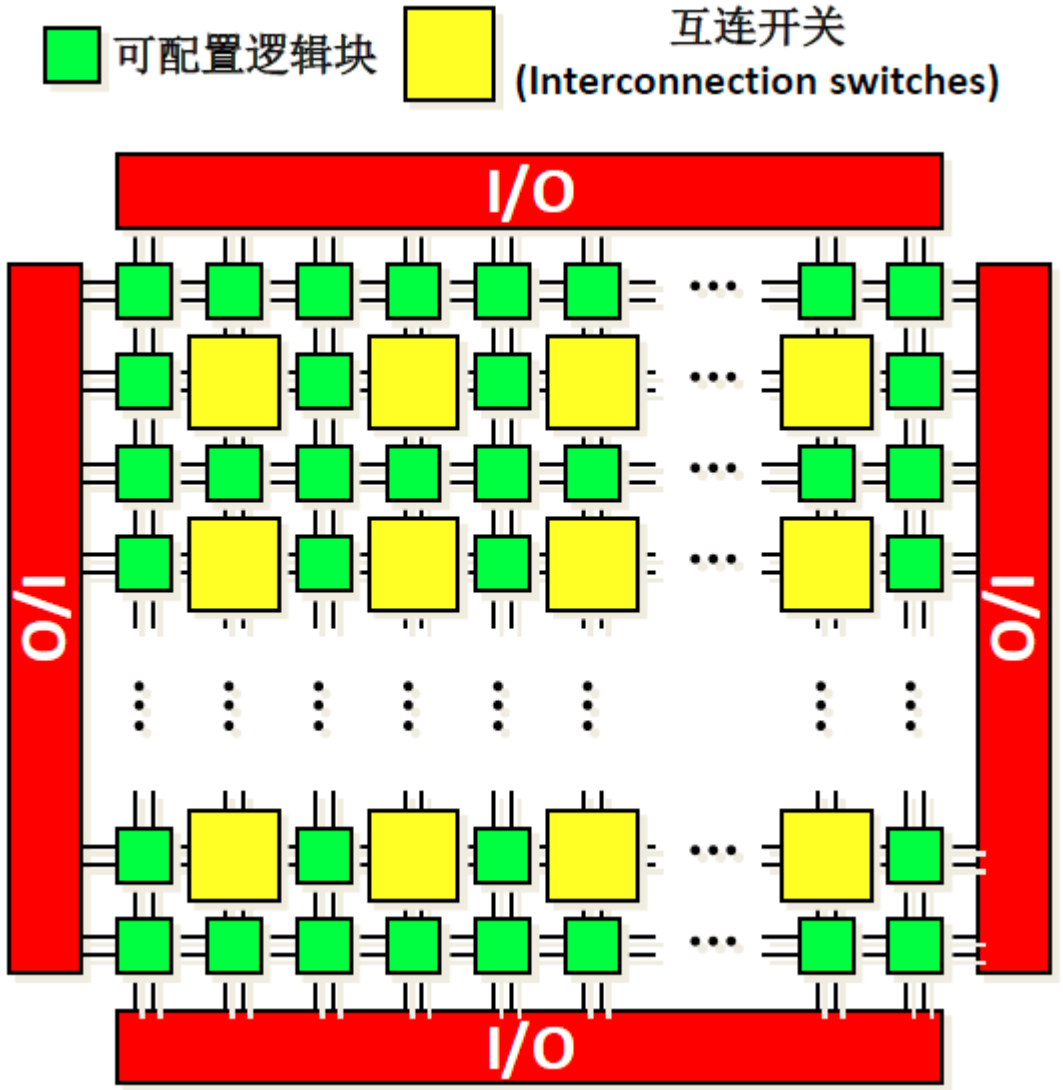


FPGA原型

FPGA基础概念

- 现场可编程逻辑门阵列 （Field Programmable Gate Array）
- 由可配置的逻辑块、输入输出缓冲、互联开关组成
- 相比于原有的可编程器件，FPGA资源量更大

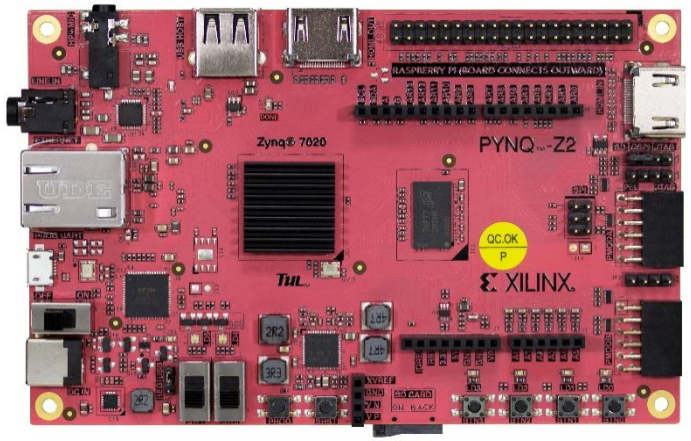
- 主要厂商
  - Xilinx
  - Intel (Altera)
  - 等



Xilinx FPGA开发平台与工具

- 开发板：PYNQ-Z2

- Xilinx Zynq-7020 FPGA芯片
- DDR3 512MB内存
- 板载SD卡、以太网等接口



- 工具：Xilinx Vivado Design Suite

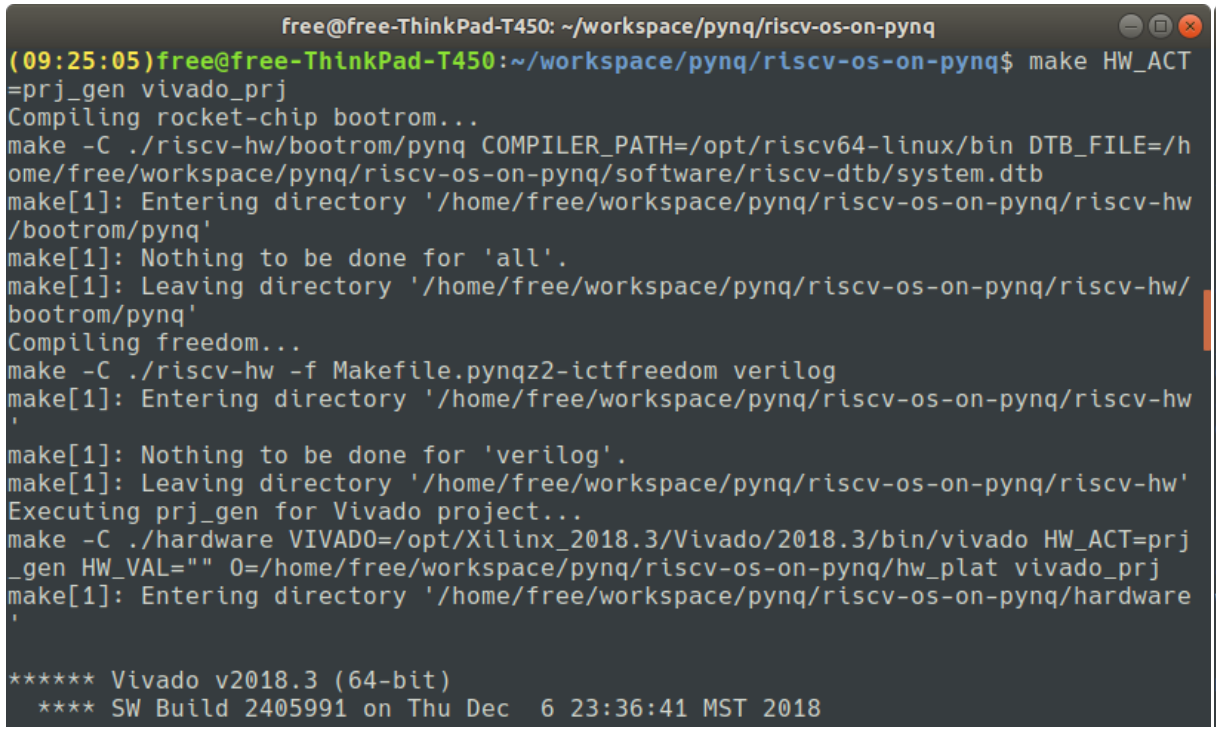
- 项目模式
  - 图形化开发界面
  - 自动管理设计流程和设计数据

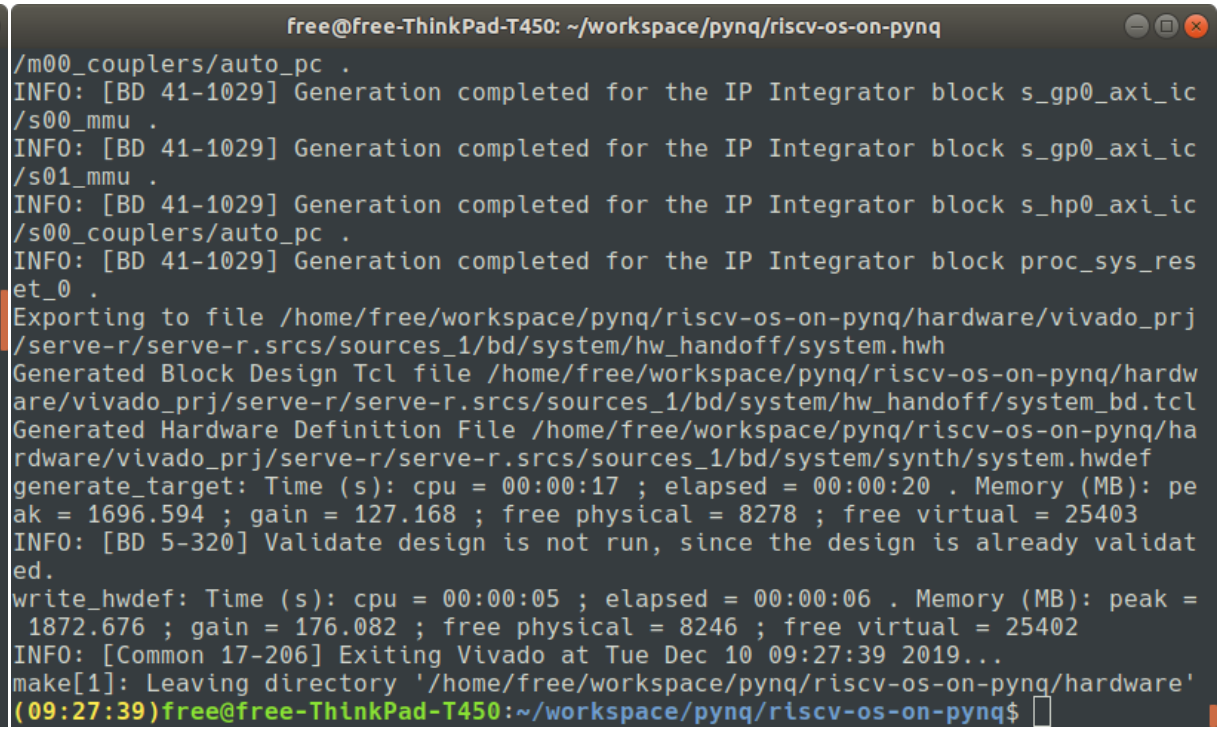
- 非项目模式
  - 基于Tcl命令行或脚本
  - 定制化的开发流程

VIVADO™

Rocket处理器综合实现

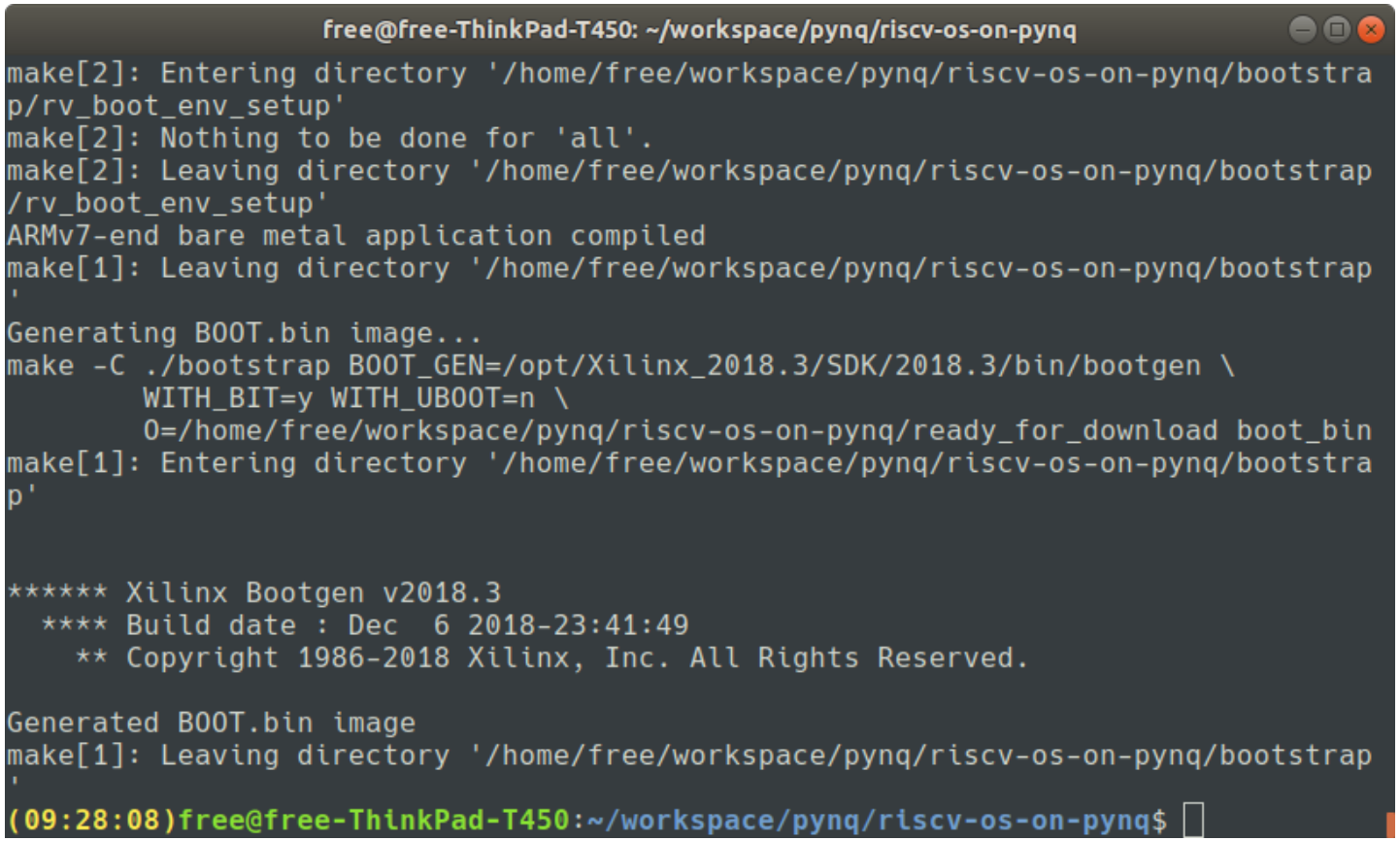
- 通过脚本创建Vivado工程，综合并布局布线（~40min.）



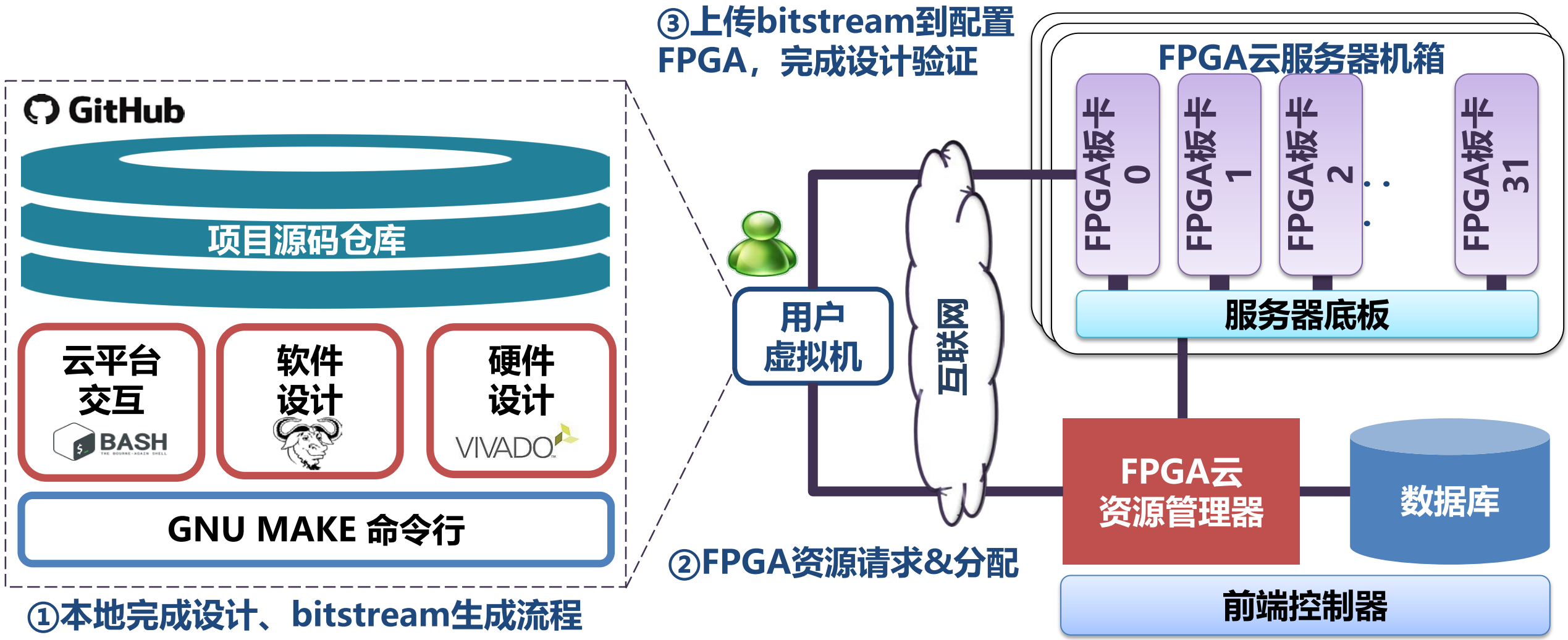


Rocket处理器综合实现

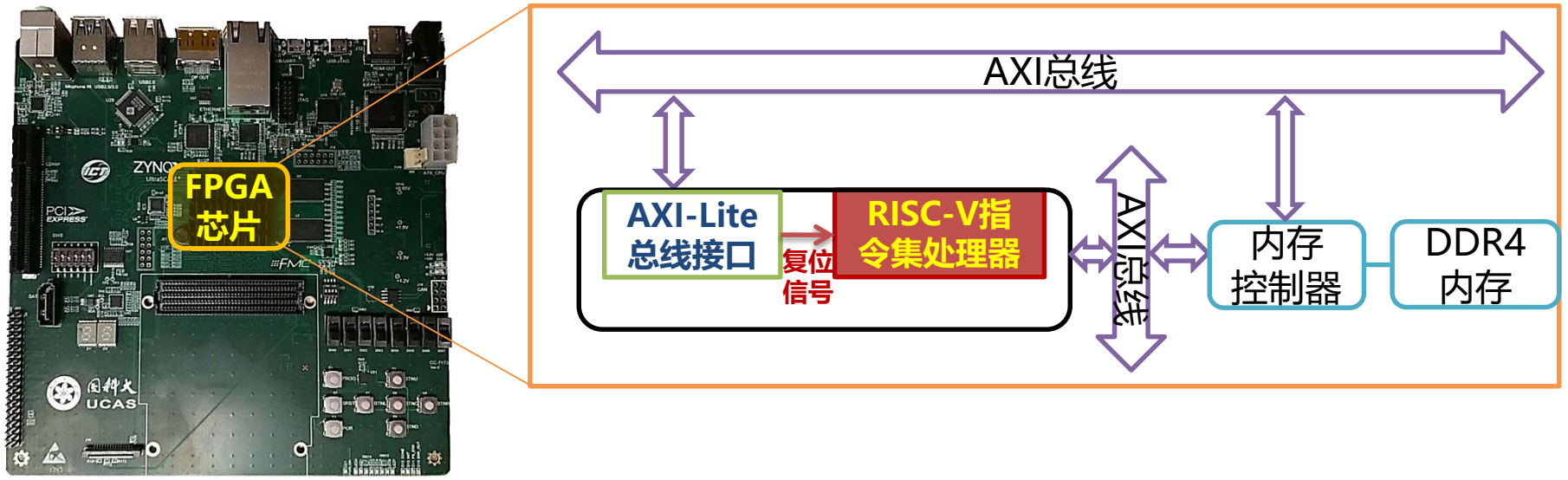
- 生成BOOT.bin（用于通过SD卡配置FPGA）



RISC-V处理器云上原型验证平台（SERVE.c）



RISC-V CPU实验环境及工程框架



- 简单多周期处理器微体系结构（ALU和寄存器堆；控制通路；状态机；指令访问接口；内存读写接口等）
  - 实现可正确译码RV32I指令集中的37条基本指令的译码器逻辑
  - 根据指令功能和访存端口时序设计状态机

- 使用30个benchmark进行RISC-V处理器功能评测

简单RISC-V处理器设计要求

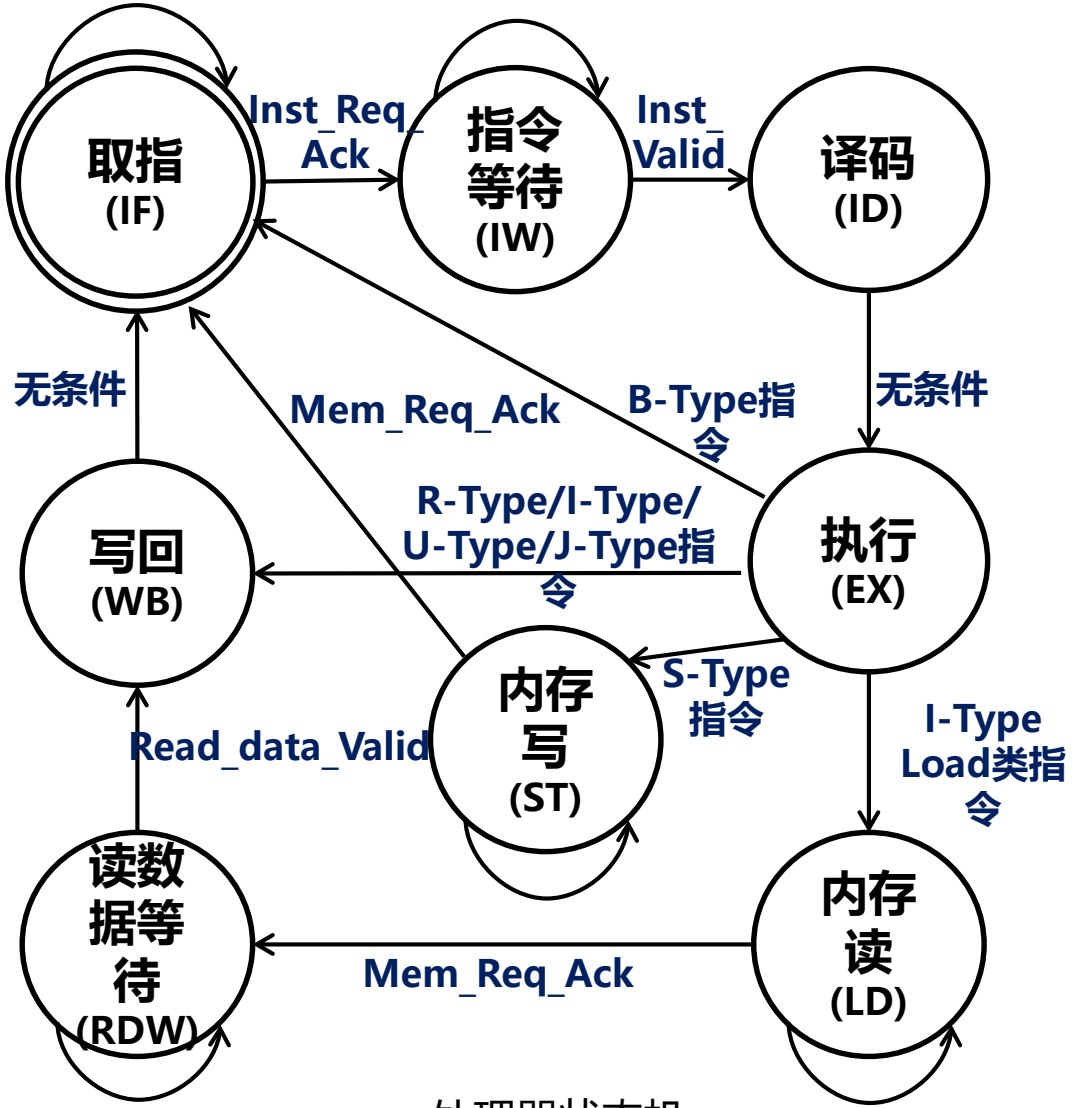
实现37条基本指令

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RISC-V的基本指令格式



处理器状态机

云端设计验证

- 使用云平台对设计的CPU进行验证
  - 自动完成bitstream上传及FPGA配置工作
  - 平台加载测试程序到板卡内存，驱动CPU执行
  - 脚本自动完成验证过程，展示验证结果

FPGA云平台
使用申请表

https://ut9xa2.fanqier.cn/f/n1ctsdzw

```
cod@cod-VirtualBox:~/ucas-cod/prj2-student$ make USER=changyisong HW_VAL="medium" cloud_run
Starting xl2tpd (via systemctl): xl2tpd.service.
Remote target: root@172.16.15.66
Warning: Permanently added '172.16.15.66' (ECDSA) to the list of known hosts.
Completed FPGA configuration
Evaluating medium benchmark suite...
Launching sum benchmark...
Hit good trap
Launching mov-c benchmark...
Hit good trap
Launching fib benchmark...
Hit bad trap
Launching add benchmark...
Hit bad trap
Launching if-else benchmark...
Hit good trap
Launching pascal benchmark...
Hit good trap
Launching quick-sort benchmark...
Hit good trap
Launching select-sort benchmark...
Hit good trap
Launching max benchmark...
Hit good trap
Launching min3 benchmark...
Hit good trap
Launching switch benchmark...
Hit good trap
Launching bubble-sort benchmark...
Hit good trap
pass 10 / 12
Stopping xl2tpd (via systemctl): xl2tpd.service.
```

第一部分实验内容

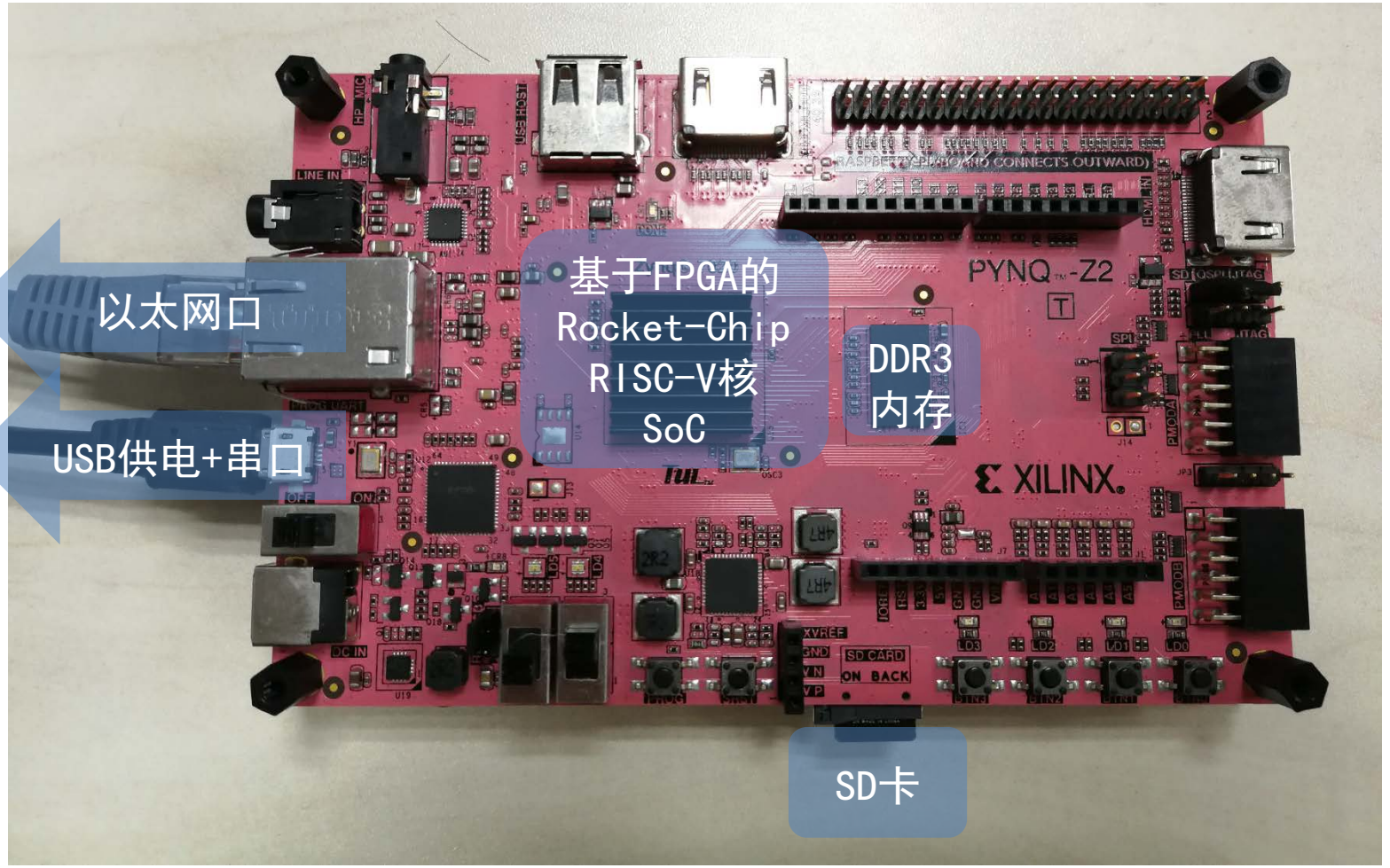
硬件实验

- Rocket处理器核参数化配置
- Rocket处理器核仿真
- Rocket处理器综合实现

软件实验

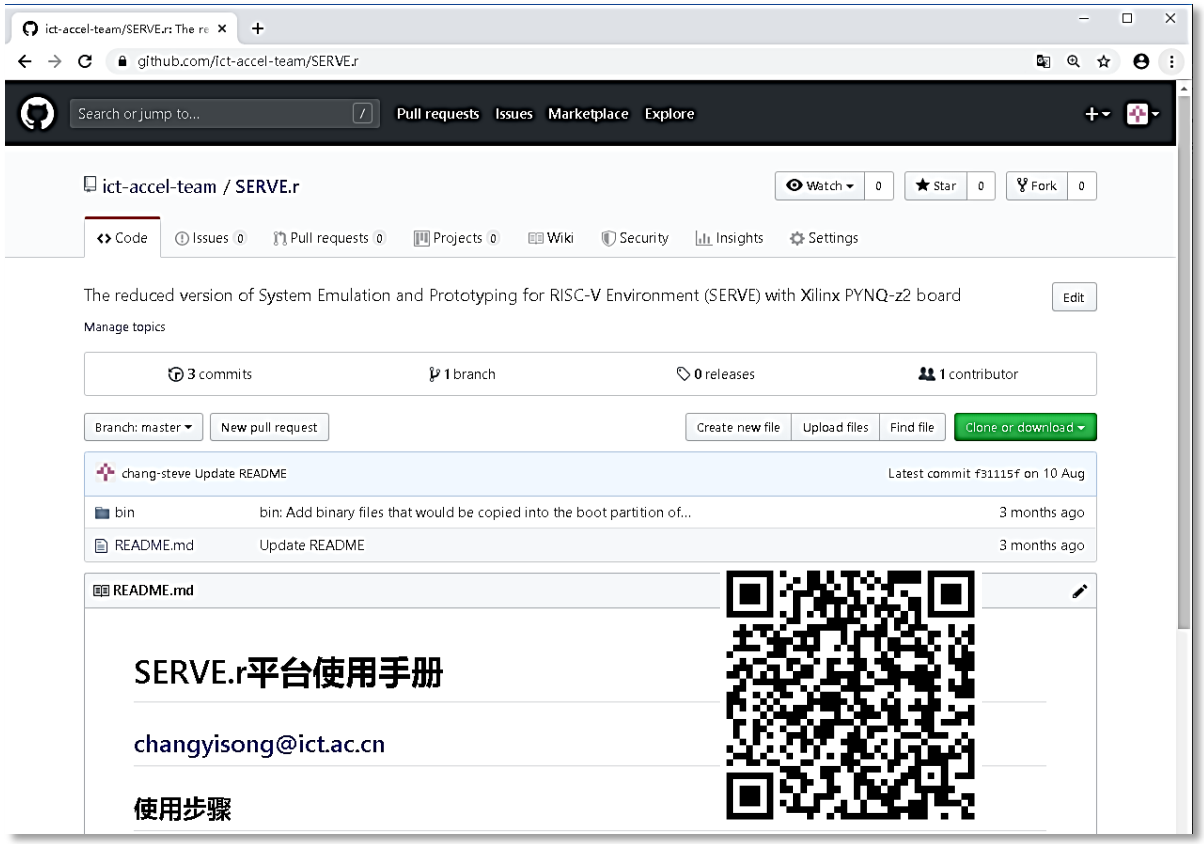
- 板卡启动Debian系统
- 板卡与电脑以太网通信
- 板上编译调试程序
- 主机调试板卡上裸程序

SERVE.r平台


以太网口
USB供电+串口
基于FPGA的
Rocket-Chip
RISC-V核
SoC
DDR3
内存
SD卡
PYNQ™-Z2
XILINX

SERVE.r镜像文件在开源托管平台提供下载

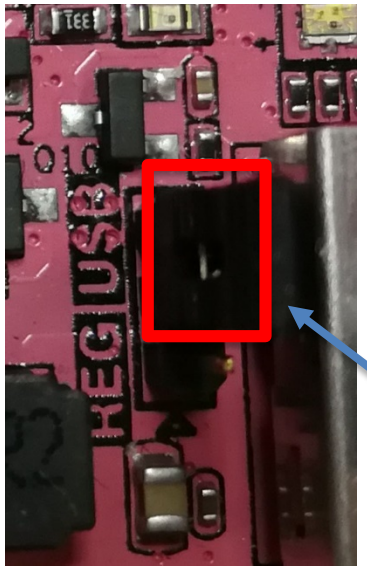
鹏城实验室iHub
https://code.ihub.org.cn/projects/373/repository/SERVE_r



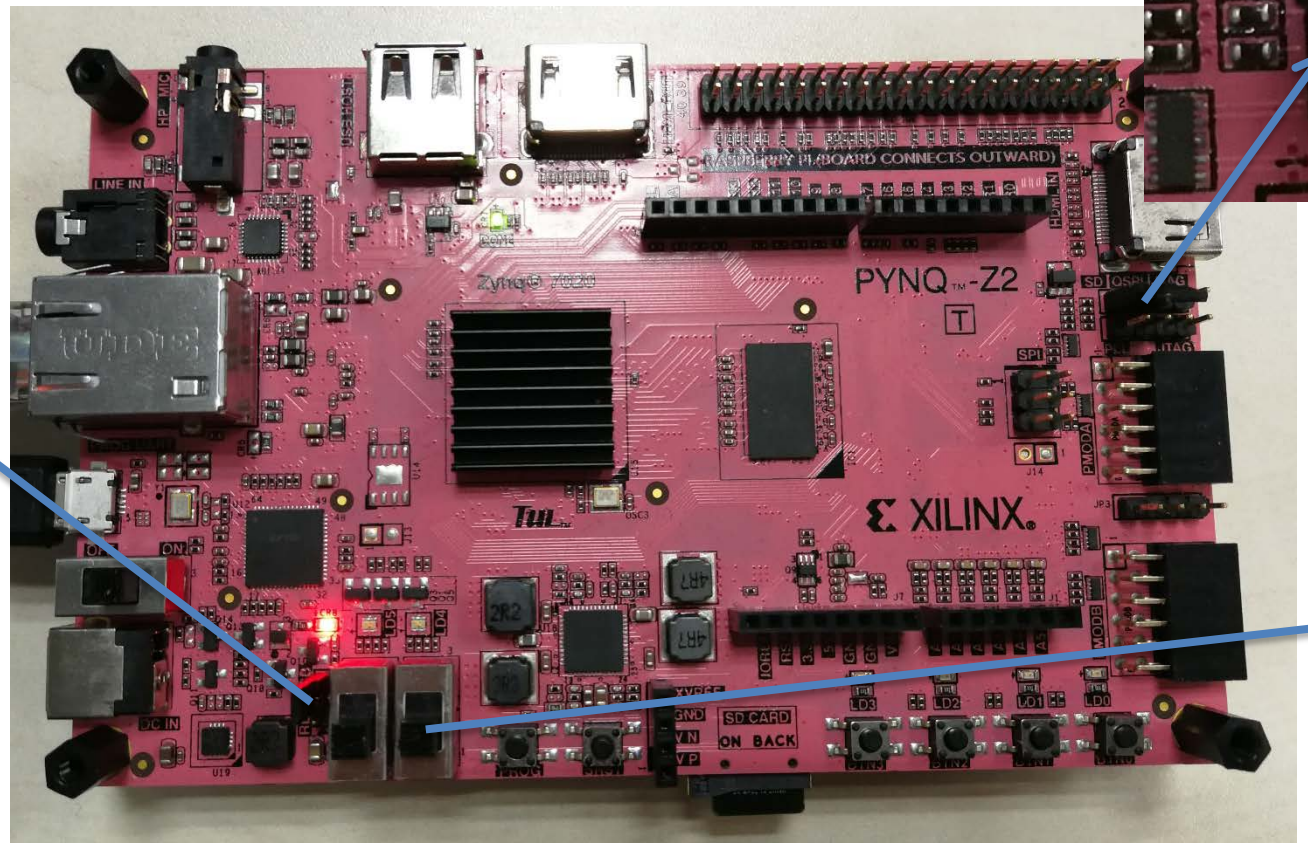
GitHub
https://github.com/ict-accel-team/serve.r

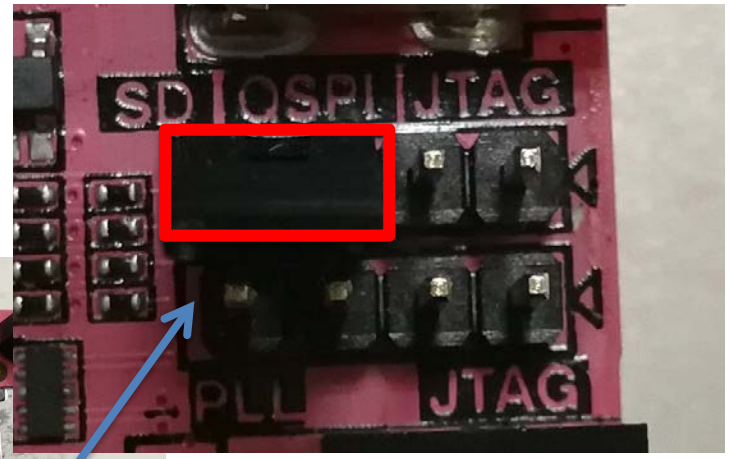
板卡设置

- 设置开发板供电方式及启动方式

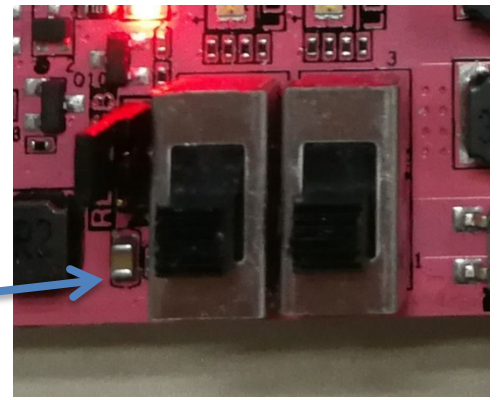


设置为USB供电
（上方两针脚）





设置为从SD卡启动
（上排左方两针脚）



将两开关拨到下方位置
（后面会介绍开关用途）

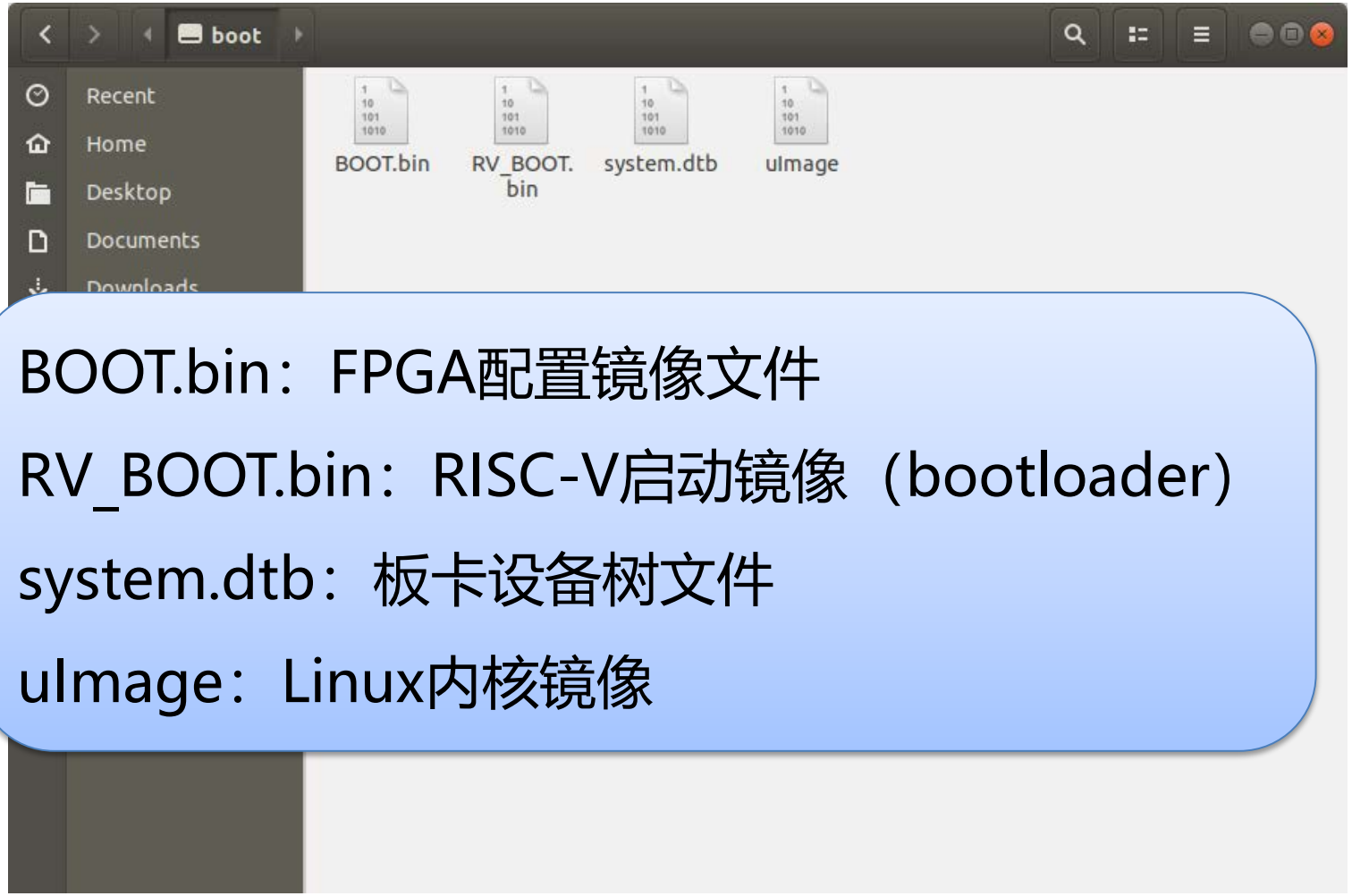
准备SD卡

- SD卡分为两个区：
  - boot分区（分区1）：FAT32文件系统，建议容量128MB以上
    - 用于存放启动镜像文件
  - root分区（分区2）：Ext3文件系统，建议容量1GB以上
    - 用于存储Debian文件系统

```
ubuntu@ubuntu-VirtualBox:~$ sudo fdisk -l /dev/sdb
Disk /dev/sdb: 7.4 GiB, 7969177600 bytes, 15564800 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x00065e7c

Device     Boot   Start      End  Sectors  Size Id Type
/dev/sdb1          2048  1050623  1048576  512M  b W95 FAT32
/dev/sdb2       1050624 15564799 14514176  6.9G 83 Linux
```

复制文件到SD卡

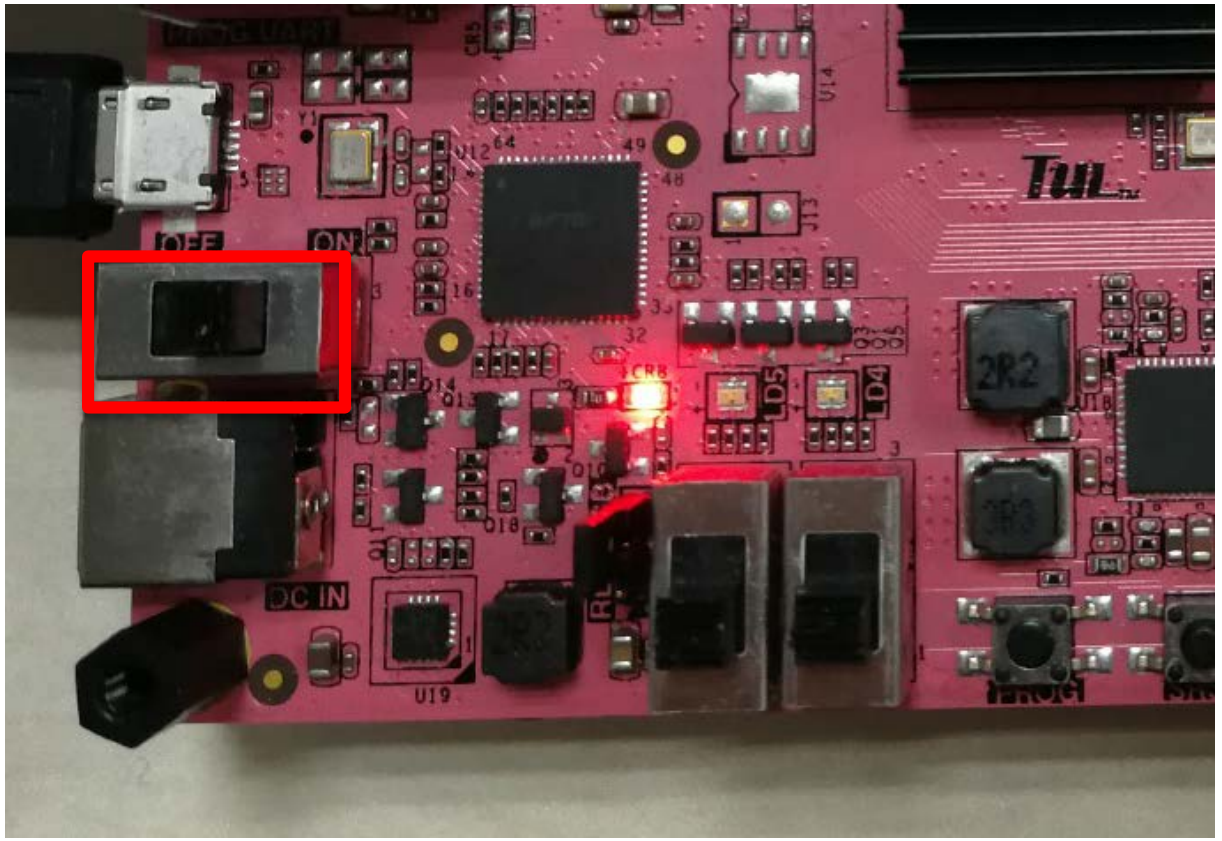
- 将SERVE.r仓库bin目录下的文件复制到boot分区根目录



连接板卡

- 将SD卡插入卡槽，板卡用USB线连接电脑
- 开启板卡电源
- 若使用虚拟机，需添加板卡对应USB设备（Xilinx TUL）

电源
开关



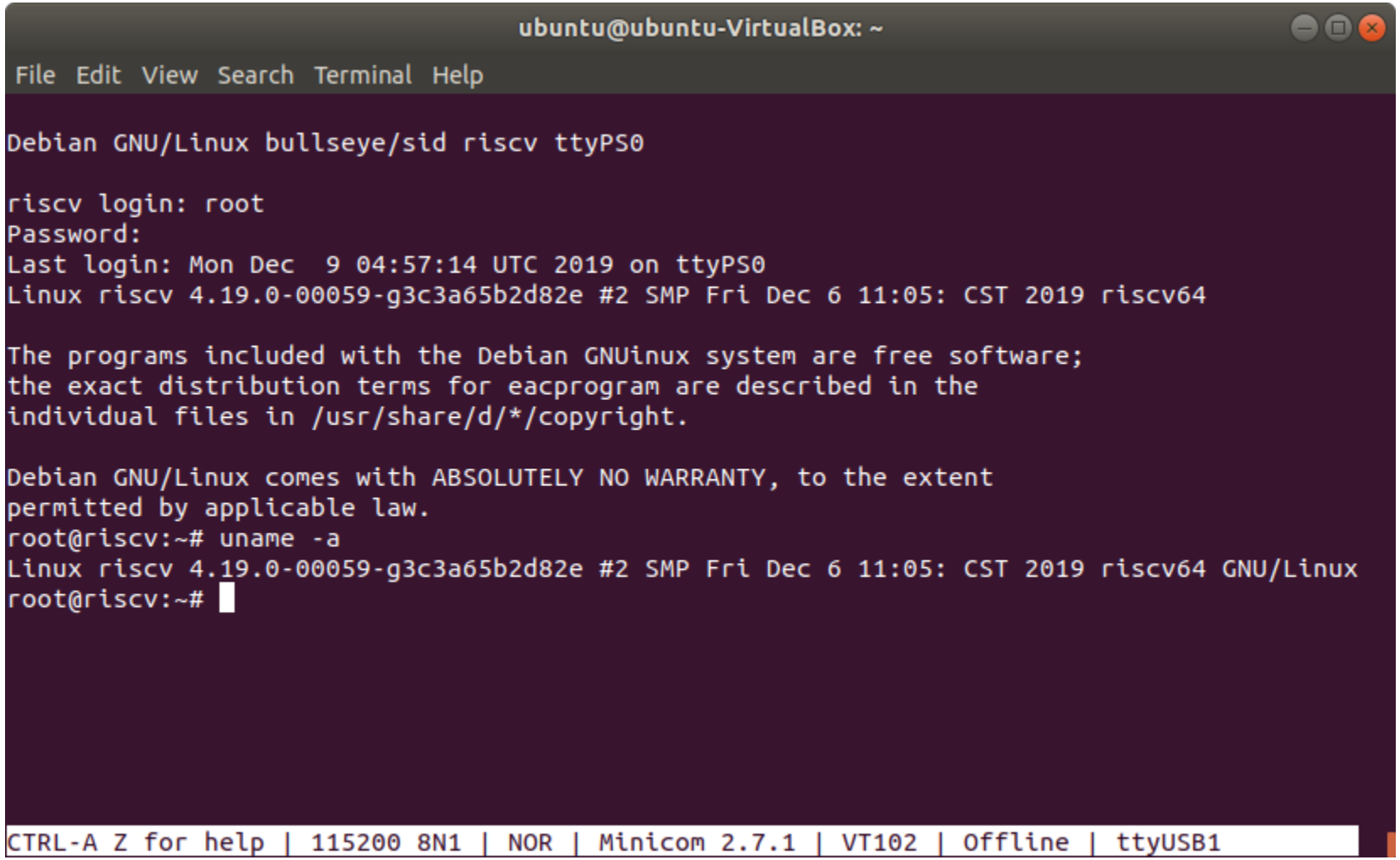
使用minicom连接串口

- 第一次使用minicom连接，使用下面的命令进入设置：
- **$ sudo minicom -s**
- 找到对应设备（通常为/dev/ttyUSBx，x为数字），设置波特率115200，无校验，无流控

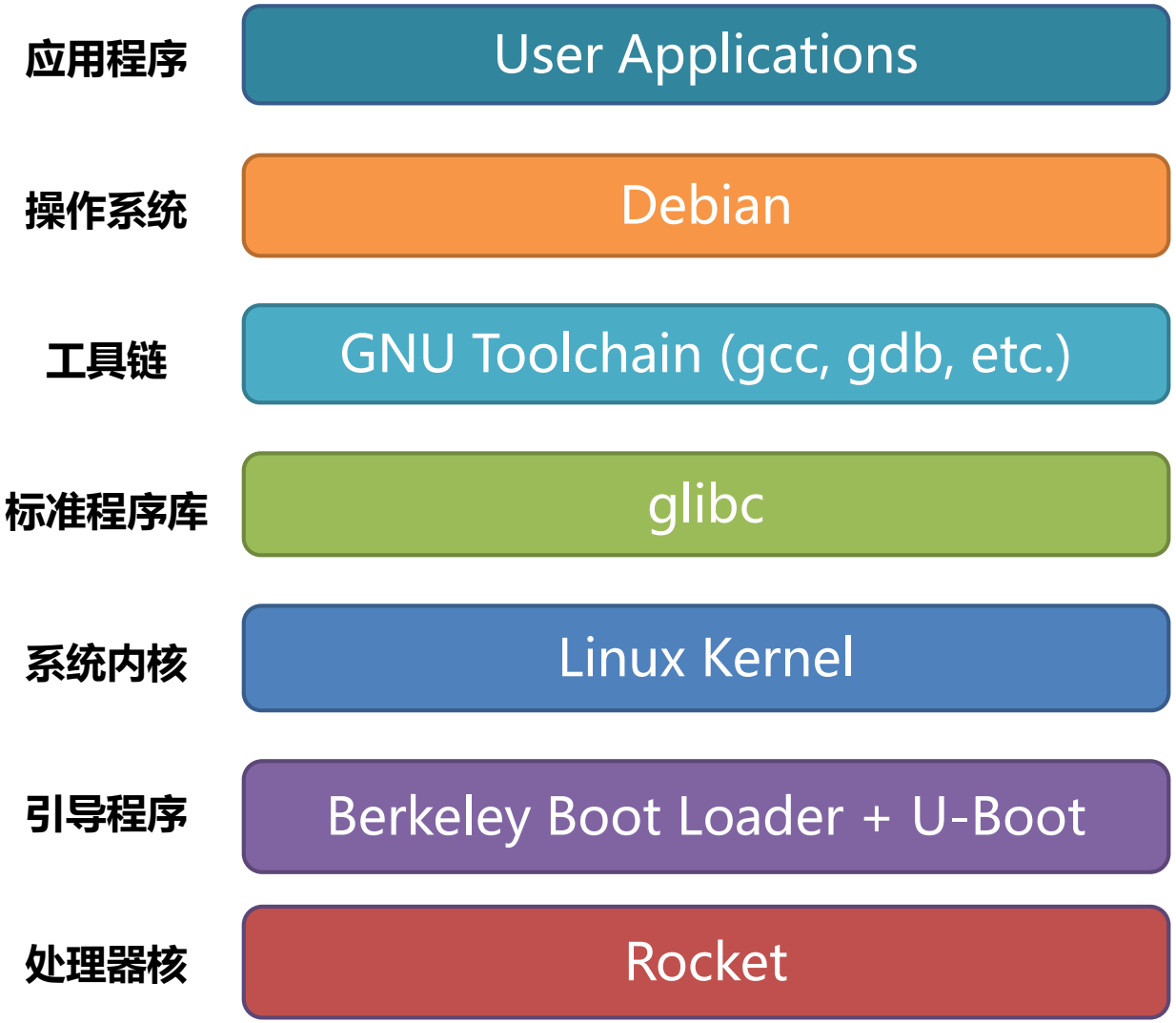
- 保存为默认配置，并退出到主界面
- 后续使用可直接运行sudo minicom

连接串口

- Linux启动后，可登录进入Shell界面（用户名和密码均为root）

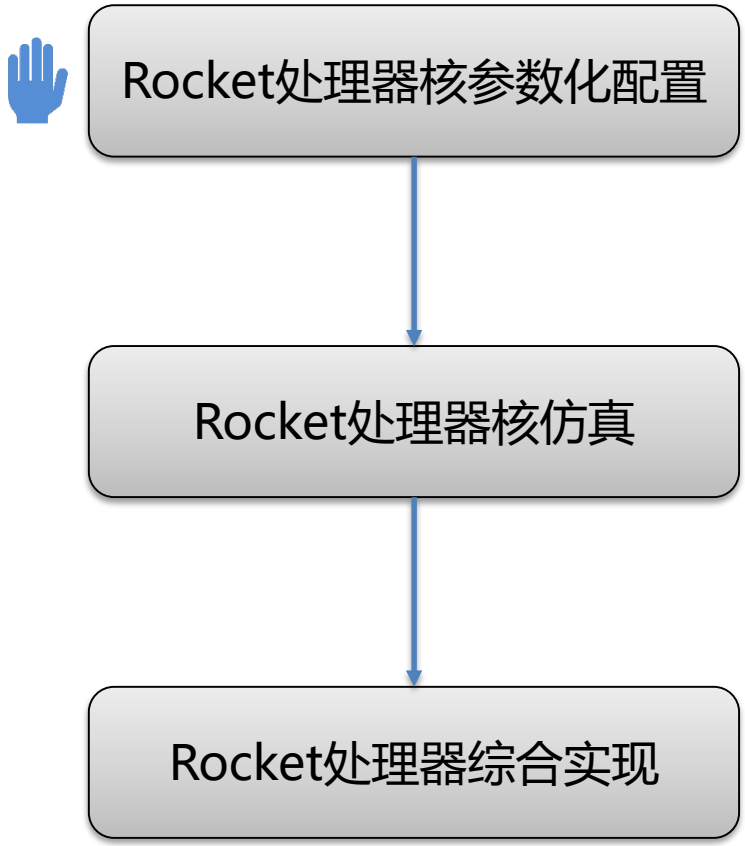


补充：SERVE.r软硬件栈环境

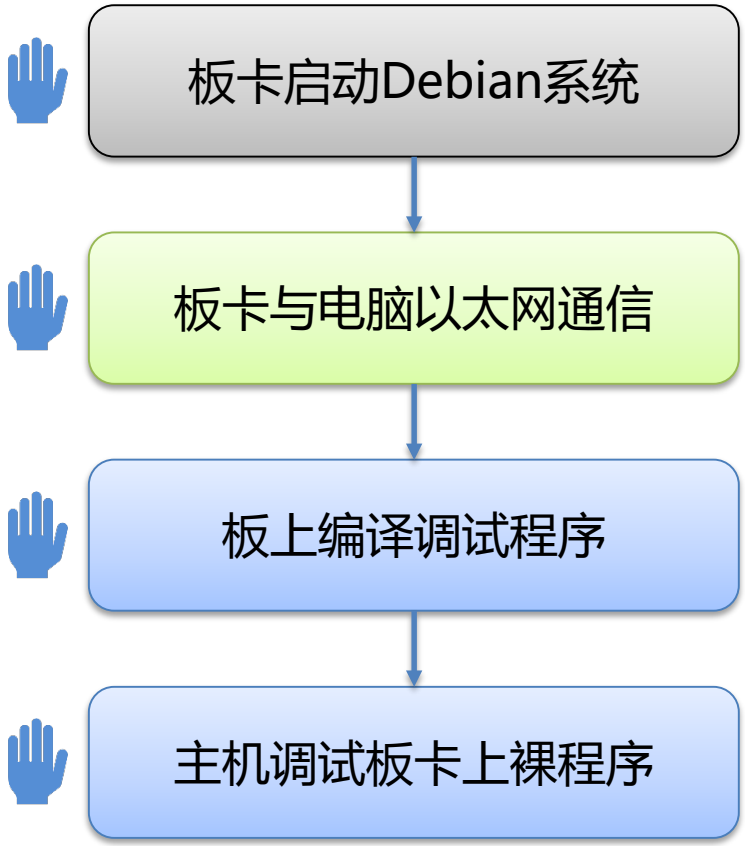


第一部分实验内容

硬件实验

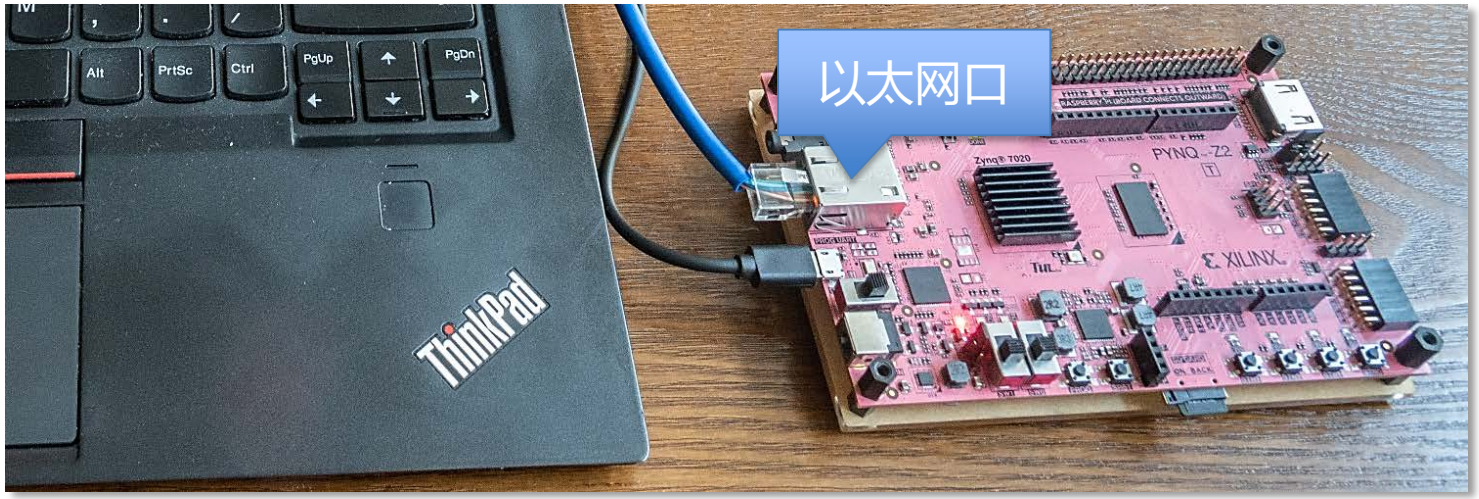


软件实验



选做实验：板卡与电脑以太网通信

- 使用网线连接板卡与电脑，用ifconfig配置IP地址并在两端相互ping通对方

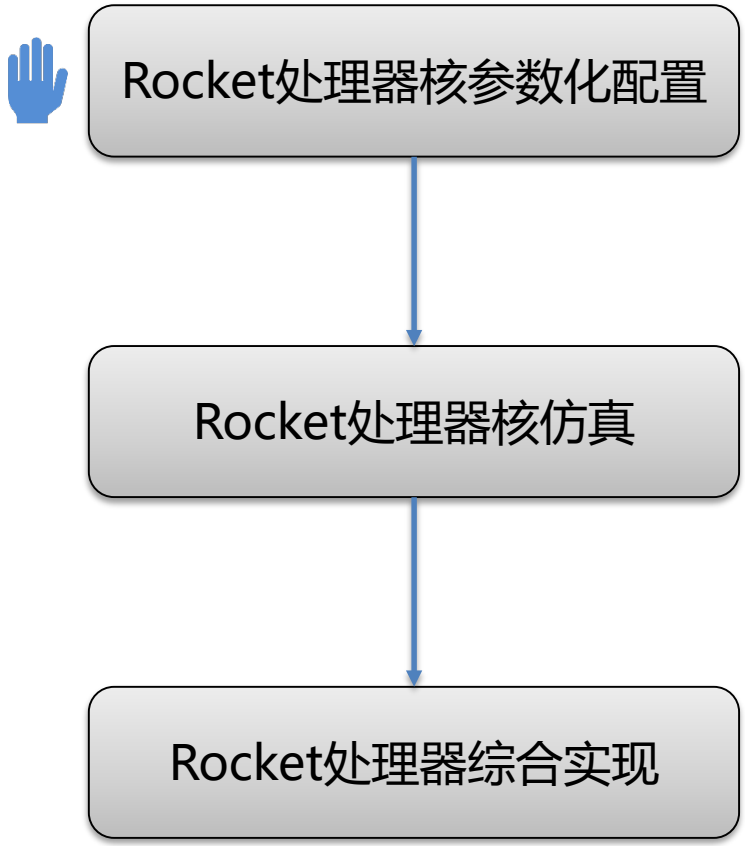



```
C:\Users\FREE>ping 192.168.100.1

正在 Ping 192.168.100.1 具有 32 字节的数据:
来自 192.168.100.1 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.100.1 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.100.1 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.100.1 的回复: 字节=32 时间<1ms TTL=64

192.168.100.1 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 0ms, 最长 = 0ms, 平均 = 0ms
```

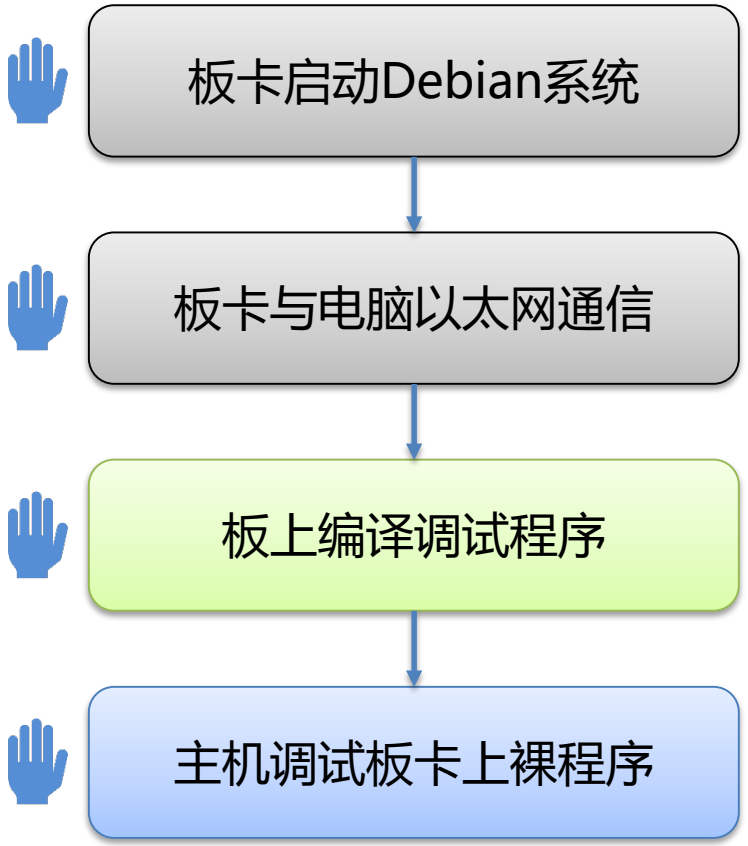
```
root@riscv:~# ifconfig eth0 192.168.100.2
root@riscv:~# ping 192.168.100.1
PING 192.168.100.1 (192.168.100.1) 56(84) bytes of data.
64 bytes from 192.168.100.1: icmp_seq=1 ttl=64 time=1.35 ms
64 bytes from 192.168.100.1: icmp_seq=2 ttl=64 time=1.17 ms
64 bytes from 192.168.100.1: icmp_seq=3 ttl=64 time=1.15 ms
64 bytes from 192.168.100.1: icmp_seq=4 ttl=64 time=1.16 ms
64 bytes from 192.168.100.1: icmp_seq=5 ttl=64 time=1.13 ms
64 bytes from 192.168.100.1: icmp_seq=6 ttl=64 time=1.08 ms
```

第一部分实验内容

硬件实验

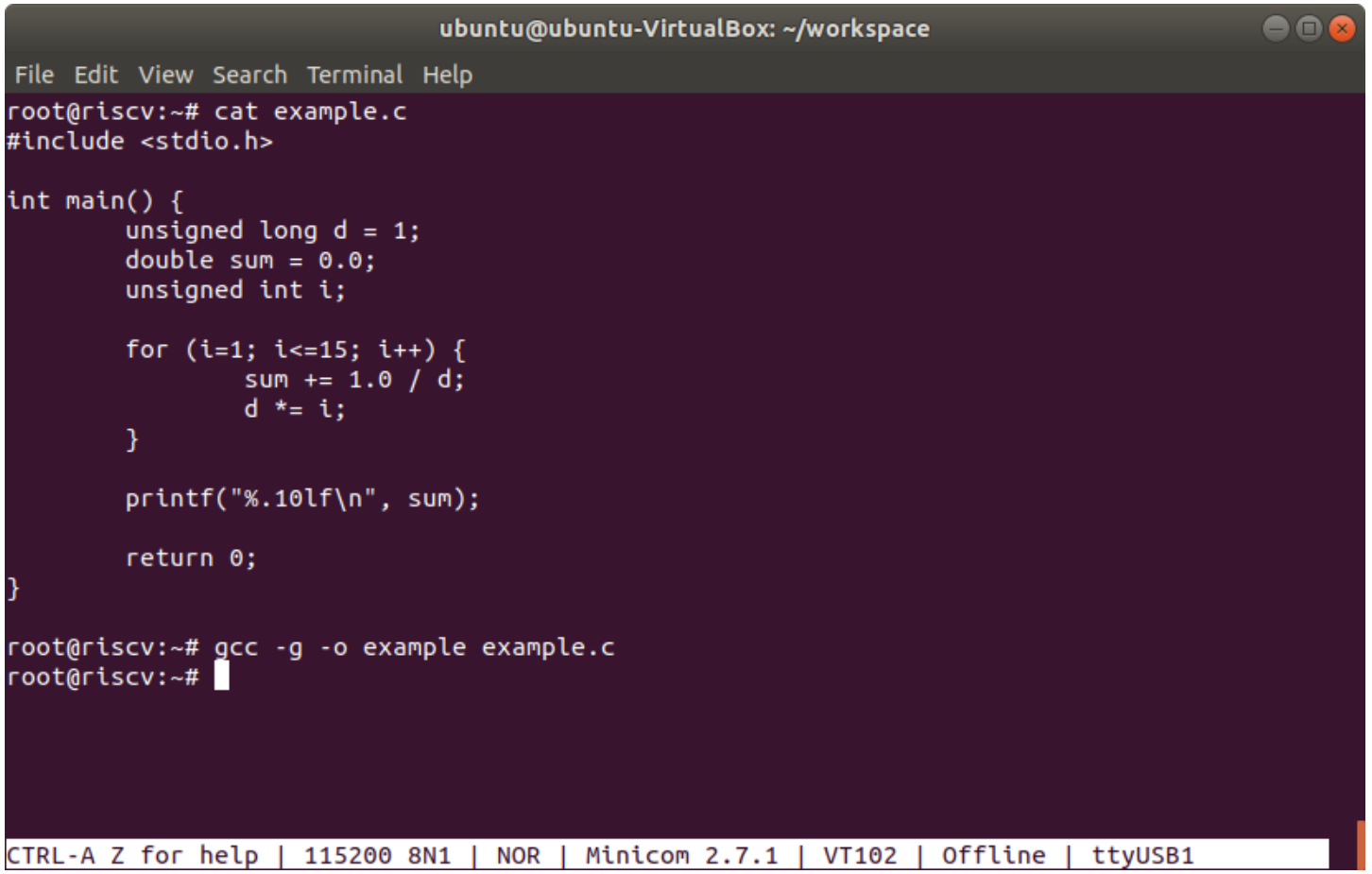


软件实验



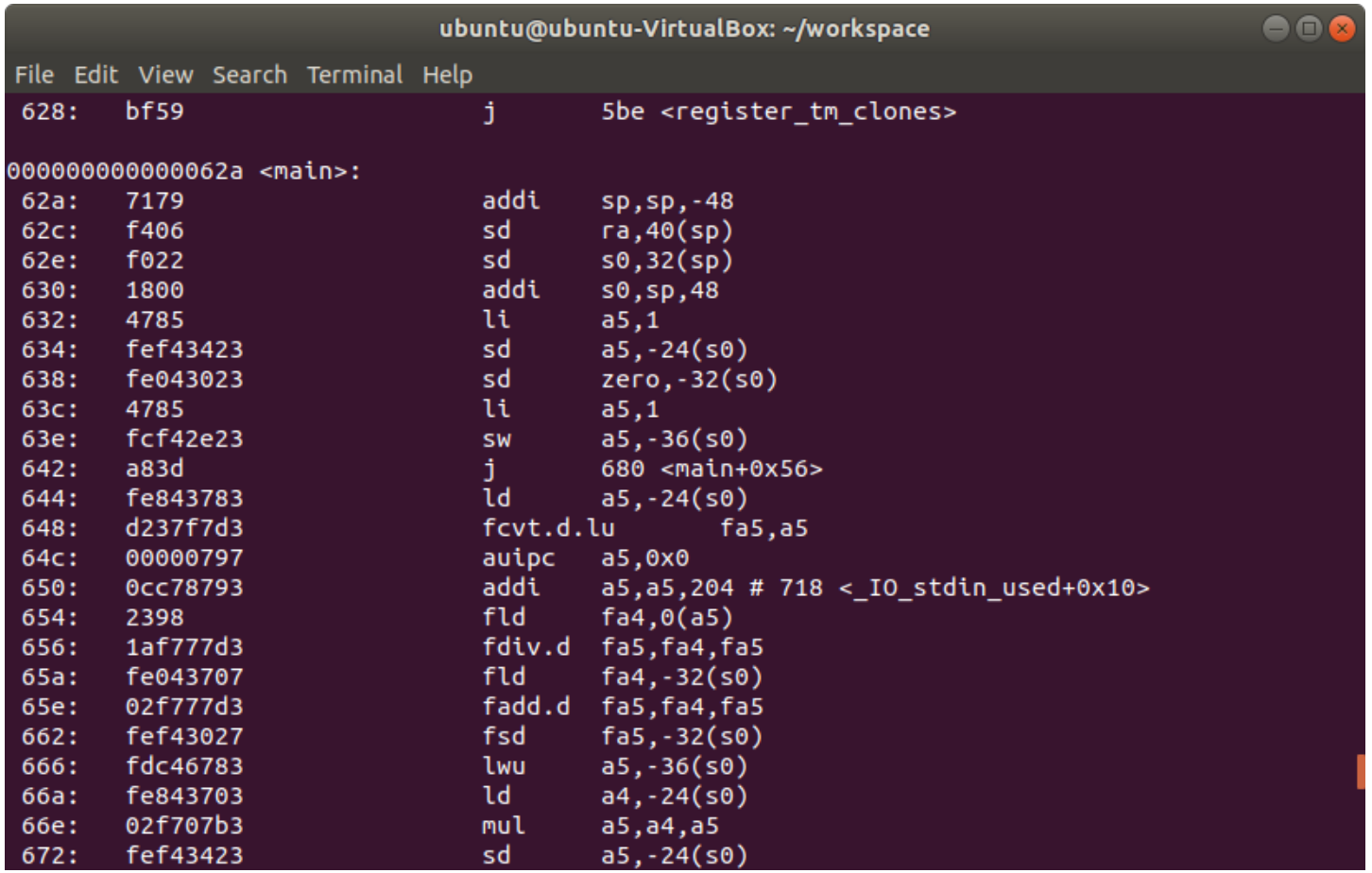
实验：板上编译调试程序 - 编译程序

- 开发板上使用GCC编译应用程序（/root目录下example.c文件）
- **$ gcc -g -o example example.c**



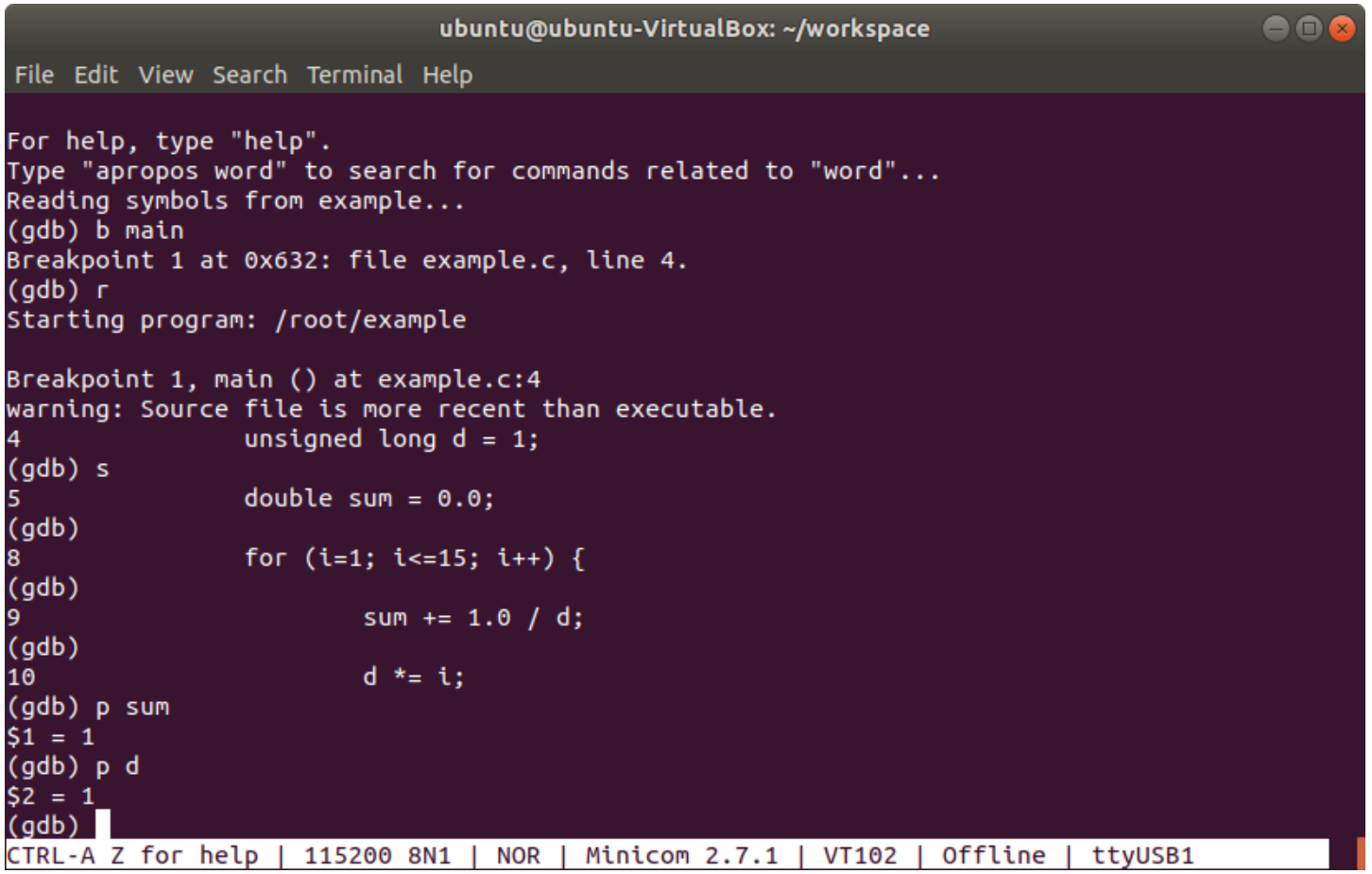
实验：板上编译调试程序 - 查看反汇编

- 使用objdump查看反汇编
- **`$ objdump -d example`**



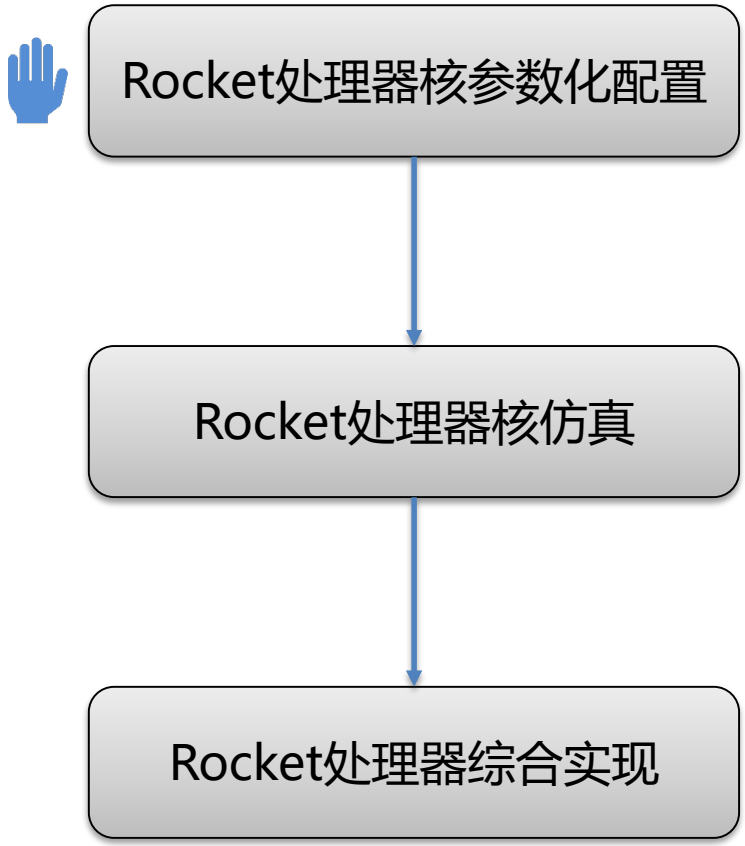
实验：板上编译调试程序 - 调试程序

- 使用GDB在板卡上调试应用程序
- **$ gdb example**

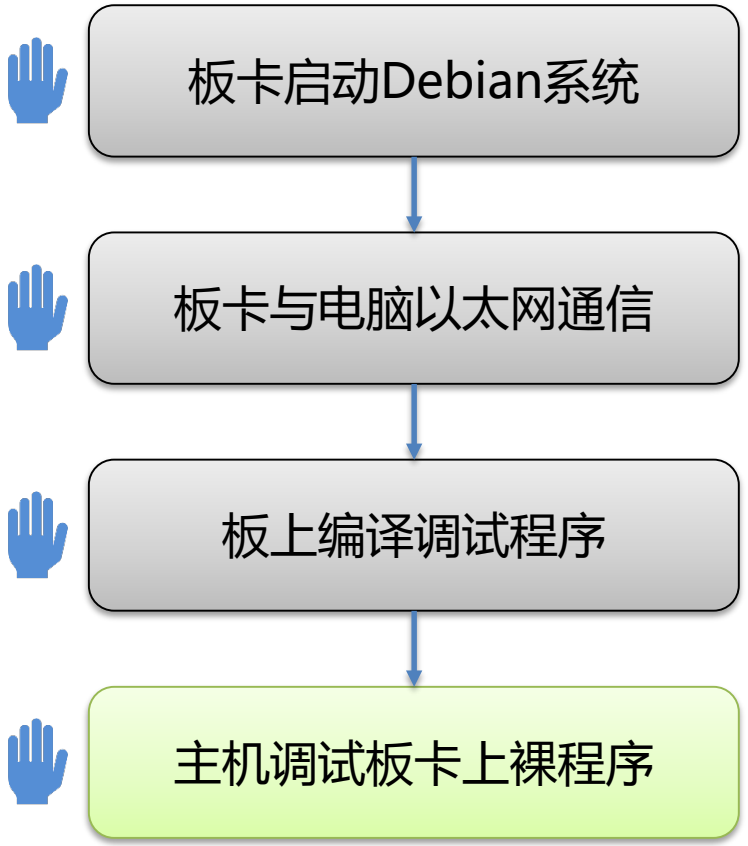


第一部分实验内容

硬件实验



软件实验

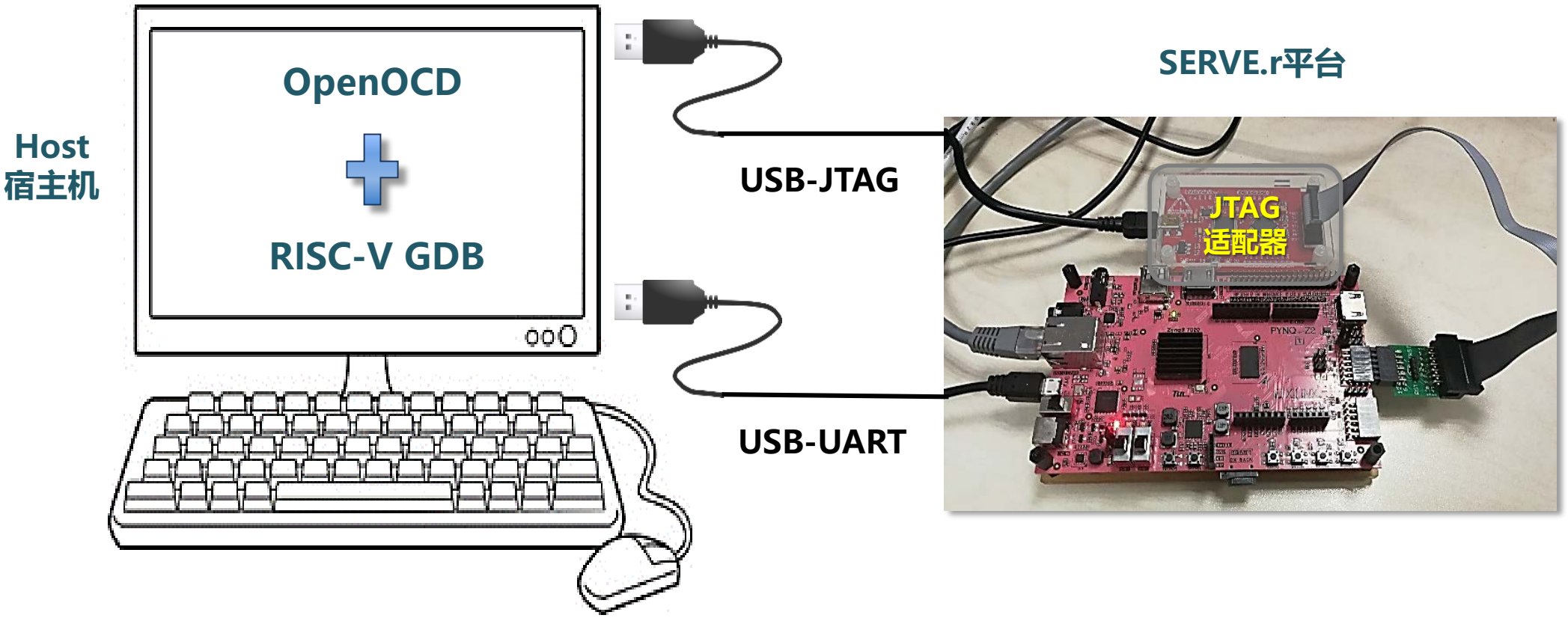


不同层次应用程序调试手段对比

程序	手段	机制
操作系统之上的 应用程序	操作系统上的调试器（GDB等）	由操作系统提供 (Linux的ptrace等)
直接运行于硬件上的 程序（裸程序）	硬件调试手段（JTAG方式等）	硬件本身提供 (调试模块)

JTAG：一种用于芯片测试及对系统进行仿真、调试的协议

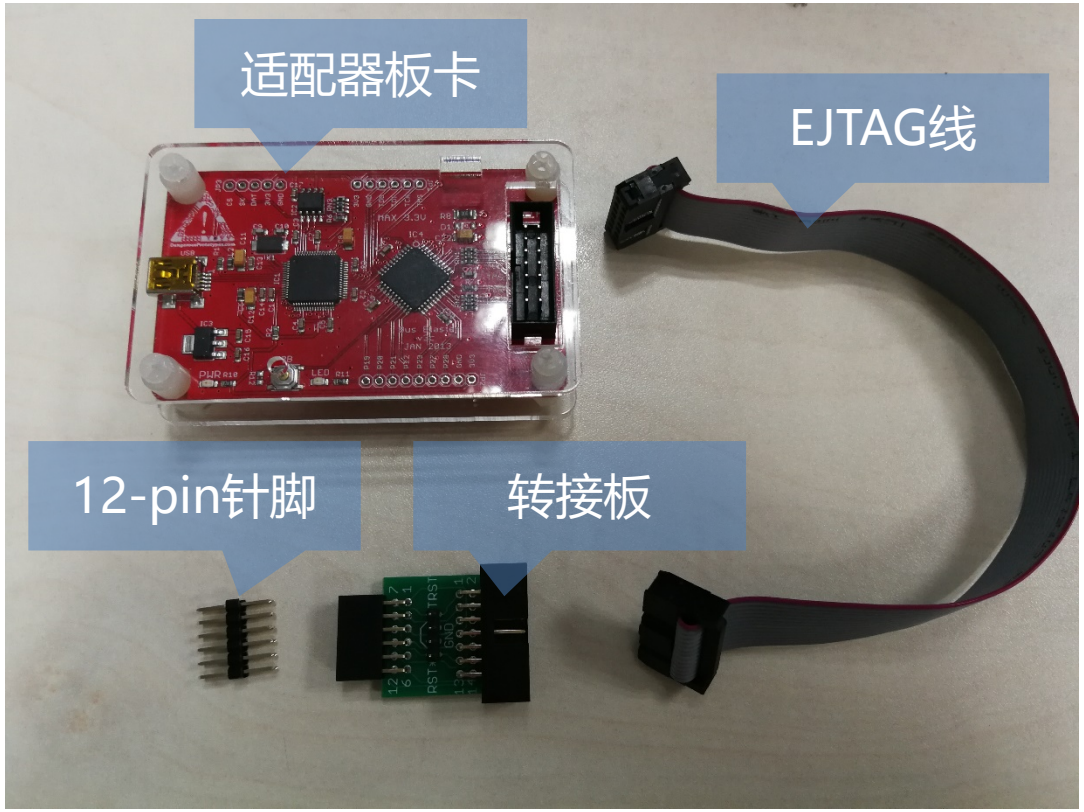
JTAG调试环境



OpenOCD：支持JTAG的芯片调试器，可作为GDB调试的中间工具

实验：主机调试板卡上裸程序 - 连接适配器

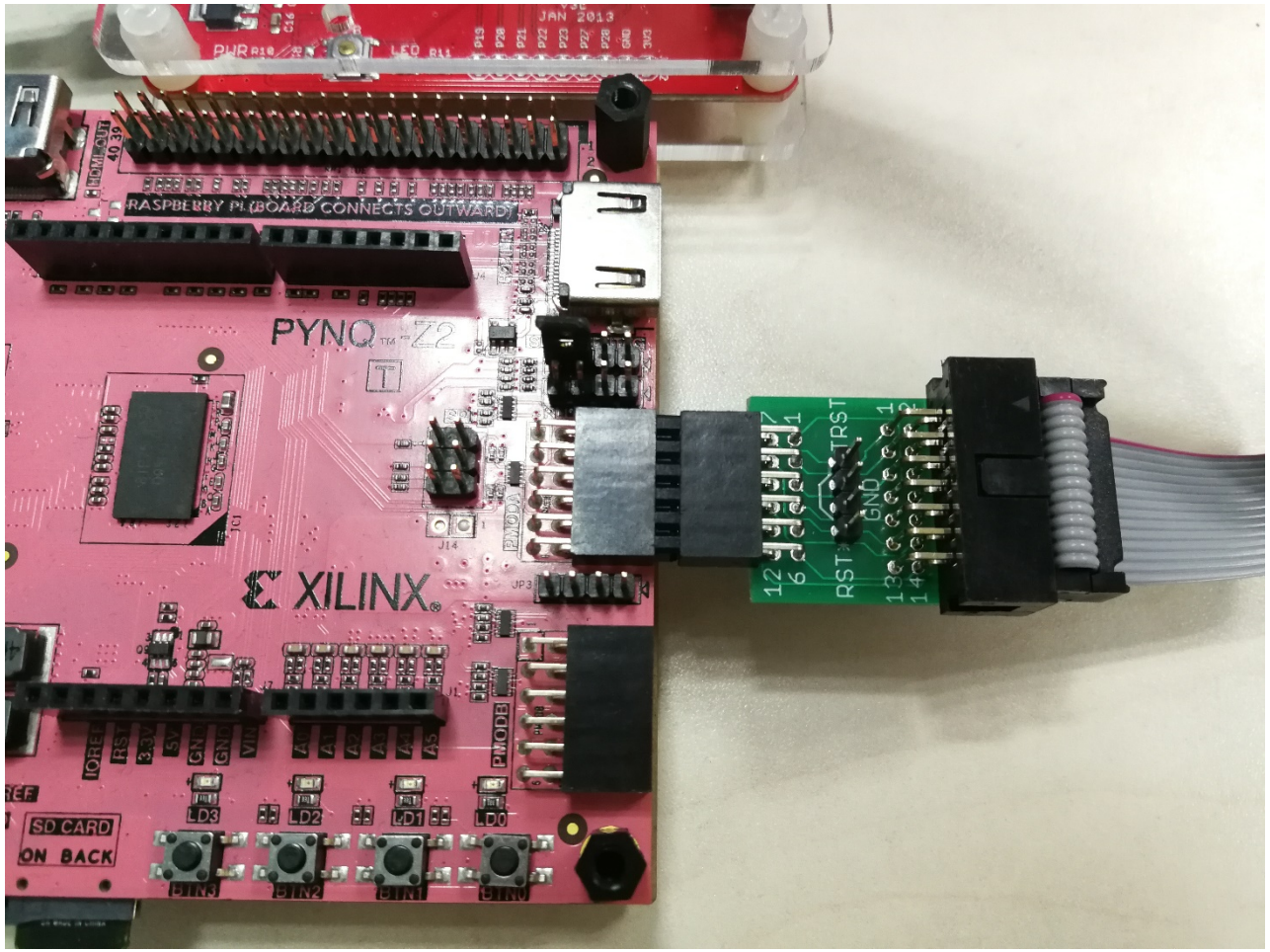
- JTAG适配器套件包含USB连接线、适配器板卡、EJTAG线、转接板、12-pin针脚



- 将图上的部件依次连接，EJTAG线按缺口对应方向连接

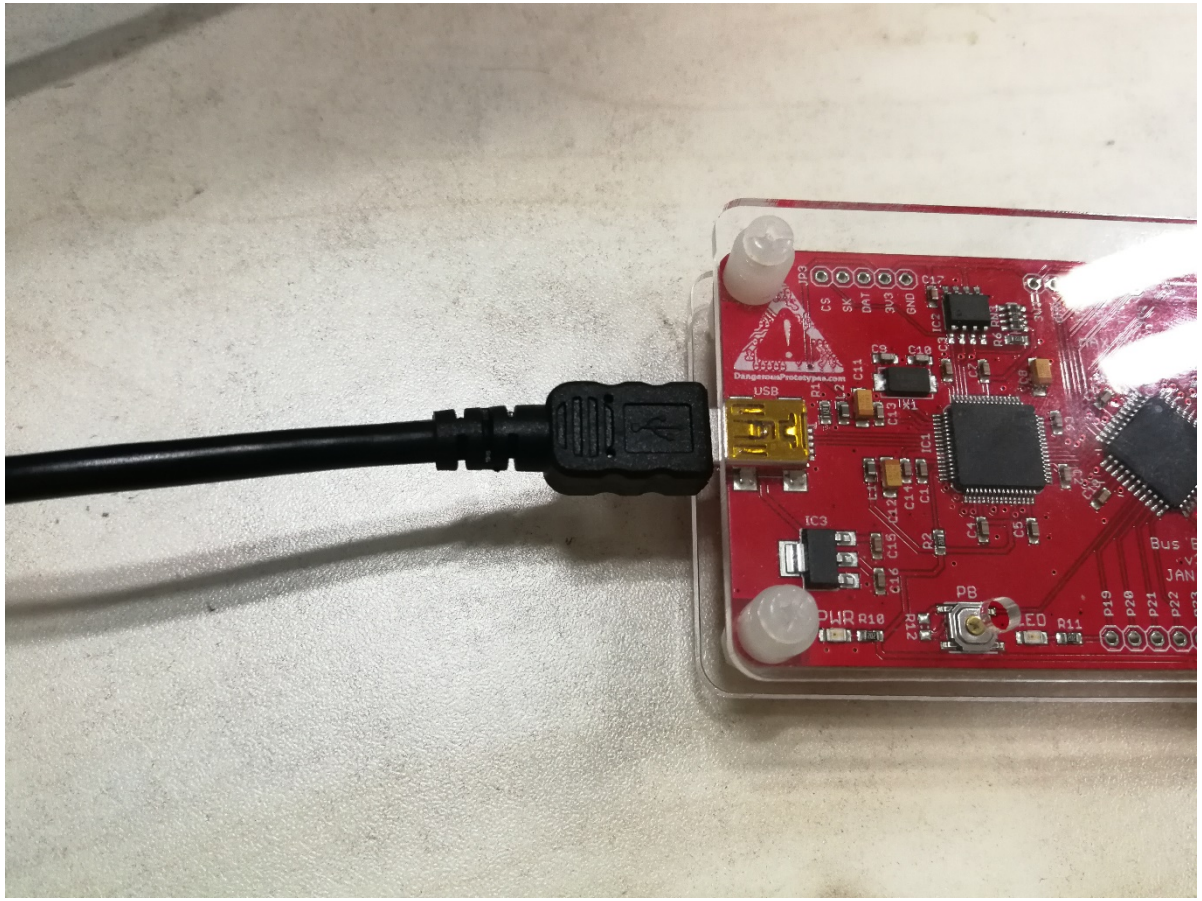
实验：主机调试板卡上裸程序 - 连接适配器

- 按如图所示方向将12-pin针脚端插入板卡Pmod A接口
- 插入后转接板正面朝上，此时1-12针脚相对应



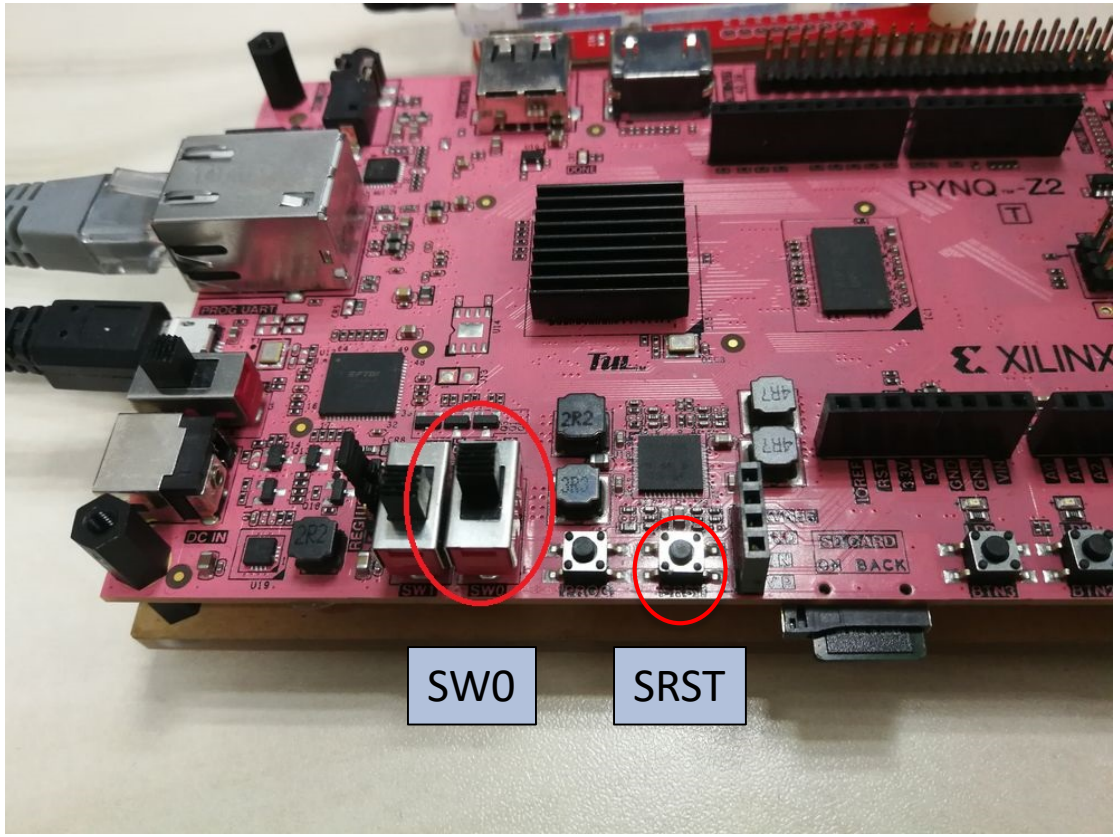
实验：主机调试板卡上裸程序 - 连接适配器

- 将USB线与适配器板卡连接
- **USB线先不要连接电脑，以免占用板卡USB串口对应的设备号**



实验：主机调试板卡上裸程序 - 调试模式

- 将板卡上的SW0开关拨到如图所示向上的位置（进入调试模式）
- 开启电源开关（或电源开启状态下按SRST键）
- 若在虚拟机内连接串口，重新开启电源后需重新添加串口设备



实验：主机调试板卡上裸程序 - OpenOCD

- 连接板卡串口，可以看到如下输出：（若看不到输出可重新开启电源）

```
RISC-V Boot Environment Setup...
Successfully load RISC-V boot file into DRAM @ 0x10000000
Paing system contrl to RISC-V core...
Execution Paused
```

- 确认串口正确连接后，将JTAG适配器USB接口与电脑连接（虚拟机同样需添加设备）

- 在拷贝的openocd文件夹中运行下面的脚本启动OpenOCD：
- **$ sh run_openocd.sh**

- OpenOCD成功通过JTAG连接板卡后，输出内容如右图

```
ubuntu@ubuntu-VirtualBox:~/share/baremetal$ /opt/riscv-serve/run_openocd.sh
Open On-Chip Debugger 0.10.0+dev-g00b543b5f (2019-12-05-14:53)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : If you need SWD support, flash KT-Link buffer from https://github.com/bha
rrisau/busblaster
and use dp_busblaster_kt-link.cfg instead
Info : auto-selecting first available session transport "jtag". To override use
'transport select <transport>'.
Info : ftdi: if you experience problems at higher adapter clocks, try the comman
d "ftdi_tdo_sample_edge falling"
Info : clock speed 10000 kHz
Info : JTAG tap: riscv.cpu tap/device found: 0x20000901 (mfg: 0x480 (<invalid>),
 part: 0x0000, ver: 0x2)
Info : datacount=2 progbufsize=16
Info : Disabling abstract command reads from CSRs.
Info : Examined RISC-V core; found 1 harts
Info :  hart 0: XLEN=64, misa=0x800000000014112d
Info : Listening on port 3333 for gdb connections
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

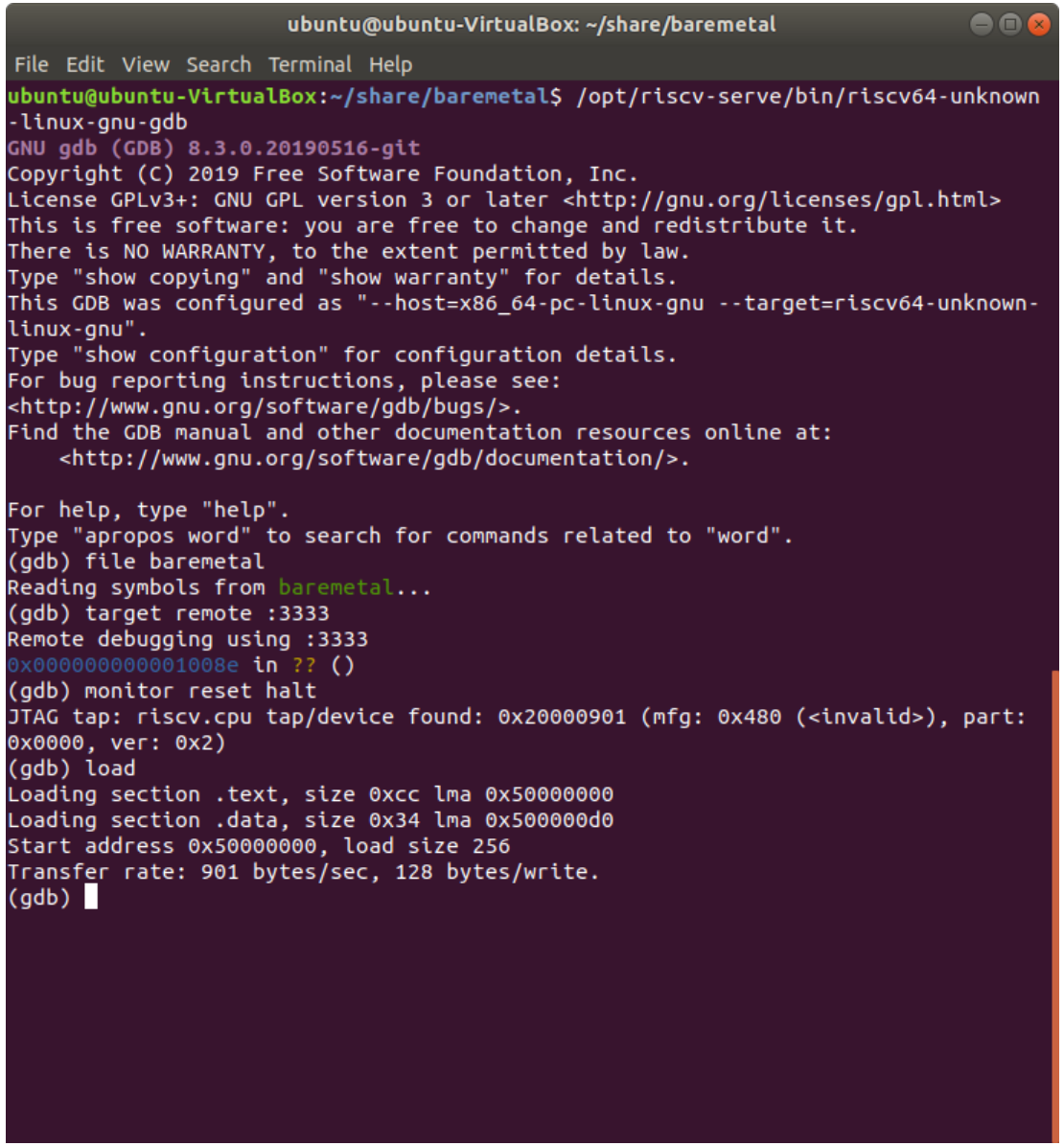
实验：主机调试板卡上裸程序 - GDB调试

- 开启新的终端窗口，在baremetal目录下启动RISC-V GDB：
- **$ /opt/riscv-serve/bin/riscv64-unknown-linux-gnu-gdb**

- GDB中执行以下命令：
- **file baremetal**（GDB加载ELF文件）
- **target remote :3333**（连接OpenOCD）
- **monitor reset halt**（CPU复位）
- **load**（加载程序到板卡内存）

- 此时ELF加载完毕且PC处于程序入口，后续与Linux下GDB调试方法一致

- 若需重新运行程序，请在GDB中再次执行CPU复位和加载操作



实验：主机调试板卡上裸程序 - GDB调试

- 调试过程中程序在串口输出文本：

```
RISC-V Boot Environment Setup...
Successfully load RISC-V boot file into DRAM @ 0x10000000
Paing system contrl to RISC-V core...
Execution Paused



**************
 Hello World!
**************

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT1
```

```
(gdb) file baremetal
Reading symbols from baremetal...
(gdb) target remote :3333
Remote debugging using :3333
0x000000000001008e in ?? ()
(gdb) monitor reset halt
JTAG tap: riscv.cpu tap/device found: 0x20000901 (mfg: 0x480 (<invalid>), part:
0x0000, ver: 0x2)
(gdb) load
Loading section .text, size 0xcc lma 0x50000000
Loading section .data, size 0x34 lma 0x500000d0
Start address 0x50000000, load size 256
Transfer rate: 901 bytes/sec, 128 bytes/write.
(gdb) b main
Breakpoint 1 at 0x50000080: file baremetal.c, line 24.
(gdb) c
Continuing.

Breakpoint 1, main () at baremetal.c:24
24          init_uart();
(gdb) n
27          for(i = 0;array[i];i ++)
(gdb)
28              _putc(array[i]);
(gdb)
27          for(i = 0;array[i];i ++)
(gdb) p i
$1 = 0
(gdb) p array
$2 = "\r\n\n\n", '*' <repeats 14 times>, "\r\n Hello World!\r\n", '*' <repeats 1
4 times>, "\r\n"
(gdb)
```

提 纲

1 开源芯片的缘起

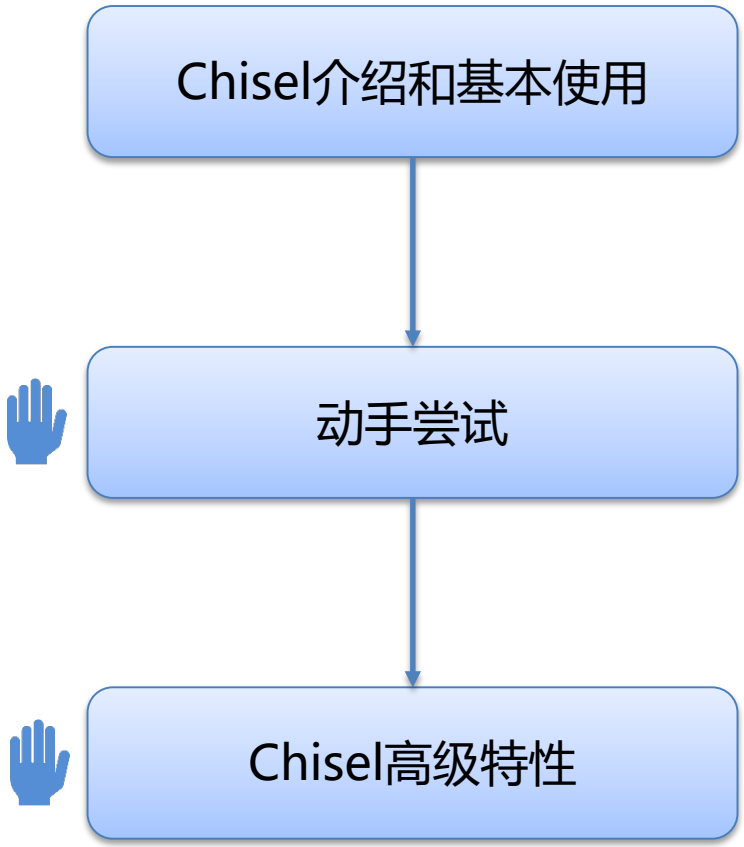
2 RISC-V开源芯片

3 Chisel硬件语言

4 敏捷开发的愿景

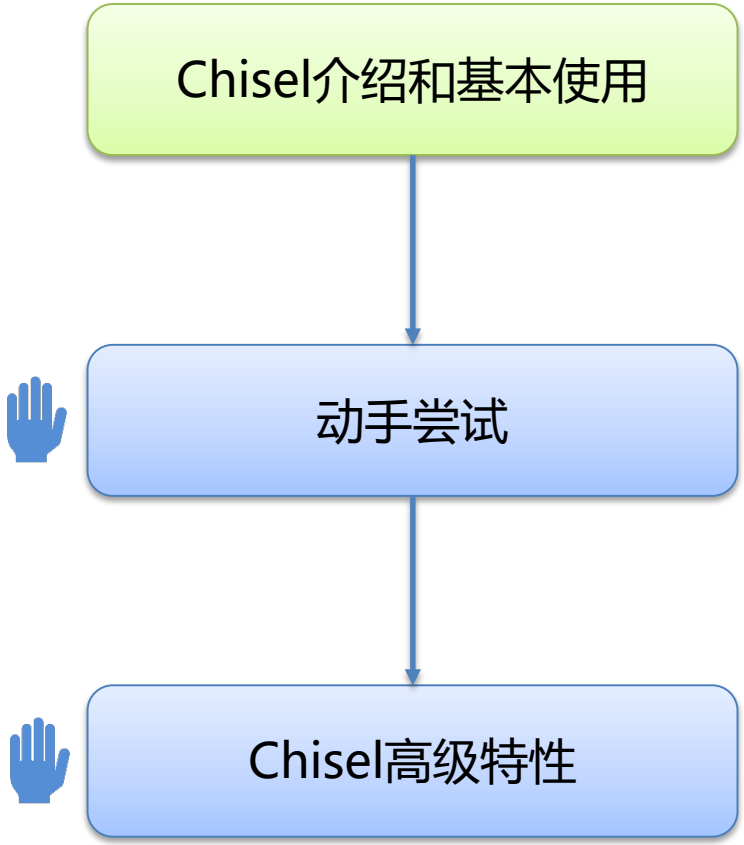
第二部分实验内容

Chisel - 面向敏捷开发的硬件设计语言



第二部分实验内容

Chisel - 面向敏捷开发的硬件设计语言

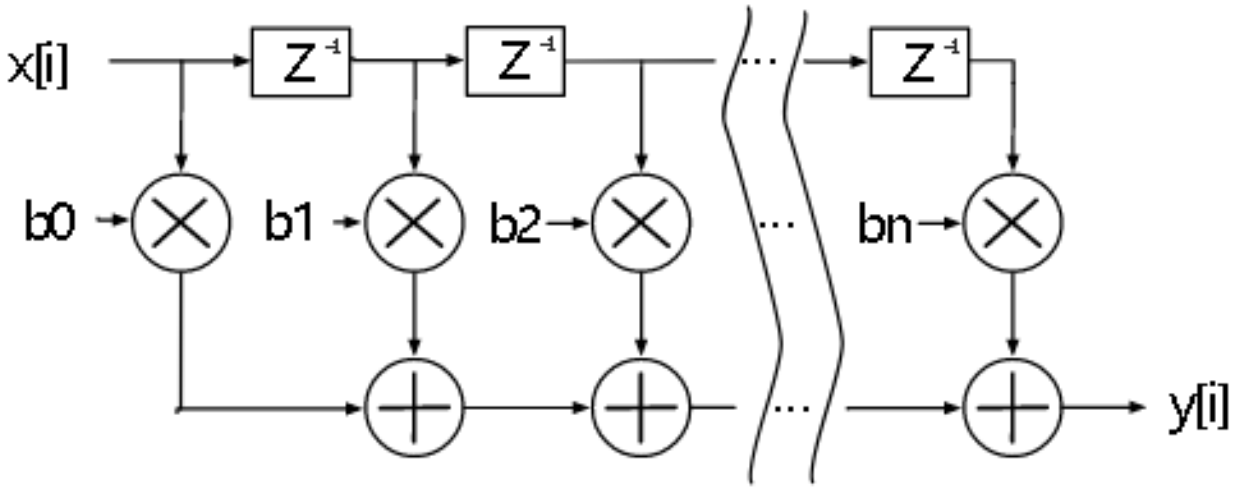


Chisel – 面向敏捷开发的硬件设计语言

- **宗旨 – 多快好省地描述电路**
  - 减少重复代码, 提升开发效率, 提升代码可读性和易维护性
- Chisel不是HLS(高层次综合)
  - Chisel仍然在描述电路, 只不过描述的方式更方便
  - HLS描述算法, 再把算法映射成电路
- Chisel为什么能敏捷?
  - 今天为大家揭开Chisel神秘的面纱

一个简单的例子 – FIR滤波器

- 例子出自chisel-bootcamp
- n = 2, b0 = b1 = b2 = 1



- 没注释也不难理解

```
class MovingAverage3(bitWidth: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))    // 宽度为bitWidth的无符号数输入
    val out = Output(UInt(bitWidth.W))
  })

  val z1 = RegNext(io.in) // 打一拍
  val z2 = RegNext(z1)

  io.out := (io.in * 1.U) + (z1 * 1.U) + (z2 * 1.U)  // Chisel常数
}
```

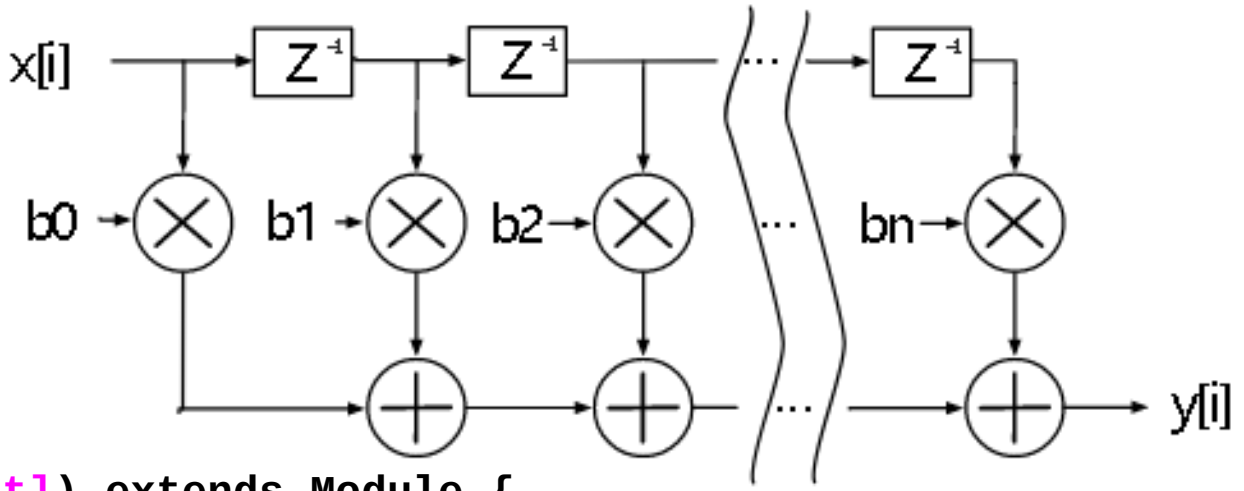
Verilog vs Chisel = 实例 vs 生成器

- 和Verilog差不多啊, Chisel有什么好处?
- 看着差不多是因为, 这个例子在编写一个实例(Instance)
  - 实例 = 一个具体的电路
- Chisel的好处体现在编写生成器(Generator)
  - 生成器 = 用来生成实例的模块

```
class MovingAverage3(bitWidth: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))    // 宽度为bitWidth的无符号数输入
    val out = Output(UInt(bitWidth.W))
  })

  val z1 = RegNext(io.in) // 打一拍
  val z2 = RegNext(z1)

  io.out := (io.in * 1.U) + (z1 * 1.U) + (z2 * 1.U)  // chisel常数
}
```

通用的FIR滤波器生成器

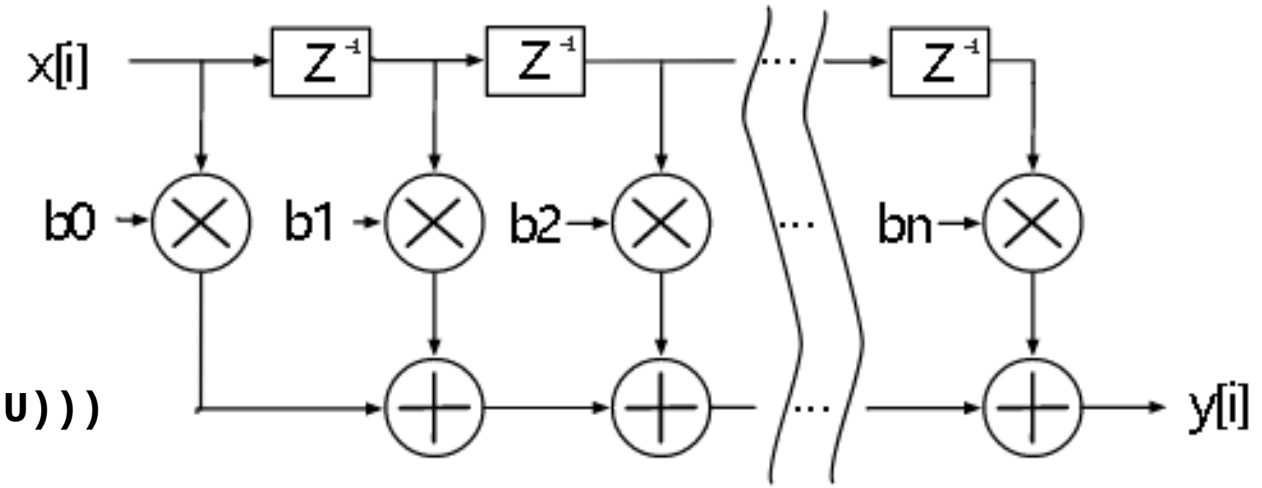
- 系数b的数值和个数均可配




```
class FirFilter(bitWidth: Int, coeffs: Seq[UInt]) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))
    val out = Output(UInt())
  })

  val zs = Reg(Vec(coeffs.length, UInt(bitWidth.W))) // coeffs.length个宽度为bitWidth的寄存器
  zs(0) := io.in
  for (i <- 1 until coeffs.length) { zs(i) := zs(i-1) }

  val products = VecInit.tabulate(coeffs.length)(i => zs(i) * coeffs(i)) // 变量和各自的系数相乘

  io.out := products.reduce(_ +& _) // 用加法进行归约
}
```

生成器 -> 实例

```
// 相同的MovingAverage3
new FirFilter(8, Seq(1.U, 1.U, 1.U)))

// 1周期延迟器
new FirFilter(8, Seq(0.U, 1.U))

// 三角脉冲响应
new FirFilter(8, Seq(1.U, 2.U, 3.U, 2.U, 1.U)))
```



- 若没有生成器, 只能编写多个模块实现各自的功能
  - 编写繁琐, 阅读困难, 维护不易, 验证重复
- **观念的改变: 优先编写和维护生成器, 而非实例**
  - 用Chisel编写实例, 并不会比Verilog方便多少

糟糕的编程习惯 – Copy-Paste

- 大家有没有这样的经历?
  - 写了一份代码(函数, 模块)
  - 后来发现又要写几份差不多的, 于是把第一份代码拷过来改改
  - 过了一段时间, 发现其中一份有bug, 赶紧改了
  - 又过了一段时间, 又调了一个bug, 但好像之前在哪里调过?
    - 你束手无策, 已经分辨不出哪些代码是什么时候从哪个地方拷过来的了

- 编写多个相似的实例 = 糟糕的Copy-Paste编程
  - 所以编写生成器是维护代码的一种正确做法
- 如果大家用Verilog进行Copy-Paste编程, 不要自责
  - 因为用Verilog编写生成器, 太困难了

编写生成器

- 我们希望生成器可以
  - 组合容易 – 小生成器构成大生成器(RocketChip)
  - 功能强大 – 有能力抽象出实例之间更多共同的电路特征
    - 不是算法特征
  - 控制精确 – 细粒度地控制实例的生成

- Scala灵活强大的语言特性可以满足这些需求
  - 所以Chisel开发团队选择Scala作为Chisel的宿主语言

- **但学习如何编写生成器并不容易，需要用抽象思维多多思考**

Chisel和Scala的关系

- Chisel是构建在Scala上的一种DSL(领域专用语言)

Scala vs Chisel	C语言 vs 计算器(看成一种DSL)
Chisel类型/对象是特定的Scala类型/对象，但反过来不是	计算器类型/变量是C类型/变量，但反过来不是（如计算器没有指针）
Chisel对象存放了电路元素的信息	计算器变量存放了表达式的信息
编译运行构建在Scala上的Chisel代码，生成电路实例(最后表现为Verilog代码)	编译运行构建在C语言上的计算器，进行计算
电路元素不能访问Scala对象	表达式不能访问C变量

Chisel基本使用

- 模块定义/实例化
- 组合电路
  - 基本数据类型
  - 运算符
- 时序电路
  - 寄存器, Memory
- 控制流

Chisel3 Cheat Sheet

Version 0.5 (beta): September 5, 2019

Notation In This Document:
For Functions and Constructors:
Arguments given as `kwd:type` (name and type(s))
Arguments in brackets (`[...]`) are optional.
For Operators:
`c`, `x`, `y` are Chisel `Data`; `n`, `m` are Scala `Int`
`w(x)`, `w(y)` are the widths of `x`, `y` (respectively)
`minVal(x)`, `maxVal(x)` are the minimum or maximum possible values of `x`

Basic Chisel Constructs

Chisel Wire Operators:

```
// Allocate a as wire of type UInt()
val x = Wire(UInt())
x := y  // Connect wire y to wire x
```

When executes blocks conditionally by `Bool`, and is equivalent to Verilog `if`

```
when(condition1) {
  // run if condition1 true and skip rest
} .elsewhen(condition2) {
  // run if condition2 true and skip rest
} .otherwise {
  // run if none of the above ran
}
```

Switch executes blocks conditionally by data

```
switch(x) {
  is(value1) {
    // run if x === value1
  }
  is(value2) {
    // run if x === value2
  }
}
```

Enum generates value literals for enumerations

```
val s1::s2:: ... ::sn::Nil
    = Enum(nodeType:UInt, n:Int)
```

`s1, s2, ..., sn` will be created as `nodeType` literals with distinct values

`nodeType`	type of `s1, s2, ..., sn`
`n`	element count

...nded up
...nded down
...s a power of 2

Basic Data Types

Constructors:

`Bool()`	type, boolean value
`true.B` or `false.B`	literal values
`UInt(32.W)`	type 32-bit unsigned
`UInt()`	type, width inferred
`77.U` or `"hdead".U`	unsigned literals
`1.U(16.W)`	literal with forced width
`SInt()` or `SInt(64.W)`	like UInt
`-3.S` or `"h-44".S`	signed literals
`3.S(2.W)`	signed 2-bits wide *value -1*

Bits, UInt, SInt Casts: reinterpret cast except for:

`UInt → SInt`	Zero-extend to SInt

State Elements

Registers retain state until updated

```
val my_reg = Reg(UInt(32.W))
```

Flavors

`RegInit(7.U(32.W))`	reg with initial value 7
`RegNext(next_val)`	update each clock, no init
`RegEnable(next, enable)`	update, with enable gate

Updating: assign to latch new value on next clock:

```
my_reg := next_val
```

Read-Write Memory provide addressable memories

```
val my_mem = Mem(n:Int, out:Data)
```

`out` memory element type
`n` memory depth (elements)

Using: access elements by indexing:

```
val readVal = my_mem(addr:UInt/Int)
```

for synchronous read: assign output to `Reg`

```
mu_mem(addr:UInt/Int) := y
```

Modules

Defining: subclass `Module` with elements, code:

```
class Accum(width:Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(width.W))
    val out  = Output(UInt(width.W))
  })
  val sum = Reg(UInt())
  sum := sum + io.in
  io.out := sum
}
```

Usage: access elements using dot notation:
(code inside a `Module` is always running)

```
val my_module = Module(new Accum(32))
my_module.io.in := some_data
val sum := my_module.io.out
```

Operators:

Chisel	Explanation	Width
`!x`	Logical NOT	1
`x && y`	Logical AND	1
`x || y`	Logical OR	1
`x(n)`	Extract bit, 0 is LSB	1
`x(n, m)`	Extract bitfield	`n - m + 1`
`x << y`	Dynamic left shift	`w(x) + maxVal(y)`
`x >> y`	Dynamic right shift	`w(x) - minVal(y)`
`x << n`	Static left shift	`w(x) + n`
`x >> n`	Static right shift	`w(x) - n`
`Fill(n, x)`	Replicate `x`, `n` times	`n * w(x)`
`Cat(x, y)`	Concatenate bits	`w(x) + w(y)`
`Mux(c, x, y)`	If `c`, then `x`; else `y`	`max(w(x), w(y))`
`~x`	Bitwise NOT	`w(x)`
`x & y`	Bitwise AND	`max(w(x), w(y))`
`x | y`	Bitwise OR	`max(w(x), w(y))`
`x ^ y`	Bitwise XOR	`max(w(x), w(y))`
`x === y`	Equality(triple equals)	1
`x != y`	Inequality	1
`x =/= y`	Inequality	1
`x + y`	Addition	`max(w(x),w(y))`
`x +% y`	Addition	`max(w(x),w(y))`
`x +& y`	Addition	`max(w(x),w(y))+1`
`x - y`	Subtraction	`max(w(x),w(y))`
`x -% y`	Subtraction	`max(w(x),w(y))`
`x -& y`	Subtraction	`max(w(x),w(y))+1`
`x * y`	Multiplication	`w(x)+w(y)`
`x / y`	Division	`w(x)`
`x % y`	Modulus	`bits(maxVal(y)-1)`
`x > y`	Greater than	1
`x >= y`	Greater than or equal	1
`x < y`	Less than	1
`x <= y`	Less than or equal	1
`x >> y`	Arithmetic right shift	`w(x) - minVal(y)`
`x >> n`	Arithmetic right shift	`w(x) - n`

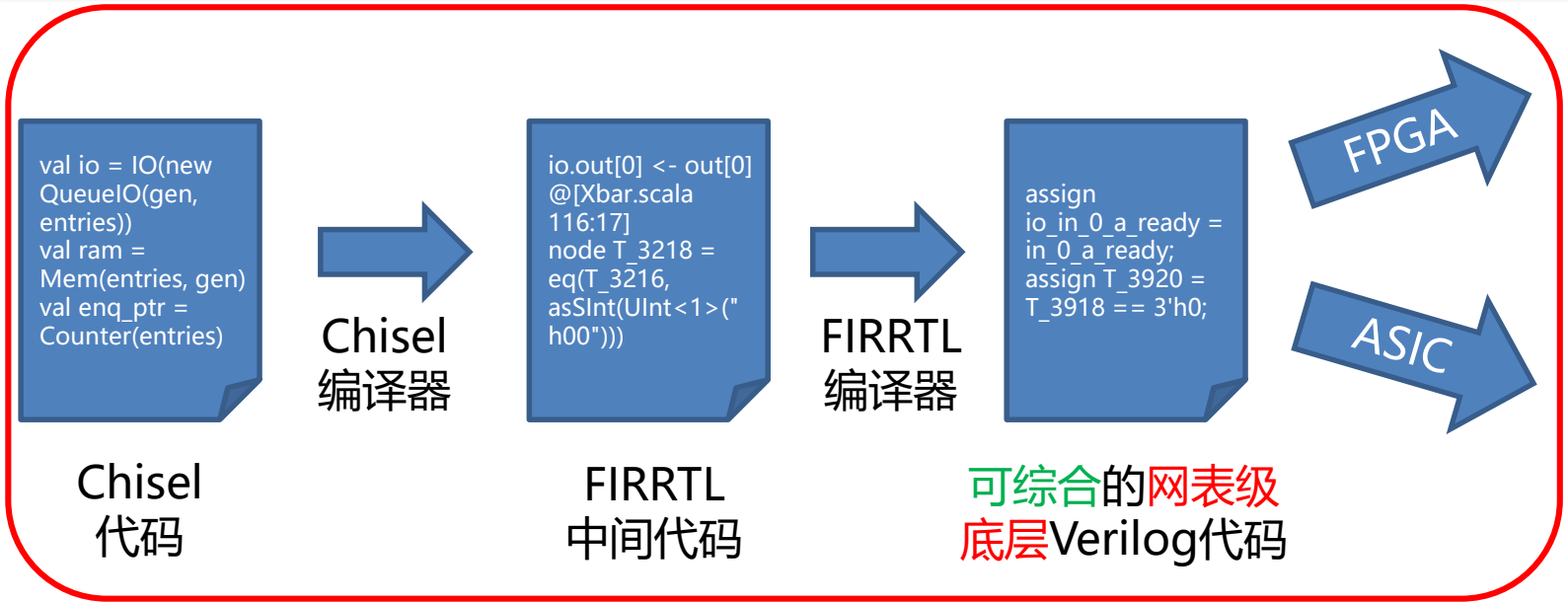
UInt bit-reduction methods:

Chisel	Explanation	Width
`x.andR`	AND-reduce	1
`x.orR`	OR-reduce	1
`x.xorR`	XOR-reduce	1

As an example to apply the andR method to an SInt use x.asUInt.andR

Chisel特性小抄 https://github.com/freechipsproject/chisel-cheatsheet/releases/latest/download/chisel_cheatsheet.pdf

开发流程




```
 1 class MovingAverage3(bitWidth: Int) extends Module {
 2   val io = IO(new Bundle {
 3     val in = Input(UInt(bitWidth.W))
 4     val out = Output(UInt(bitWidth.W))
 5   })
 6
 7   val z1 = RegNext(io.in)
 8   val z2 = RegNext(z1)
 9
10   io.out := (io.in * 1.U) + (z1 * 1.U) + (z2 * 1.U)
11 }
```

bitWidth = 8

```
module MovingAverage3(
  input        clock,
  input        reset,
  input  [7:0] io_in,
  output [7:0] io_out
);
  reg [7:0] z1; // @[cmd2.sc 7:19]
  reg [7:0] z2; // @[cmd2.sc 8:19]
  wire [8:0] _T; // @[cmd2.sc 10:20]
  wire [8:0] _T_1; // @[cmd2.sc 10:33]
  wire [8:0] _T_3; // @[cmd2.sc 10:27]
  wire [8:0] _T_4; // @[cmd2.sc 10:46]
  wire [8:0] _T_6; // @[cmd2.sc 10:40]
  assign _T = io_in * 8'h1; // @[cmd2.sc 10:20]
  assign _T_1 = z1 * 8'h1; // @[cmd2.sc 10:33]
  assign _T_3 = _T + _T_1; // @[cmd2.sc 10:27]
  assign _T_4 = z2 * 8'h1; // @[cmd2.sc 10:46]
  assign _T_6 = _T_3 + _T_4; // @[cmd2.sc 10:40]
  assign io_out = _T_6[7:0]; // @[cmd2.sc 10:10]

  always @(posedge clock) begin
    z1 <= io_in;
    z2 <= z1;
  end
endmodule
```

开发流程中的五类错误

- Scala编译错误
  - 语法错, 静态类型检查错, 继承错
- Scala运行错误
  - 引用null对象, 动态类型检查错, require断言失败
- Chisel编译错误
  - 连线方向错, Chisel类型/Chisel对象检查错
- FIRRTL编译错误
  - 未连接信号, 组合回环
- 电路仿真/运行错误
  - assert断言失败, 电路功能错

```
val a = WireInit(Bool())
val b = Reg(8.U)
```

```
val a = Wire(Bool())
val b = RegInit(8.U)
```

我只是想写个电路而已，为什么要这么麻烦?

- 软件工程领域中调试理论的三个概念
  - fault: 有错误的代码, 例如变量忘记初始化
  - error: 程序执行时不符合预期的状态, 例如函数返回垃圾值
  - failure: 能直接观测到的错误, 例如程序触发了段错误
- Bug传播链条
  - fault ->(不一定触发) error ->(不一定触发) failure

- 在Verilog中, 连错信号 -> CPU读数据不对 -> 访问非法地址
  - 结果: 调试一周

我只是想写个电路而已，为什么要这么麻烦？(2)

- 在Verilog中，连错信号 -> CPU读数据不对 -> 访问非法地址
  - 结果：调试一周
- 在Chisel中，连错信号 ->(很大可能) scala静态类型检查错
  - 结果：很大可能马上发现fault

- 能编译的代码 ≠ 正确的代码
- 更强大的编译器能把更多潜在的fault暴露成可观测的failure
  - 使用更强大的编译器 = 代码正确的概率更大
  - 即使花一上午解决一个Chisel编译错误，长远来看也是值得的

Scala类型/对象和Chise类型/对象

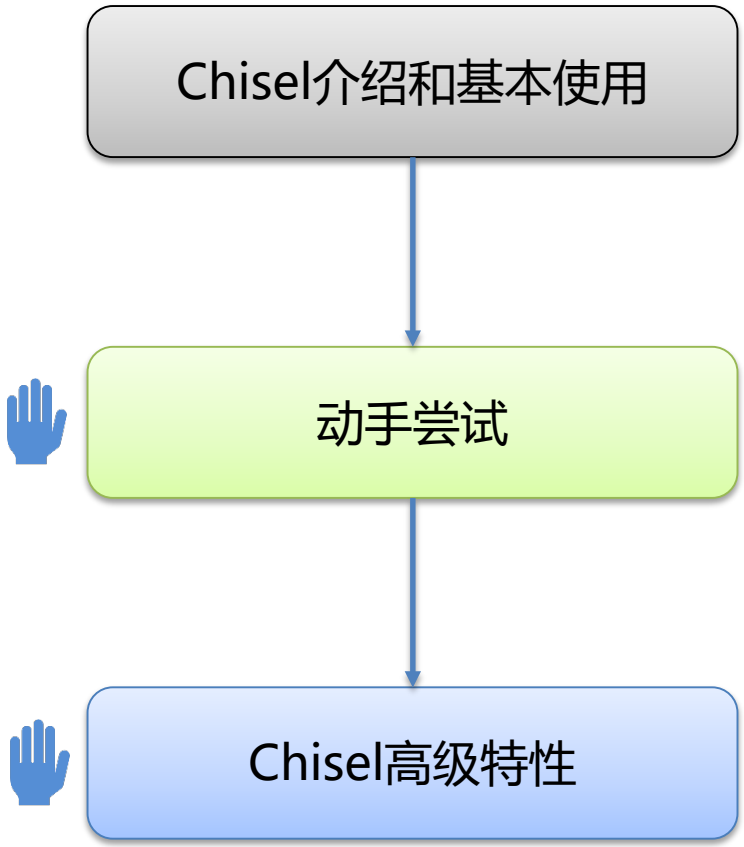
- Chisel对象是电路元素, 最后出现在电路中
- Scala对象用来辅助生成器进行工作, 最后不出现在电路中
  - 对比之前C语言和计算器的例子

	Scala	例子	Verilog	Chisel	例子	Verilog
整数	Int, BigInt	val x: Int = 10	localparam x = 10;	UInt, SInt	3.U(4.W)	4'd3
布尔	Boolean	val x: Boolean = true	localparam x = true;	Bool	false.B	1'b0
赋值	= (取名)	val x = true.B	wire x = 1'b1;	:= (连接)	y := true.B	assign y = 1'b1;
运算	+	val x: Int = 1 + 2	localparam x = 1 + 2;	+	y := a + b	assign y = a + b;

- Verilog的parameter, for, generate也可看做一个生成器系统
  - 但语法上和电路代码一样, 而Chisel和Scala严格区分两者
- **识别方法: 这里是在写电路, 还是在写生成电路的生成器?**

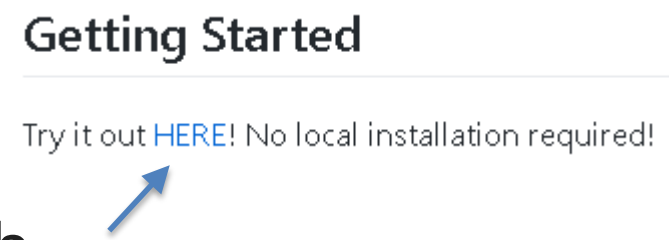
第二部分实验内容

Chisel - 面向敏捷开发的硬件设计语言



动手尝试

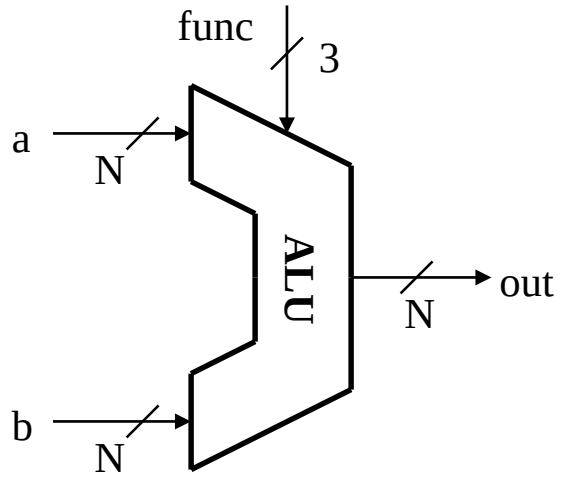
- 打开https://github.com/freechipsproject/chisel-bootcamp
- 点击如下链接, 进入在线学习环境jupyter



- 打开0_demo.ipynb
- 选中前两个代码块, 按下shift + enter运行, 来加载环境
- 加载完成后, 在第三个代码块(class MovingAverage3)最后换行, 键入println(getVerilog(new MovingAverage3(8))), 并按下shift + enter运行, 成功后可看到生成的Verilog代码
- 可以删除代码块的内容, 键入自己编写的代码并运行
- 若掉线, 则需要从第二步重新开始进入jupyter

动手尝试1 – 简单ALU

- N可配置
- 尝试在chisel-bootcamp中编写Chisel, 生成并观察Verilog



func[2:0]	说明
000	按位与
001	按位或
010	加
110	减

Chisel特性小抄 https://github.com/freechipsproject/chisel-cheatsheet/releases/latest/download/chisel_cheatsheet.pdf

动手尝试2 – 布尔开关

- 实现布尔开关
  - start有效时, 开关输出1; stop有效时输出0; 都无效则保持输出; 同时有效时输出0
  - 通过Scala函数实现可方便使用

```
object BoolToggle {
  def apply(start: Bool, stop: Bool) = {
    // add code here
}}
class MyTest {
  val io = …
  io.out := BoolToggle(io.start, io.stop)
}
// 使用举例, 其中x.fire()含义为 x.valid && x.ready
// val rBusy = BoolToggle(axi.ar.fire(), axi.r.fire())
```

提示: Chisel可以对电路元素进行多次赋值, 电路元素的新值以最后一次有效的赋值为准

```
a := b0
when (c1) { a := b1 }
when (c2) { a := b2 }
```

c1	c2	a的新值
true.B	true.B	b2
true.B	false.B	b1
false.B	true.B	b2
false.B	false.B	b0

- 添加一个Boolean类型的生成器参数startHighPriority
  - 若startHighPriority为true, 则同时有效时改为输出1, 其余不变

动手尝试3 – 计数器和生成器

- (1) 写一个最大值为N的计数器, 每周期加1, 超过最大值后归0, 计数器的值作为模块的输出
- (2) 改成每T周期加K的计数器生成器, T和K均为生成器参数
- (3) 增加一个生成器参数isConst
  - isConst为true时, T和K从生成器参数获得
  - isConst为false时, T和K从模块输入端获取
  - 提示: 可以通过以下方式声明并使用可选端口

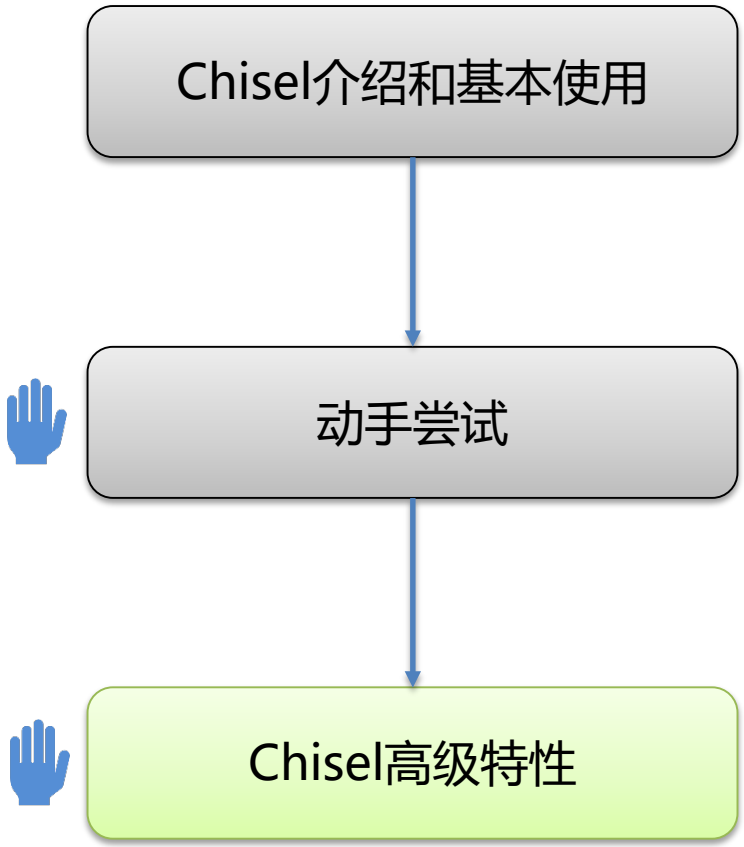
```
class MyModule(flag: Boolean) extends Module {
  val io = IO(new Bundle {
    val a = if (flag) Some(Output(UInt(2.W))) else None
  })
  if (flag) { io.a.get := 0.U }
}
```

你已经可以用Chisel写出所有可综合电路了!

- 用Verilog写可综合电路只是在做两件事
  - 实例化: 包括各种门电路(与, 或, 加法器等), 触发器, 模块
  - 连线: 用wire把这些实例连起来
- 这两件事Chisel也能做!
  - 从这个角度来说, Chisel入门也不难
  - 但要用好就是另一回事了
- 而且Chisel只能编写可综合电路
  - Chisel中没有"不可综合"的概念
  - 事实上, Verilog的本质是一门事件建模语言
    - 它并不是天生就用来写可综合电路的, 所以多驱动代码符合Verilog标准

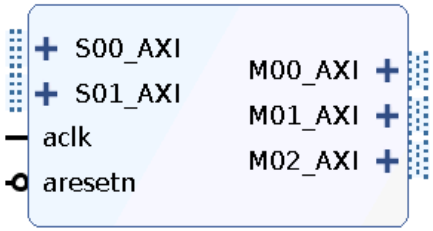
第二部分实验内容

Chisel - 面向敏捷开发的硬件设计语言



Verilog编码"基本功" – 连线

- 大部分模块之间通过总线连接
  - 完整的AXI总线有40多个信号: 有点麻烦, 耐心点连, 还行
  - 实例化一个2进3出的AXI crossbar: 要连40 * 5 = 200根信号, 忍!
  - 实例化3个这样的crossbar: 你肯定受不了



来源: Vivado IP核配置界面

- 别高兴太早, 还会连错呢
  - 连线错误经常发生在模块集成的过程中
    - input/output弄反了
    - 不小心漏了个信号没连
    - arvalid和rvalid只差一个a
    - 信号位宽定义疏忽

```
.mem_axi4_0_aw_ready(dut_mem_axi4_0_aw_ready),
.mem_axi4_0_aw_valid(dut_mem_axi4_0_aw_valid),
.mem_axi4_0_aw_bits_id(dut_mem_axi4_0_aw_bits_id),
.mem_axi4_0_aw_bits_addr(dut_mem_axi4_0_aw_bits_addr),
.mem_axi4_0_aw_bits_len(dut_mem_axi4_0_aw_bits_len),
.mem_axi4_0_aw_bits_size(dut_mem_axi4_0_aw_bits_size),
.mem_axi4_0_aw_bits_burst(dut_mem_axi4_0_aw_bits_burst),
.mem_axi4_0_aw_bits_lock(dut_mem_axi4_0_aw_bits_lock),
.mem_axi4_0_aw_bits_cache(dut_mem_axi4_0_aw_bits_cache),
.mem_axi4_0_aw_bits_prot(dut_mem_axi4_0_aw_bits_prot),
.mem_axi4_0_aw_bits_qos(dut_mem_axi4_0_aw_bits_qos),
```

- 手工连线 = 毫无技术含量的重复性脏活

信号整体连接

- Chisel可以定义新的信号类型
  - 类似C语言的结构体

```
class AXI4 extends Bundle {
  val ar = new AXI4BundleA
  val  r = Flipped(new AXI4BundleR)  // 翻转端口方向
  val aw = new AXI4BundleA
  val  w = new AXI4BundleW
  val  b = Flipped(new AXI4BundleB)
}
```

- 然后通过运算符<>对同类型的两组信号进行整体连接
  - core(0).out <> arbiter.in(0)
- 动手尝试4 – 总线连接

```
class MySoC extends Module {
  val io = IO(new Bundle {}) // 无需其他端口
  val cpu = Module(new CPU) // out: cpu.io.icache, cpu.io.dcache
  val mem = Module(new AXI4Mem) // in: mem.io.in
  val uart = Module(new AXI4UART) // in: uart.io.in
  val timer = Module(new AXI4Timer) // in: timer.io.in
  val xbar = Module(new AXI4Xbar(in = 2, out = 3)) // in: xbar.io.in(0)…

  // add code to connect the SoC
}
```

信号整体连接

- Labeled RISC-V项目: 在总线上添加标签(新成员)并传播只要4行代码
  - 原因 – 代码中使用<>对总线进行连接, 修改总线定义时, 一改全改

```
+trait HasDsid extends HasTileLinkParameters {
+  val dsid = UInt(width = tlDsidBits)
+}
 class AcquireMetadata(implicit p: Parameters) extends ClientToManagerChannel
   with HasCacheBlockAddress
+  with HasDsid
   with HasClientTransactionId

     core(0).out <> arbiter.in(0)
```

- 展示了需求变更时可快速拥抱变化
  - Verilog需要进行全局修改, 非常麻烦
  - SystemVerilog有interface特性, 但无法嵌套

基于Scala的元编程 – 泛型

- 借助Scala的泛型特性对抽象出Chisel代码的共性部分
  - 可以编写更强大的生成器(类型也可以参数化了)
  - Chisel库中的例子 – 用泛型实现队列的原型

```
1 class Queue[T <: Data](gen: T, val entries: Int) extends Module() {
2   val io = IO(new QueueIO(gen, entries))
3   val ram = Mem(entries, gen)
4   // ...
  val q = Module(new Queue(new TileLinkBundle, 8))
```

- Labeled RISC-V项目: 我们希望标签随请求一同穿过各种缓冲队列, 但无需修改任何代码
  - 原因 – 用泛型特性实现的队列原型, 可适用于任意类型的元素
  - 若使用Verilog, 需全局手动增加队列元素的宽度, 十分繁琐
  - SV也支持模板, 但代码不可综合[1], 大多用于编写测试激励

[1] https://stackoverflow.com/questions/20312810/what-systemverilog-features-should-be-avoided-in-synthesis

面向对象编程 – 继承

- 重用父类的特性, 减少冗余代码
  - 还能让类型检查的过程更严格
- 例子 – 继承总线的各种参数

```
 1 class TileLinkTrafficGenerator(implicit p: Parameters) extends TLModule()(p) {
 2   val io = IO(new Bundle {
 3     val out = new ClientTileLinkIO
 4     val traffic_enable = Bool().asInput
 5   })
20  // ...
```

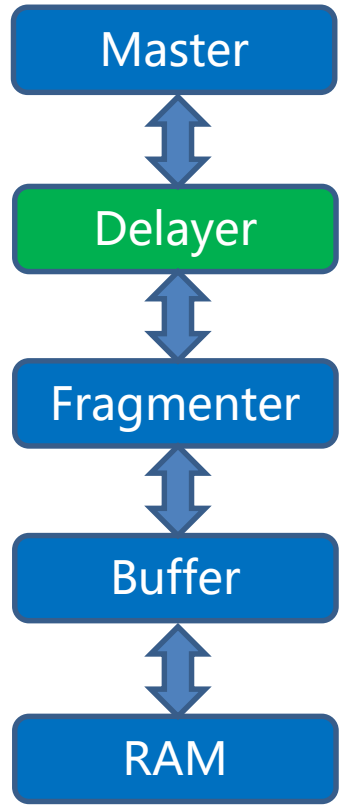
- 负载发生器模块自动拥有TLModule的所有特性
  - 包括总线的大量参数, 如地址位宽, 数据位宽, ID位宽等
  - 可在第3行直接定义TileLink端口, 无需显式指定参数
- Verilog不支持继承, 需用10倍代码编写此模块
- SystemVerilog部分支持继承, 但代码不可综合[1]

[1] https://stackoverflow.com/questions/20312810/what-systemverilog-features-should-be-avoided-in-synthesis

面向对象编程 – 重载

- 运算符重载可重新定义运算符的行为
  - 提高代码的可读性
- 例 – Diplomacy对":="运算符进行重载
  - 让其两侧AXI4节点的主从端口使用<>连接
- 可通过少量代码在数据通路上添加延迟器

```
val node = AXI4MasterNode(List(edge.master))
val sram = LazyModule(new AXI4RAM(...))
sram.node := AXI4Buffer() := AXI4Fragmenter() := AXI4Delayer(0, 150) := node
```



- Verilog和SystemVerilog均不支持重载
  - 只能用模块来实例化, 并引入大量连线

函数式编程

- 编写更紧凑, 可读性更好的代码
- 使用容器(collection)来抽象电路元素
  - 容器中可以是信号, 寄存器, 端口, 模块, 映射等等
    - 或者是这些元素的复合
- 使用map/reduce算子对容器中的对象进行批量操作
  - 操作可以是连接, 归约, 算术和逻辑运算, 选择, 实例化, 函数调用, 计算新映射等等
    - 或者是这些操作的复合
  - 操作结果返回1个新的容器

函数式编程

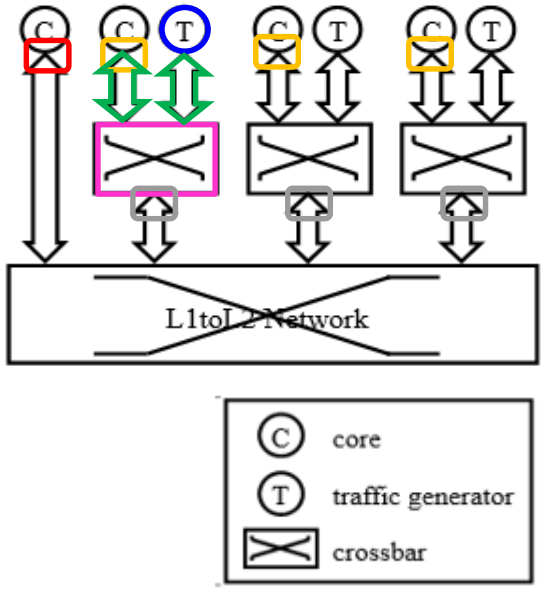
- 例1 – 性能计数器

```
val perfCnts = List.fill(N)(RegInit(64.W))) // 定义N个性能计数器
// 定义(地址, 性能计数器)的二元关系
val addrMap = (0 until N).map { case i => (0xb00 + i, perfCnts(i)) }
rdata := LookupTree(raddr, addrMap.map { case (a, r) => (a.U, r) }) //读
(perfCnts zip conds).map { case (r, c) => when (c) { r := r + 1.U } } //更新
```

- 例2 –

如图所示接入负载发生器
只在第2~n个核外接入, n可变

```
1 val ports = cachedOut.take(1) ++
2   cachedOut.drop(1).map { case p => {
3       val tg = Module(new TileLinkTrafficGenerator()(p_alter))
4       tg.io.traffic_enable := io.traffic_enable
5       val arb = Module(new ClientTileLinkIOArbiter(2)(p_alter))
6       arb.io.in <> List(p, tg.io.out)
7       arb.io.out
8     }
9   }
10 network.in <> ports
```



动手尝试5 – 地址空间判断

- 判断一个地址是否落在MMIO的范围

```
object AddressSpace {
  def mmio = List(
   (0x40000000, 0x1000), // uart
   (0x41000000, 0x4000)  // timer
  )
  def isMMIO(addr: UInt) = ??? // add code here
}

class MyTest {
  val io = IO(new Bundle {
    val addr = Input(UInt(32.W))
    val isMMIO = Output(Bool())
  })

  io.isMMIO := AddressSpace.isMMIO(io.addr)
}
```

```
// 提示: 可以通过_1, _2等取出元组中的元素
val tupple = (“cat”, “dog”, “fish”)
val second = tupple._2 // “dog”
```

动手尝试6 – key-value查找

- 实现UInt到任意Chisel数据类型的key-value查找
  - 利用Chisel库函数Mux1H, 它可以进行基于独热码的选择

```
// Mux1H[T <: Data](in: Iterable[(Bool, T)]): T

// example
val list = Iterable((false.B, 1.U), (true.B, 2.U), (false.B, 3.U))
val result = Mux1H(list) // result = 2.U


object LookupTree {
  def apply[T <: Data](key: UInt, mapping: Iterable[(UInt, T)]): T = ???
}
```

Chisel高级特性小结

- 更多说明可参考《芯片敏捷开发实践：标签化RISC-V 》[1]

特性	Chisel	SystemVerilog	Verilog
信号整体连接	支持	支持，但有限制	不支持
元编程	支持	部分支持，且不可综合	不支持
面向对象编程	支持	部分支持，且不可综合	不支持
函数式编程	支持	不支持	不支持

- Chisel这些强大的功能，都是基于Scala强大的类型系统
  - Scala的类型可以携带丰富的语义信息
  - 从这些语义出发，又可以衍生出各种复杂的操作
  - Verilog的类型无法做到

[1] http://crad.ict.ac.cn/CN/10.7544/issn1000-1239.2019.20180771

小结: Chisel, HLS和传统HDL

- Chisel的优势在于编写生成器, 来多快好省地描述电路
  - 无论生成器写得多复杂, 我们仍然在描述电路
- **更本质地: Chisel的抽象并未失去对生成电路的精确控制**
  - 例如容器中的对象和算子的操作都是电路中的概念
  - HLS描述算法, 失去了对生成电路的精确控制
- 传统HDL层次低(实例化+连线), 难以方便地描述复杂电路
  - Verilog的for和generate机制只能对整数迭代
- 敏捷开发需要改变观念: 优先编写和维护生成器, 而非实例

Chisel学习资料

- Chisel-bootcamp在线学习环境
  - https://github.com/freechipsproject/chisel-bootcamp
- Chisel特性小抄
  - https://github.com/freechipsproject/chisel-cheatsheet/releases/latest/download/chisel_cheatsheet.pdf
- Chisel Users Guide
  - https://github.com/freechipsproject/chisel3/wiki/Short-Users-Guide-to-Chisel
- Chisel常见问题
  - https://github.com/freechipsproject/chisel3/wiki
- Chisel API文档
  - https://www.chisel-lang.org/api/latest/#package
- Chisel本地开发环境
  - https://github.com/freechipsproject/chisel-template

Chisel学习资料 – 国内的社区贡献

chisel-bootcamp中文翻译版

https://github.com/sequencer/chisel-bootcamp/tree/zh_cn

译者: sequencer (右一)

Chisel内部机制解释

http://funof.ml/build-chisel-from-scratch/post/getting-started

作者: 咧威 (左一)

OS2ATC 2019 – CPU Tutorial

RISC-V开源处理器及Chisel硬件敏捷开发语言入门

谢 谢！

张　科、余子濠、陈欲晓

crva@ict.ac.cn

RV教程群

CRVA联盟

中国科学院计算技术研究所
Institute of Computing Technology, Chinese Academy of Sciences

鹏城实验室
Peng Cheng Laboratory

中国开放指令生态(RISC-V)联盟
China RISC-V Alliance