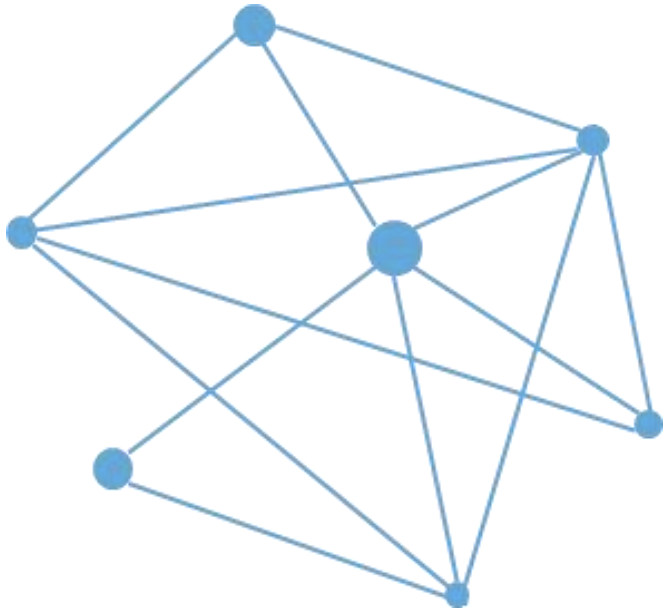


Risc-V SoC
在硅径迹探测器上的研发进展

报告人：崔宇鑫

合作导师：严琪 研究员

参与研发团队的其他成员：骆首栋、张理凯、程鼎麟

时　间：2025-5-16

个人简介

本人毕业于中国科学院紫金山天文台，硕博期间一直从事硅微条探测器相关模拟、标定、重建工作，先后参与了暗物质粒子探测卫星（DAMPE）、高能宇宙辐射探测设施（HERD）、甚大面积伽马射线空间望远镜计划(VLAST)项目

1、负责开发了DAMPE卫星的整体位置校准方法，成功解决了DAMPE合作组原有准直算法中的径迹偏移及校准算法速度过慢的问题。

doi:10.1016/j.nima.2022.167670

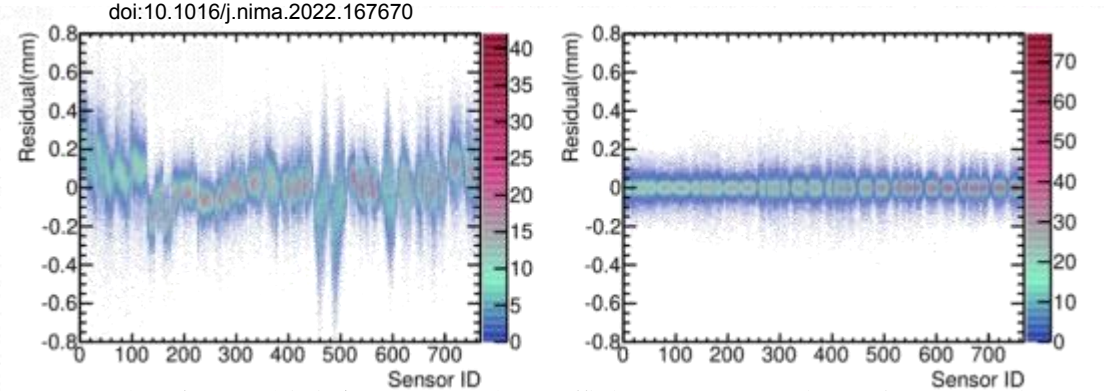


图1、左、右图分别为准直前后，不同探测器模块(Sensor)的位置残差分布。稳定的残差分布表明很好的消除了原有的位置偏移

2、利用Allpix²框架实现了DAMPE硅微条探测器的高精度模拟，解决了原有模拟和在轨数据差异较大的问题，同时也为后续半导体项目搭建了平台

doi:10.1016/j.nima.2023.168685

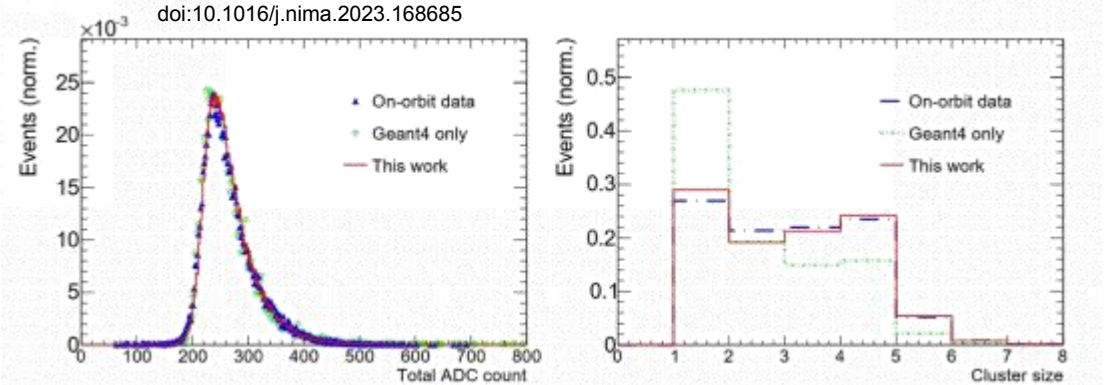
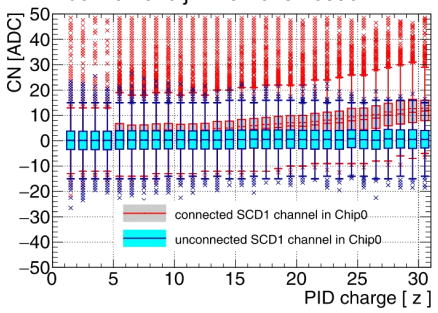


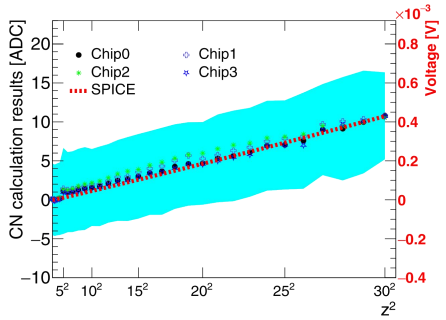
图2、垂直入射的氦MIP（最小电离粒子）信号簇宽度的在轨数据和模拟数据之间的比较。红色的线条为该工作的结果，相比于原有结果和数据更加吻合。

3、利用高能宇宙辐射探测设施（HERD）项目束流试验数据和SPICE仿真研究了硅微条探测器中共模噪声的来源及其计算方法，为后续半导体探测器项目的共模噪声算法提供了理论基础。

doi:10.1016/j.nima.2023.168962

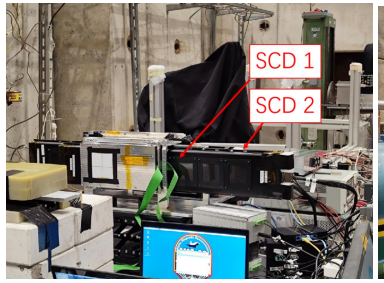


悬空通道和连接传感器通道共模噪声与电荷关系



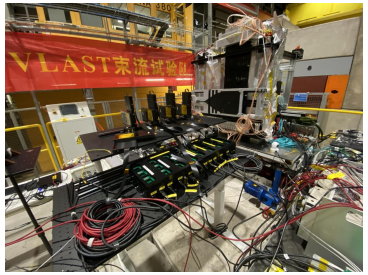
共模噪声与电荷平方的关系

4、负责甚大面积伽马射线空间望远镜计划(VLAST)预研项目的硅微条探测器的测试、数据处理等工作，并多次完成了在CERN进行的束流实验的相关数据处理工作。



2022年束流试验

2023年VLAST束流试验



2024年VLAST束流试验

目录

- System-on-Chip简介
  - 指令集架构(ISA)简介
  - Risc-V简介
  - Risc-V SoC的部分应用
- 当前研究工作
  - 基于Risc-V的读出测试板研发
  - 集成于ASIC的Risc-V 微处理器（MCU）研发
- 总结

一、System-on-Chip简介

System on Chip(SoC)的研究动机

SoC: 将微处理器、模拟IP核、数字IP核和存储器(或片外存储控制接口)集成在单一芯片上，通常是面向特定用途的标准产品，CERN DRD 7.2 正是有关抗辐照的Riscv SoC的应用，SoC应用到高能物理领域可以实现：

实现“可编程”探测器

- 允许将ASIC重新定向用于不同应用场景
- 在运行时可以动态更改算法

简化系统集成并缩短设计时间

- 模块化设计，具有很好的兼容性与可拓展性
- 加速数字设计流程并加快验证速度
- 加速物理实现

提供抗辐射数字IP模块库

- 提供预先验证的模块
- 所有IP均与标准化互连协议兼容
- 促进社区内设计复用

应用方向：

- 控制和监控
  - 独立的抗辐照MCU：LHC束流监测、前端集成的慢控制核心、电源管理、探测器状态监控（温度电压等）
  - 嵌入ASIC的控制器（例如硅探测器芯片集成CPU控制模块）：动态芯片参数配置、像素芯片读出等（可重新编程）
- 前端数据处理
  - 数据压缩与实时处理：实时数据过滤[1][2]、波形采样数据预处理等

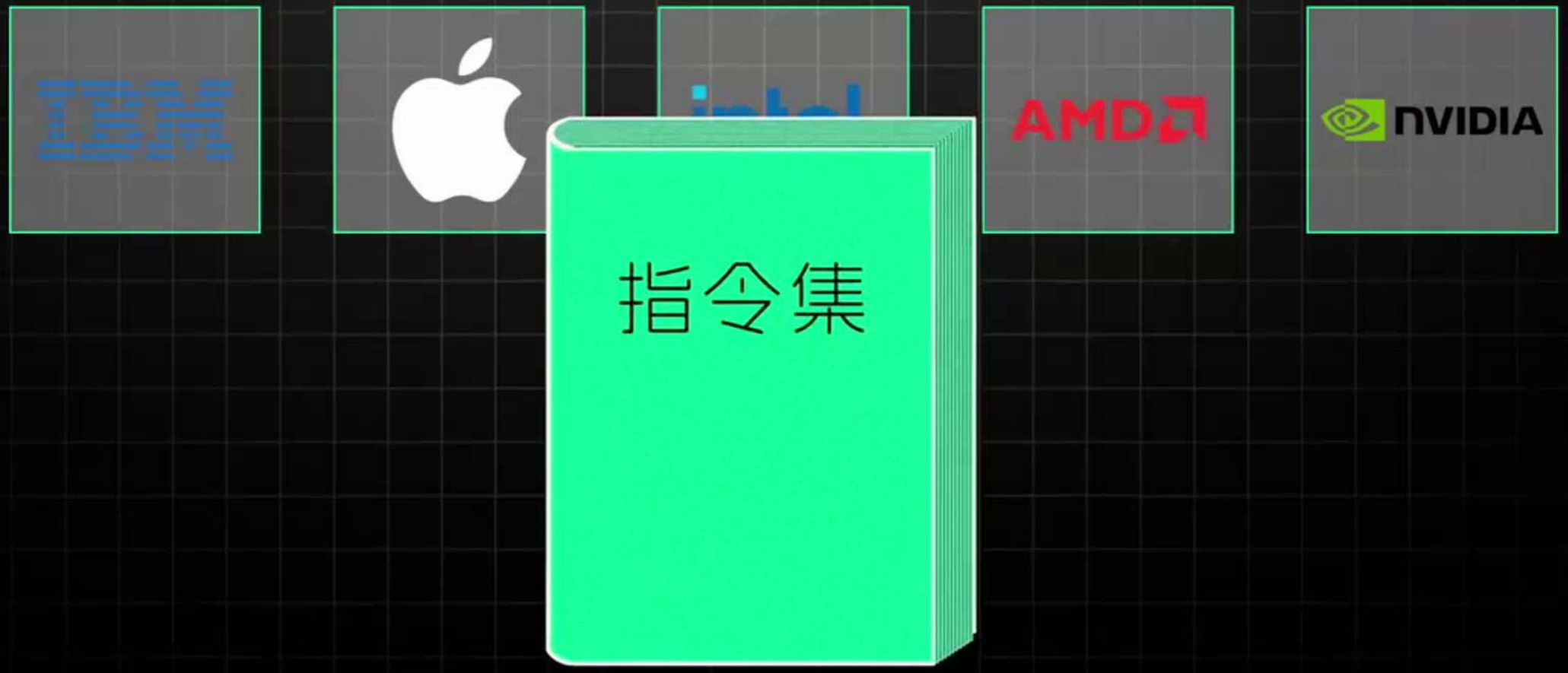
[1] On-Sensor Data Filtering using Neuromorphic Computing for High Energy Physics Experiments
[2] Intelligent pixel detectors: towards a radiation hard ASIC with on-chip machine learning in 28 nm CMOS
cite:https://indico.cern.ch/event/1436991/contributions/6046661/attachments/2893748/5131661/2024.09.09_DRD7_SOC.pdf

指令集架构Instruction Set Architecture (ISA)

为什么要ISA：为上层软件提供一层抽象层，制定规则和约束，让编程者不用操心具体的电路结构

- **CISC(Complex Instruction Set Computer，复杂指令集)**: X86
  - 指令系统复杂庞大，指令数目一般多达200~300条
  - 指令长度不固定，指令格式种类多，寻址方式种类多
  - 可以访存的指令不受限制（RISC只有取数/存数指令访问存储器）
  - 各种指令执行时间相差很大，大多数指令需多个时钟周期才能完成
  - 编译器优化难度较大

- **RISC(Reduced Instruction Set Computer，精简指令集)**: Arm、Risc-V、MIPS
  - 选取使用频率较高的一些简单指令、很有用但不复杂的指令，让复杂指令的功能由简单指令的组合来实现
  - 指令长度固定，指令格式种类少（通常几十到上百条），寻址方式种类少
  - 只有取数/存数指令访问存储器，其余指令的操作都在寄存器内完成，有多个通用寄存器（RiscV一般是32个、Arm有31个、X86是16个）
  - 编译器容易优化



就轮到指令集登场了

Risc-V的优势

- 完全开源（自由、免费、可控）
  - X86极少对外授权，较为封闭；Arm授权费和版税结合，较为昂贵
  - 为避免政治风险，2019年，Risc-V 基金会总部从美国迁移至瑞士
  - 活跃的开源社区：全球开发者贡献工具链（如 GCC、LLVM）、调试器、仿真器（如 QEMU）和 IP 核，多家公司、机构（如 SiFive、香山、阿里）提供开源软核和开发工具
    - **PULP**，提供Risc-V软核和IP的开源平台，覆盖范围从MCU到多核 Risc-V芯片
    - **SOCRATES**，以 SoCMake 为中心的抗辐照 SoC 生成器工具集
    - **OpenLANE**，可提供从RTL到GDSII的全自动流片流程的开源项目
    - **EXTREM-EDGE**，辅助设计Risc-V，实现硬件加速的工具包
    - ……

- 模块化设计，架构简单
  - Arm和X86经过长期发展，积累了许多历史包袱：
    - 例如,AMD64面向 64 位开发与应用环境;但它同时仍须要向后兼容 32 位甚至 16 位的 x86 架构
  - “基础指令集+扩展指令集”的架构设计，可根据需求灵活定制指令集。
    - 例如，添加浮点运算（F/D扩展）、原子操作（A扩展）等模块，适配从嵌入式设备到高性能计算的不同场景

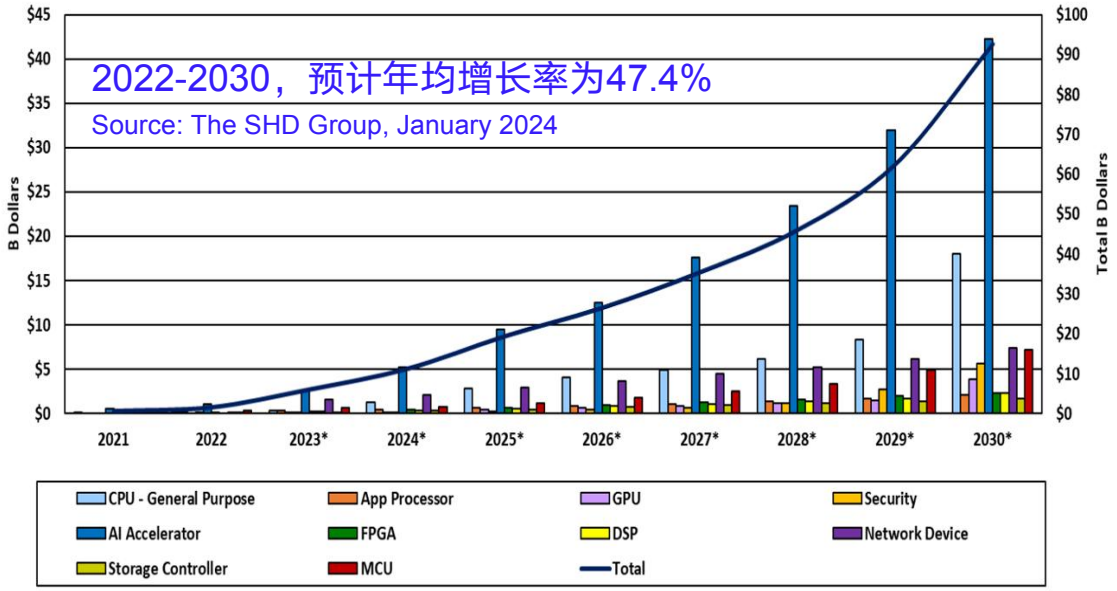
X86指令数量不断增加，日渐复杂

- 符合软硬件协同的趋势，可以实现面向需求的系统精准定制化
  - 英伟达的CUDA 通用并行计算架构的成功证明了软硬件协同的优势[1]
  - 可以根据特定应用场景，实现硬件加速，如NASA HPSC项目[2]

[1]10.13328/j.cnki.jos.006490
[2] https://www.nasa.gov/game-changing-development-projects/high-performance-spaceflight-computing-hpsc/

Risc-V的发展现状

Risc-V市场规模预测（2021-2030）



RISC-V 成员超过4,500个，分布于70 个国家/地区

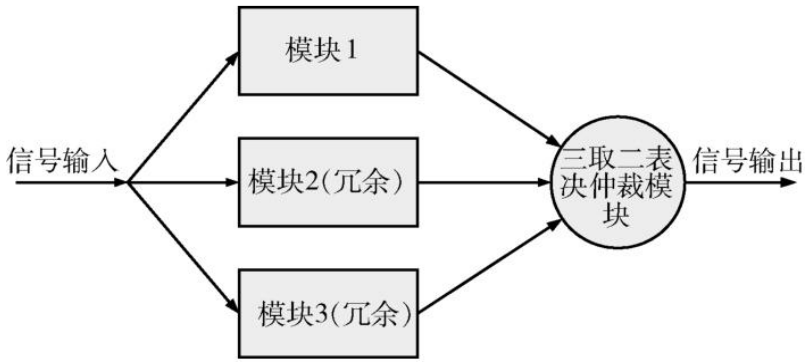
大量知名机构、公司参与，市场规模日益壮大

- **阿里玄铁910**：32位RISC-V处理器，主频达1GHz，采用40纳米工艺制造，主要用于物联网设备。
- **华为HiSilicon Hi3559A**：32位RISC-V处理器，主频达1.5GHz，采用28纳米工艺制造，主要用于智能家居和安防监控设备。
- **香山**：第三代“昆明湖”架构开源高性能RISC-V处理器核：设计工艺为7nm，主频达到3GHz，SPECINT2006 评分为15分/GHz，性能对标Arm Neoverse N2内核，可广泛应用于服务器芯片、AI芯片、GPU、DPU等高端芯片领域
- **玄铁C930**：今年2月发布，支持 512 bit 的 RVV 以及 Matrix extension，SPECINT2006 达到 15/GHz，用于服务器级高性能处理器

香山、玄铁、PULP平台等均开放设计工具和源码

Risc-V SoC在高能物理领域最大挑战：抗辐照设计

➢ 最简单也是最传统的方式：三模冗余(Triple Modular Redundancy,TMR)



三模冗余

- 原理简单
- 效果显著
- 更大面积和功耗开销，~3X

应用案例

• A. Walsemannet al., STRV —a radiation hard RISC-V microprocessor for high-energy physics applications, 2023, JINST 18

• M. Andorno et al., Rad-hard RISC-V SoC and ASIP ecosystems studies for high-energy physics applications, 2023, JINST 18

• A. E. Wilson and M. Wirthlin, Neutron Radiation Testing of Fault Tolerant RISC-V Soft Processor on Xilinx SRAM-based FPGAs, 2019, IEEE SCC

➢ 选择性加固、特殊处理机制

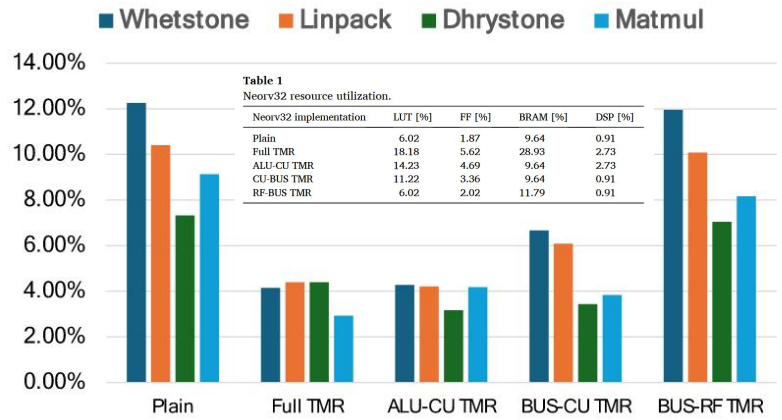
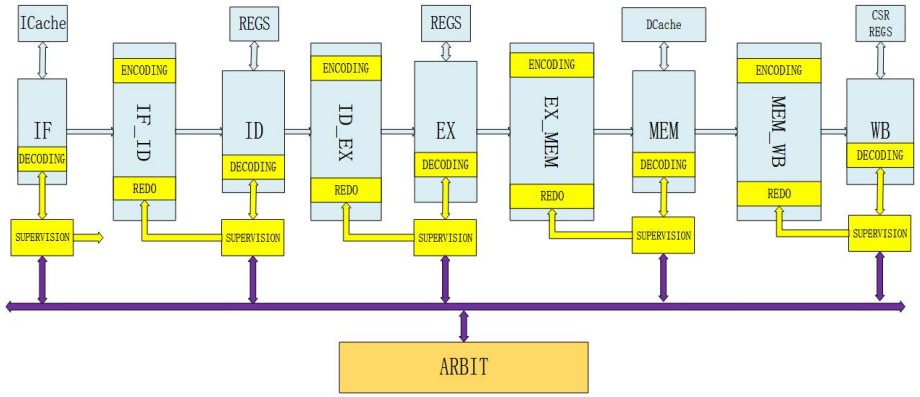


Table 1
Neorv32 resource utilization.

Neorv32 implementation	LUT [%]	FF [%]	BRAM [%]	DSP [%]
Plain	6.02	1.87	9.64	0.91
Full TMR	18.18	5.62	28.93	2.73
ALU-CU TMR	14.23	4.69	9.64	2.73
CU-BUS TMR	11.22	3.36	9.64	0.91
RF-BUS TMR	6.02	2.02	11.79	0.91

多比特翻转条件下不同模块加固的性能

doi.org/10.1016/j.microrel.2025.115667



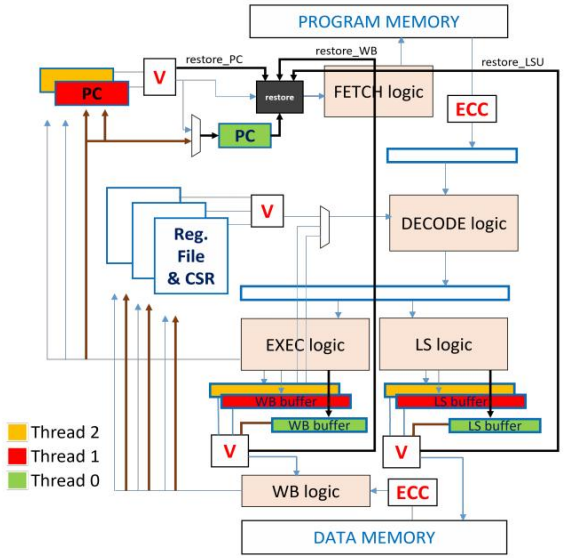
改进编码技术(32数据+7校验)和流水线回滚机制

doi.org/10.3390/electronics11010122

cite:https://indico.cern.ch/event/1214423/contributions/5184065/attachments/2609938/4510055/20230314_drdt7.2_FE_programmability.pdf

Risc-V SoC在高能物理领域面临的挑战：抗辐照设计

- 动态三重模块冗余（DTMR）方法



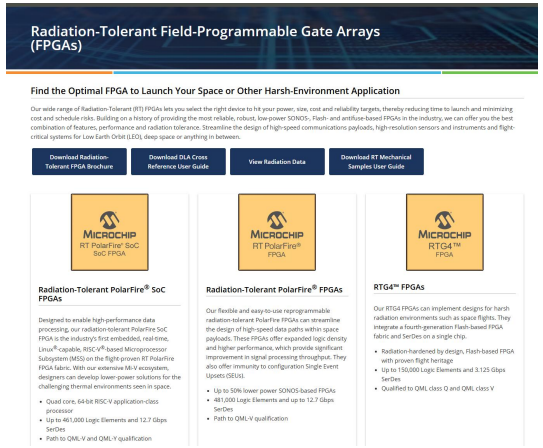
正常模式（DMR）：仅激活两个线程（Thread 1和2），共享流水线逻辑，通过投票检测错误。

恢复模式（TMR）：检测到故障后，唤醒第三个备用线程（Thread 0），三线程共同执行最后正确指令，通过多数投票纠正错误。

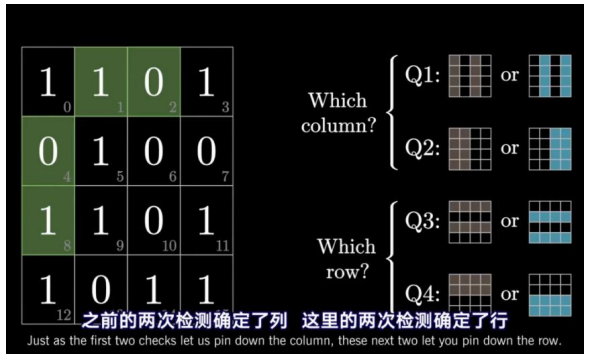
适合用于空间环境这类辐照不是很强的领域

cite: doi.org/10.3390/jlpea13010002

- 其他方法



抗辐照FPGA



汉明码（BCH码等）

https://www.bilibili.com/video/BV1WK411N7kz/?share_source=copy_web&vd_source=a3419c61e9172134d7d2b5a59a247988

Risc-V SoC实例1：混合像素探测器的片上校准

混合像素探测器通常由传感器层（例如硅传感器）和读出芯片 (ASIC) 两部分组成，由于晶体管尺寸非常小，制造过程中的微小工艺偏差会导致每个像素的读出电路的特性略有不同。需要对基线和增益进行标定

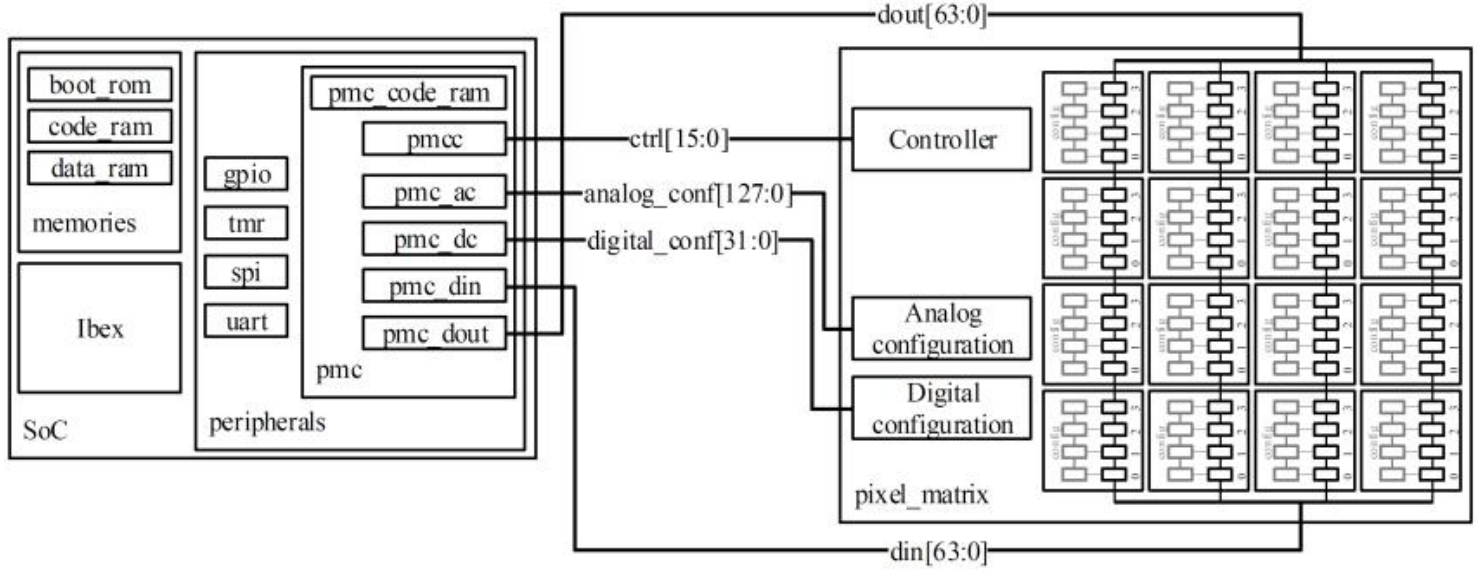
每个像素的读出电路内部都集成了数字模拟转换器 (DACs)。通过向这些DACs输入特定的数字值（称为“微调值”或“trimming values”），可以微调每个像素的模拟特性（如偏置电压或电流），从而补偿其偏移和增益。

例如，以下标定步骤：

- 用已知的、均匀的输入信号（例如，均匀的X射线束流）照射整个探测器。
- 读取所有像素的响应。
- 比较像素的响应，计算修正参数
- 调整这些像素的DAC微调值。
- 重复以上步骤，直到所有像素的响应都达到期望的均匀性和准确性。

TABLE III
SINGLE-PIXELS CALIBRATION TIMES.

	time [ms]	pixels	pixel calibration time [ms]
[8]	40 000	18 432	2.17
this work	805	1 024	0.79

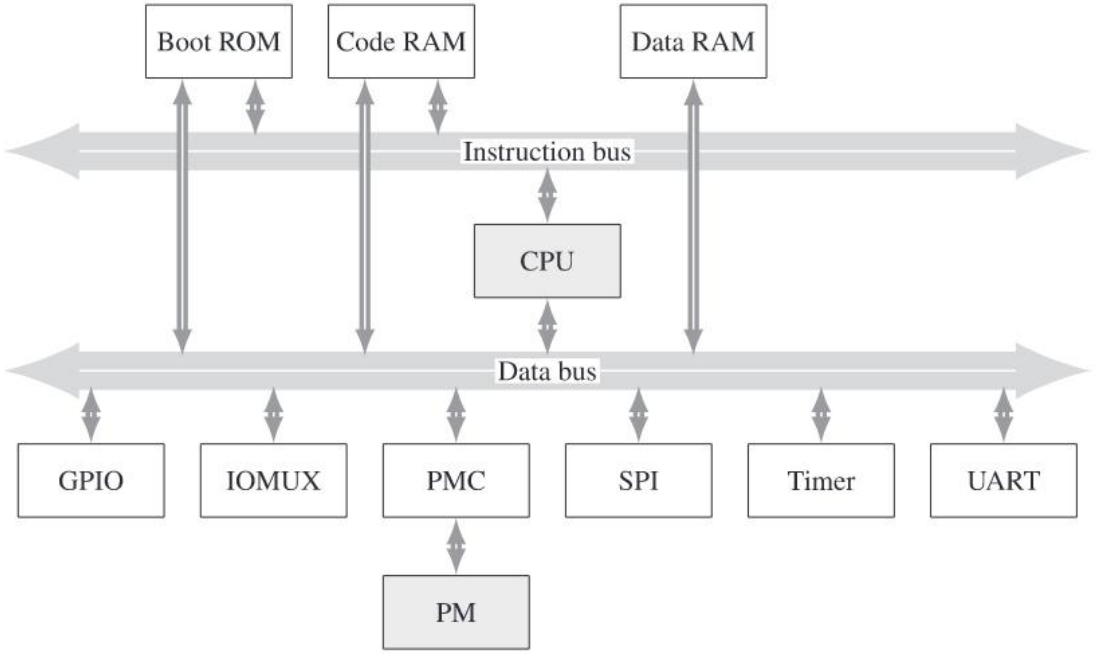


RISC-V在这项工作中充当了一个可编程的、高效的控制核心：与依赖外部辅助设备的传统校准方法相比，这种片上校准方案快了2.75倍。

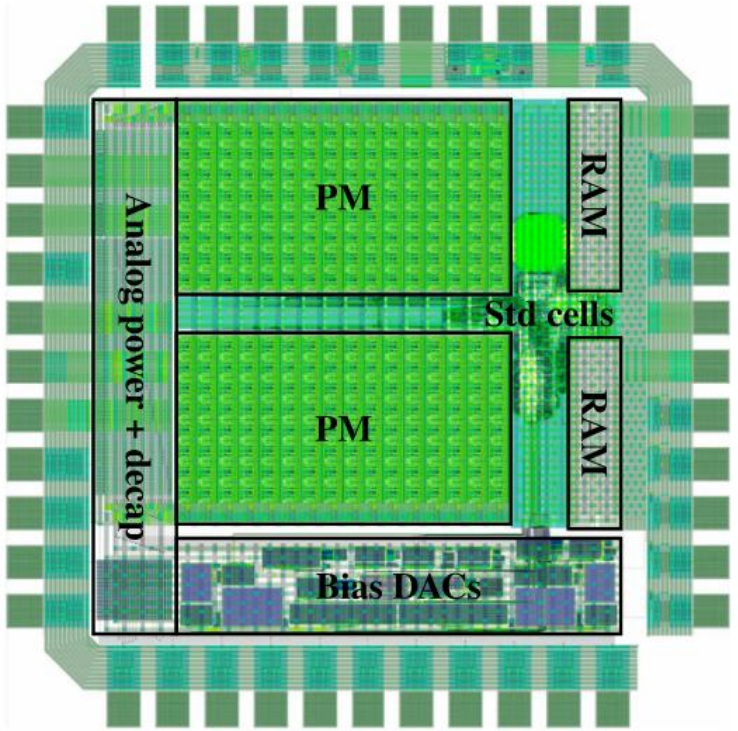
cite:10.1109/DDECS52668.2021.941702

Risc-V SoC实例2：像素探测器的片上微处理器

基于Risc-V架构，通过FPGA原型验证，并采用40纳米CMOS工艺生产。通过将像素矩阵与Risc-V中央处理器集成，设备无需外部辅助装置即可独立工作，并能执行如校准、阈值扫描和数据滤波、智能实时感兴趣区域过滤技术等功能



处理器架构图



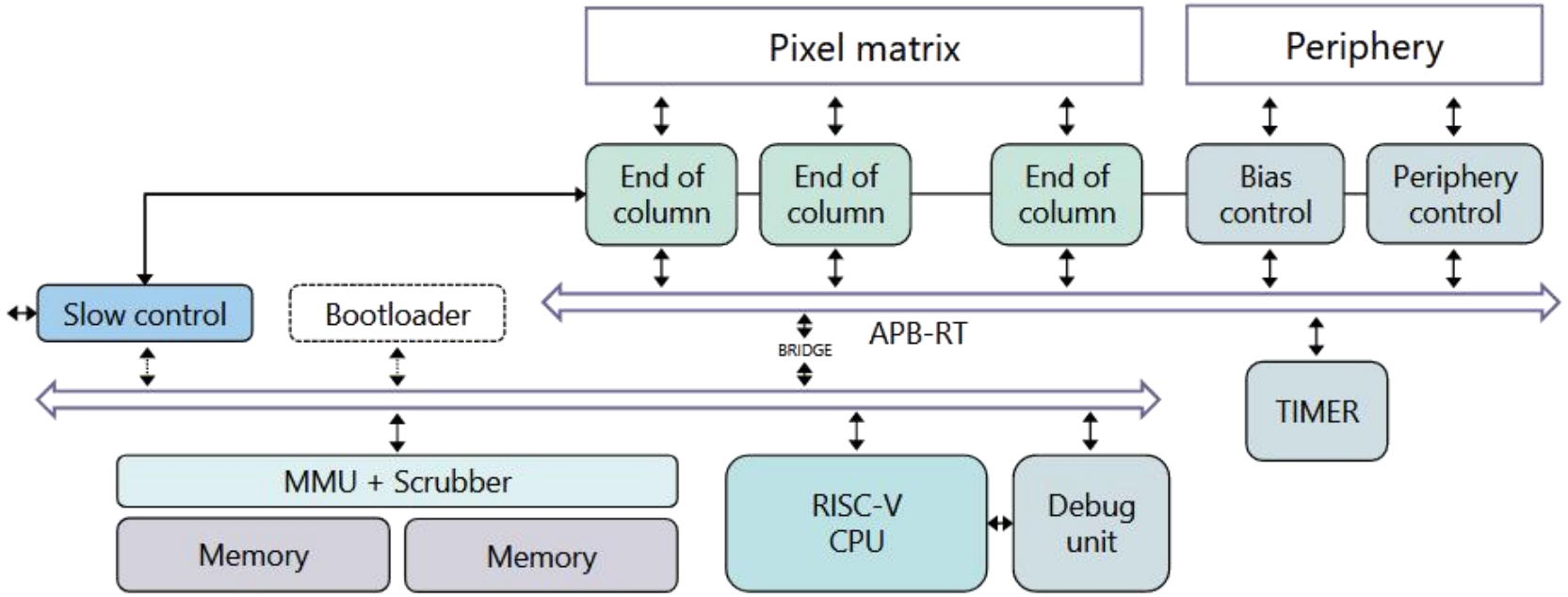
专用集成电路的设计版图

项目后续：研究千兆级数据收发器设计和高带宽片内互连技术，以突破现有系统在高速数据传输方面的限制

cite:10.1088/1748-0221/18/01/C01030

Risc-V SoC实例3:径迹探测器ASIC--Picopix

设计服务于：LHCb VELO探测器、抗辐照监测系统、束流监测、X射线和中子成像、质谱、X射线光子相关光谱等。



探测器指标：

- 实现<30ps RMS时间分辨率
- 输出带宽：102.4Gb/s

Risc-V功能：

- 日常监测和控制操作的自动化
- 偏压监测
- 片上标定
- 随时对其进行重新编程，使其非常灵活地适用于新的应用

该工作后续研究计划：

1、SoCMake工具链功能扩展（用户友好性提升）
2、IP库扩充（计划新增PCIe Gen3/4、高速SerDes等接口IP）
3、故障注入验证框架开发（基于UVM方法论）
4、实现异构加速架构（Risc-V+专用DSP）
5、建立HEP社区设计规范标准

cite:https://indico.cern.ch/event/1436991/contributions/6046661/attachments/2893748/5131661/2024.09.09_DRD7_SOC.pdf

Risc-V SoC实例4：xTern三元神经网络加速推理

1. 研究背景与动机

- 边缘AI的挑战：物联网设备受限于内存、算力和功耗，传统深度神经网络（DNN）难以直接部署。量化技术通过降低数据位宽减少资源需求，其中三元神经网络（TNN）在精度和能效间展现出比二元网络（BNN）更优的权衡。
- 现有问题：TNN的高效执行依赖专用加速器，但此类硬件占用面积较大，不适用于低成本边缘设备。现有RISC-V ISA缺乏对三元运算的原生支持，导致软件实现效率低下。

TABLE I
Hardware figures of merit for the xTern system and the baseline implementing only XpulpNN.

		XPulpNN	xTern (this work)
Core	A_{core} [a]	163.5 kGE	168.4 kGE (+3.0 %)
	$P_{MM,8b}$ [b]	4.1 mW	4.2 mW(+2.4 %)
	$P_{Conv,LP}$ [c]	4.0 mW	4.3 mW (+7.8 %)
Cluster	A_{clus} [a]	3.29 MGE	3.32 MGE (+0.9 %)
	Density	70.2 %	71.0 % (+0.8 pp.)
	f_{clk} [b]	500 MHz (376 MHz)	500 MHz (389 MHz)
	$P_{MM,8b}$ [c]	58.1 mW	58.9 mW(+1.4 %)
	$P_{Conv,LP}$ [c]	57.7 mW	60.7 mW (+5.2 %)
	$\text{Eff}_{Conv,LP}$ [c]	383.9 GOp/J	603.3 GOp/J (+**57.1 %**)

[a] Obtained from synthesized netlists. One gate equivalent (GE) in 22 nm FDX is 0.199 µm², the size of a NAND2 gate.
[b] The first number is obtained at HV operating conditions ($V_{DD} = 0.72$ V, $T = 25$ °C), the number in parentheses is obtained at LV operating conditions ($V_{DD} = 0.65$ V, $T = 25$ °C).
[c] Evaluated at LV operating conditions.

- parallel ultra-low power (PULP) 集成8个RI5CY处理器核心，单程序多数据（SPMD）编程模型
- 拓展3条指令乘加指令（MADD）、逐元素比较指令、阈值压缩指令（thrc）

相比于从头设计一个专用TNN硬件加速器，具有更好的灵活性、更低的集成开销和对现有生态的兼容性。

cite：arXiv:2405.19065v1 [cs.AR] 29 May 2024

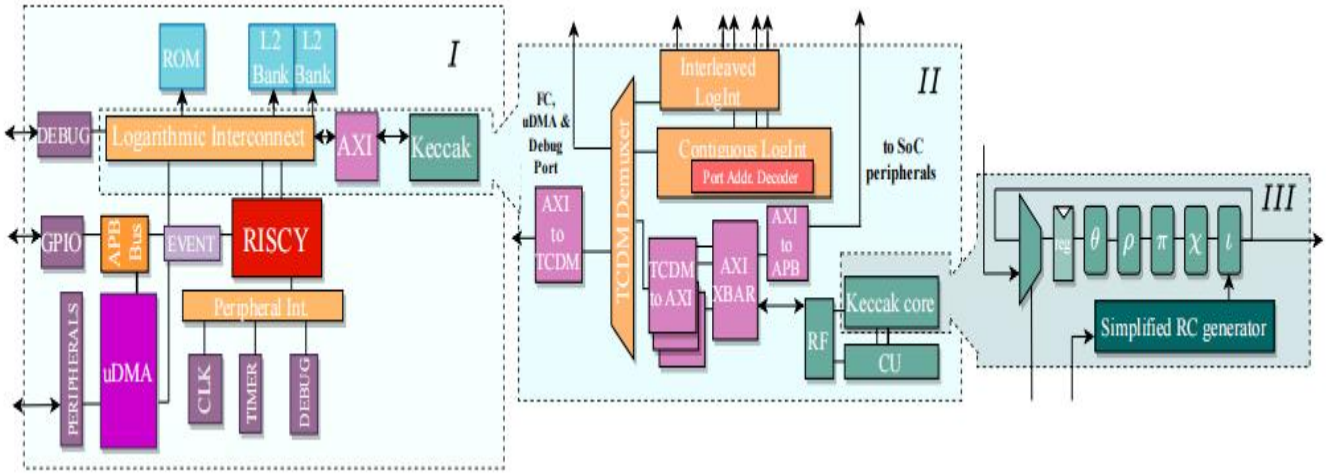
Risc-V SoC实例5：密码算法专用加速器

1、基于开源的RISC-V PULPissimo SoC（RI5CY核心），集成Keccak硬件加速器，大幅提高速度

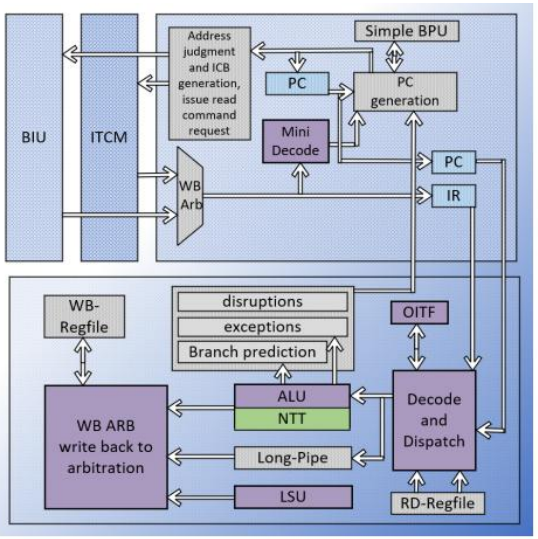
Table 2: Speed-up factor comparison (keygen/enc/dec)

	Kyber-512	Kyber-768	Kyber-1024
[5]	1.72/1.88/2.29	1.72/1.81/2.13	1.69/1.75/2.01
Our Work	2.79/2.60/1.97	2.67/2.66/2.08	2.76/2.71/2.20

cite:doi/10.1145/3587135.3591432



2、基于蜂鸟E203，提升后量子密码算法Kyber中核心运算模块Number-Theoretic Transform（NTT）的执行效率



额外添加指令ibutterfly、ntt/intt，并在硬件上扩展解码单元、调度单元、计算单元

相比于参考文献，在新增647 LUTs、8.6%的功耗下，执行时间从17,216到11,043周期

cite:10.1145/3587135.3591432

3、基于PULP开源的64位RISCV核Araine，在通用处理器架构的基础上根据密码处理特征进行优化,实现了高能效的RISCV密码专用指令处理器

表 5 密码处理器能效对比分析(Mbps/mW)

	文献[17]	文献[18]	密码专用处理器
AES	9.39	20.74	5.30
DES	2.91	6.91	3.03
SHA256	3.55	0.26	5.56
SNOWV	–	–	1.61
MD5	3.30	0.64	35.16

cite:10.11999/JEIT210004

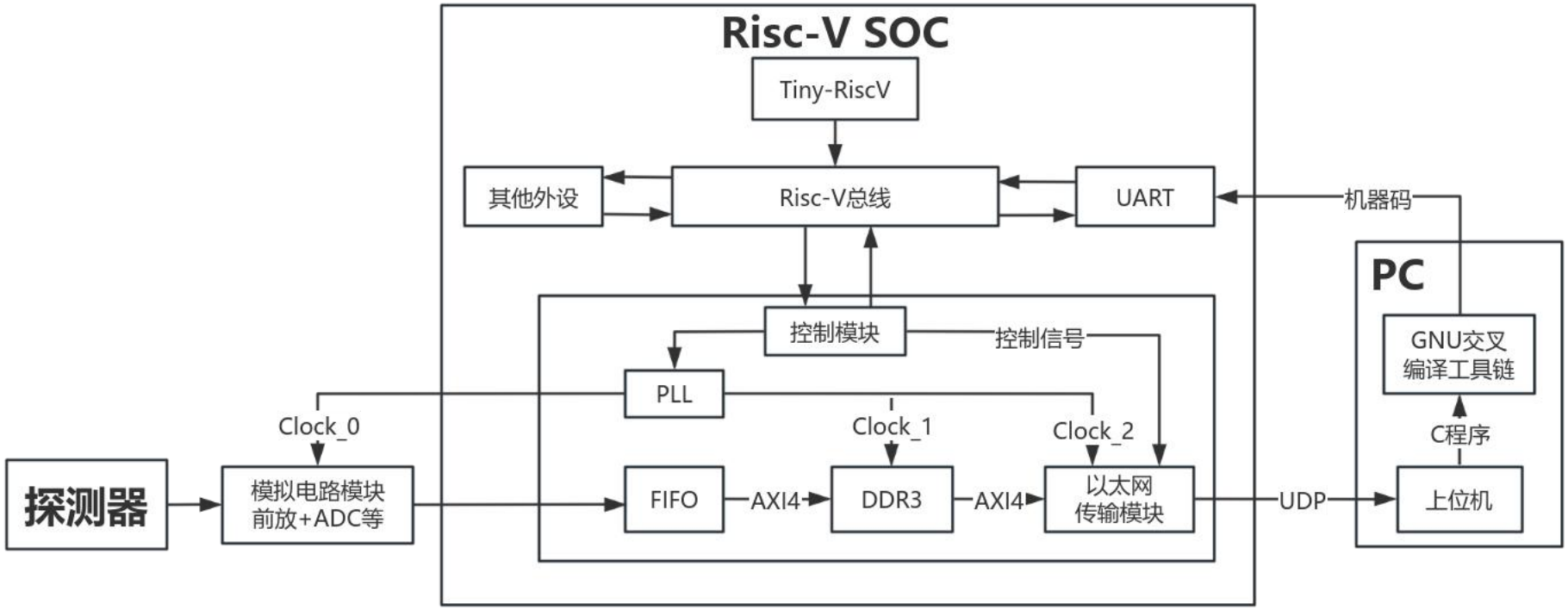
一、System-on-Chip简介

二、当前研究工作

- 基于Risc-V的读出测试板

基于Risc-V的读出测试板

在FPGA开发板上搭载Risc-V软核，用于硅探测器数据读出和测试，并熟悉Risc-V及Soc开发的相关内容



读出测试板功能框架示意图

读出测试板已完成工作

- ✓ Tiny-Riscv上板验证
- ✓ GPIO拓展
- ✓ 动态PLL时钟
- ✓ DDR3
- ✓ UDP
- ✓ 上位机

------初步实现读出功能------

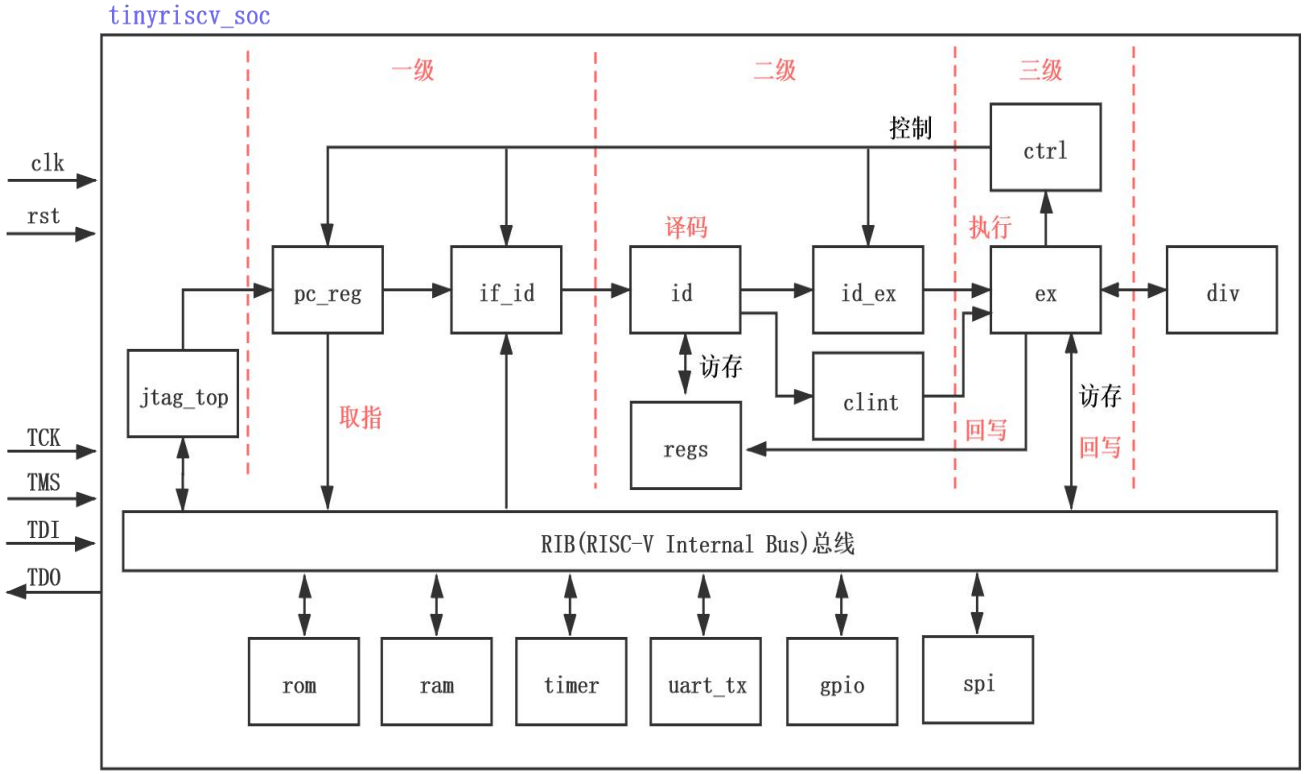
后续计划

- ☐ 完成硅探测器的读出测试
- ☐ 更换到PULP综合平台（计划将Risc-V核心从Tiny换到Ibex）
- ☐ 完善模拟电路模块
- ☐ FMC
- ☐ Linux
- ☐ .....

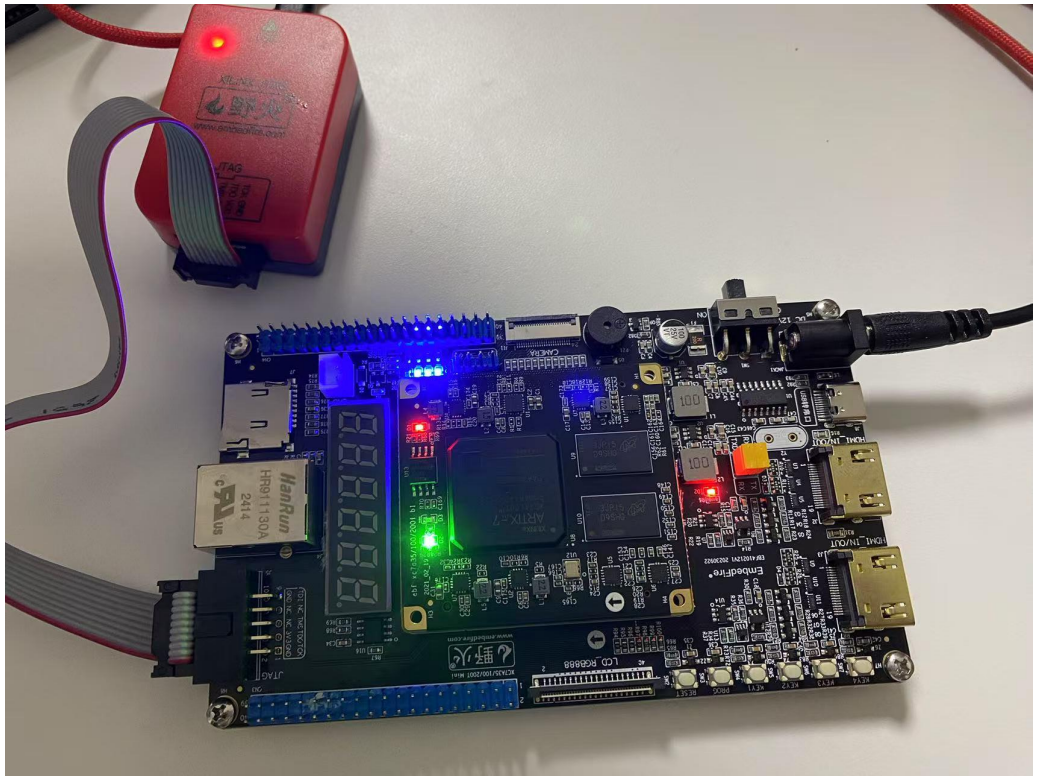
读出测试板：软核—Tiny Riscv

支持RV32IM指令集；
采用三级流水线
支持JTAG，可以通过openocd读写内存(在线更新程序)；
支持中断、FreeRTOS；
支持通过UART更新程序；

Tiny-Riscv：文档详细，入门友好



Tiny-Riscv架构



Xlinx Arty7开发板

Tiny-RiscV：https://gitee.com/liangkangnan/tinyriscv.git

读出测试板：GPIO拓展

Tiny-Riscv仅配置有2个GPIO口，需要对硬件代码和程序进行修改，将其拓展到足够的数量，目前已完成对硬件的修改，将接口数量参数化

硬件部分拓展

```
// GPIO模块
module gpio(

    input wire clk,
    input wire rst,

    input wire we_i,
    input wire[31:0] addr_i,
    input wire[31:0] data_i,

    output reg[31:0] data_o,

    input wire[1:0] io_pin_i,
    output wire[31:0] reg_ctrl,
    output wire[31:0] reg_data

    );


    // GPIO控制寄存器
    localparam GPIO_CTRL = 4'h0;
    // GPIO数据寄存器
    localparam GPIO_DATA = 4'h4;

    // 每2位控制1个IO的模式，最多支持16个IO
    // 0: 高阻, 1: 输出, 2: 输入
    reg[31:0] gpio_ctrl;
    // 输入输出数据
    reg[31:0] gpio_data;


    assign reg_ctrl = gpio_ctrl;
    assign reg_data = gpio_data;


    // 写寄存器
    always @ (posedge clk) begin
        if (rst == 1'b0) begin
            gpio_data <= 32'h0;
            gpio_ctrl <= 32'h0;
        end else begin
            if (we_i == 1'b1) begin
                case (addr_i[3:0])
                    GPIO_CTRL: begin
                        gpio_ctrl <= data_i;
                    end
                    GPIO_DATA: begin
                        gpio_data <= data_i;
                    end
                endcase
            end else begin
                if (gpio_ctrl[1:0] == 2'b10) begin
                    gpio_data[0] <= io_pin_i[0];
                end
                if (gpio_ctrl[3:2] == 2'b10) begin
```

```
module gpio #(
    parameter GPIO_NUM = 16  // 支持1-128个GPIO
)(
    input wire clk,
    input wire rst,

    // 总线接口
    input wire we_i,
    input wire[31:0] addr_i,
    input wire[31:0] data_i,
    output reg[31:0] data_o,

    // GPIO引脚
    input wire [GPIO_NUM-1:0] io_pin_i,
    output wire [((GPIO_NUM+15)/16)*32-1:0] reg_ctrl,  // 控制寄存器组
    output wire [((GPIO_NUM+31)/32)*32-1:0] reg_data   // 数据寄存器组
);

// 自动计算寄存器数量
localparam CTRL_REG_NUM = (GPIO_NUM + 15) / 16;  // 每个控制寄存器管理16个GPIO
localparam DATA_REG_NUM = (GPIO_NUM + 31) / 32;  // 每个数据寄存器管理32个GPIO

// 寄存器定义
reg [31:0] gpio_ctrl [0:CTRL_REG_NUM-1];
reg [31:0] gpio_data [0:DATA_REG_NUM-1];

// 地址映射参数
localparam CTRL_BASE = 8'h00;      // 控制寄存器基地址
localparam DATA_BASE = 8'h40;      // 数据寄存器基地址
localparam REG_ADDR_WIDTH = 8;     // 地址总线使用8位

// 寄存器输出扁平化
generate
    genvar i;
    for(i=0; i<CTRL_REG_NUM; i=i+1) begin
        assign reg_ctrl[i*32 +: 32] = gpio_ctrl[i];
    end
    for(i=0; i<DATA_REG_NUM; i=i+1) begin
        assign reg_data[i*32 +: 32] = gpio_data[i];
    end
endgenerate

// 写寄存器逻辑
always @(posedge clk) begin
    if (!rst) begin
        for(integer j=0; j<CTRL_REG_NUM; j=j+1)
            gpio_ctrl[j] <= 32'h0;
        for(integer j=0; j<DATA_REG_NUM; j=j+1)
            gpio_data[j] <= 32'h0;
    end
    else begin
        if (we_i) begin
            // 控制寄存器写
            if(addr_i[REG_ADDR_WIDTH-1:0] >= CTRL_BASE &&
               addr_i[REG_ADDR_WIDTH-1:0] < (CTRL_BASE + CTRL_REG_NUM*4)) begin
```

软件部分拓展

```
#include <stdint.h>
#include "../include/utils.h"

#define GPIO_BASE       (0x40000000)
#define GPIO_CTRL       (GPIO_BASE + (0x00))
#define GPIO_DATA       (GPIO_BASE + (0x04))

#define GPIO_REG(addr) (*((volatile uint32_t *)addr))

int main()
{
    GPIO_REG(GPIO_CTRL) |= 0x1;       // gpio0输出模式
    GPIO_REG(GPIO_CTRL) |= 0x1 << 3;  // gpio1输入模式

    while (1) {
        // 如果GPIO1输入高
        if (GPIO_REG(GPIO_DATA) & 0x2)
            GPIO_REG(GPIO_DATA) |= 0x1;  // GPIO0输出高
        // 如果GPIO1输入低
        else
            GPIO_REG(GPIO_DATA) &= ~0x1; // GPIO0输出低
    }
}
```

```
// 新地址映射方案（假设基地址保持0x40000000不变）
#define GPIO_BASE           0x40000000

// 寄存器组偏移定义
#define GPIO_CTRL_OFFSET    0x00  // 控制寄存器组起始偏移
#define GPIO_DATA_OFFSET    0x40  // 数据寄存器组起始偏移
#define REG_SIZE            0x04  // 每个寄存器占4字节

// 寄存器访问宏
#define GPIO_CTRL_REG(n)    (GPIO_BASE + GPIO_CTRL_OFFSET + ((n) * REG_SIZE))
#define GPIO_DATA_REG(n)    (GPIO_BASE + GPIO_DATA_OFFSET + ((n) * REG_SIZE))

// GPIO编号到寄存器的转换宏
#define GPIO_CTRL_INDEX(pin)  ((pin) / 16)  // 每16个GPIO一个控制寄存器
#define GPIO_DATA_INDEX(pin)  ((pin) / 32)  // 每32个GPIO一个数据寄存器
#define GPIO_CTRL_POS(pin)    (((pin) % 16) * 2)  // 控制位起始位置
#define GPIO_DATA_POS(pin)    ((pin) % 32)        // 数据位位置

// 模式定义
typedef enum {
    GPIO_MODE_INPUT  = 0x2,  // 10b
    GPIO_MODE_OUTPUT = 0x1,  // 01b
    GPIO_MODE_HIZ    = 0x0   // 00b
} GpioMode;

// 寄存器访问函数
static inline void gpio_set_mode(uint32_t pin, GpioMode mode) {
    volatile uint32_t *ctrl_reg = (volatile uint32_t*)GPIO_CTRL_REG(GPIO_CTRL_INDEX(pin));
    uint32_t shift = GPIO_CTRL_POS(pin);
    *ctrl_reg = (*ctrl_reg & ~(0x3 << shift)) | (mode << shift);
}

static inline void gpio_write(uint32_t pin, uint8_t value) {
    volatile uint32_t *data_reg = (volatile uint32_t*)GPIO_DATA_REG(GPIO_DATA_INDEX(pin));
    uint32_t shift = GPIO_DATA_POS(pin);
    *data_reg = (*data_reg & ~(0x1 << shift)) | ((value & 0x1) << shift);
}

static inline uint8_t gpio_read(uint32_t pin) {
    volatile uint32_t *data_reg = (volatile uint32_t*)GPIO_DATA_REG(GPIO_DATA_INDEX(pin));
    return (*data_reg >> GPIO_DATA_POS(pin)) & 0x1;
}

int main() {

    for(int i = 1; i <= 29; i=i+2){
        gpio_set_mode(i, GPIO_MODE_OUTPUT);  // 必须先配置模式
    }

    // 配置GPIO1为输入模式
    gpio_set_mode(1, GPIO_MODE_INPUT);

    while (1) {
        // 读取GPIO1状态
        if (gpio_read(1)) {
```

读出测试板：PLL动态配置时钟

顶层模块+总线模块

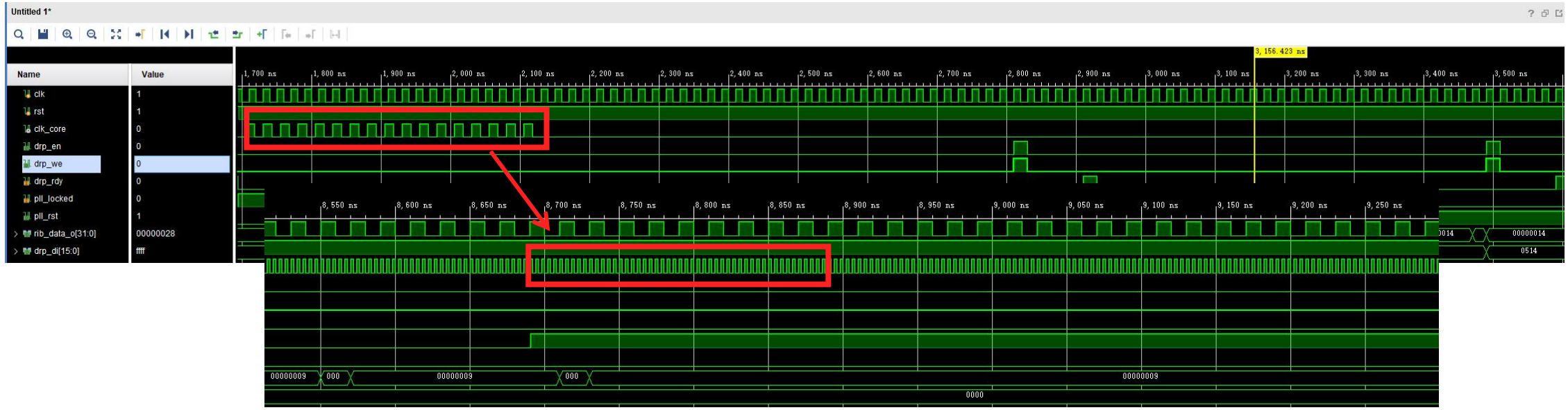
```
wire [6:0]  drp_addr;    // DRP地址总线，7位
wire        drp_en;      // DRP使能
wire [15:0] drp_di;      // DRP写数据
wire        drp_we;      // DRP写使能
wire [15:0] drp_do;      // DRP读数据
wire        drp_rdy;     // DRP操作完成
wire        pll_reset;   // PLL复位
wire        pll_locked; // PLL锁定状态

// 实例化PLL控制器
pll_rib_slave pll_ctrl (
    .rib_clk     (clk),
    .rib_rst     (rst),
    .rib_addr    (s6_addr_o),   // 连接到RIB的s6_addr_o
    .rib_data_i  (s6_data_o),   // 连接到RIB的s6_data_o
    .rib_data_o  (s6_data_i),   // 连接到RIB的s6_data_i
    .rib_we      (s6_we_o),     // 连接到RIB的s6_we_o
    .drp_addr    (drp_addr),
    .drp_en      (drp_en),
    .drp_di      (drp_di),
    .drp_we      (drp_we),
    .drp_do      (drp_do),
    .drp_rdy     (drp_rdy),
    .pll_rst     (pll_reset),    // PLL复位信号
    .pll_locked  (pll_locked)
);

// 实例化Xilinx PLL IP核
clk_wiz_0 pll_inst (
    .clk_out1 (clk_core),       // 输出的核心时钟
    .daddr    (drp_addr),       // DRP地址
    .dclk     (clk),            // DRP时钟
    .den      (drp_en),         // DRP使能
    .din      (drp_di),         // DRP输入数据
    .dout     (drp_do),         // DRP输出数据
    .drdy     (drp_rdy),        // DRP就绪
    .dwe      (drp_we),         // DRP写使能
    .reset    (pll_reset),        // input reset
    .locked   (pll_locked),   // PLL锁定状态
    .clk_in1  (clk)             // 输入时钟
);
```

PLL控制模块

```
// 写逻辑: 处理 RIB 总线写操作
always @(posedge rib_clk or posedge rib_rst) begin
    if (!rib_rst) begin
        drp_addr      <= 7'h0;
        drp_di        <= 16'h0;
        drp_we        <= 1'b0;
        drp_en        <= 1'b0;
        pll_rst       <= 1'b0;
    end else begin
        // 默认值: DRP 使能信号为单周期脉冲
        drp_en <= 1'b0;
        drp_we <= 1'b0;

        if (rib_we) begin
            case (rib_addr[4:0])
                4'h0: begin  // 0x60000000: 设置 DRP 地址
                    drp_addr <= rib_data_i[6:0];
                end
                4'h4: begin  // 0x60000004: 触发 DRP 写操作
                    drp_di   <= rib_data_i[15:0];
                    drp_we   <= 1'b1;
                    drp_en   <= 1'b1;  // 单周期脉冲
                end
                4'h8: begin  // 0x60000008: 触发 DRP 读操作 (可选)
                    drp_we   <= 1'b0;
                    drp_en   <= 1'b1;  // 单周期脉冲
                end
                4'hC: begin  // 0x6000000C: 控制 PLL 复位
                    pll_rst  <= rib_data_i[0];
                end
            endcase
        end
    end
end

// 读逻辑: 返回 DRP 数据和状态
always @(*) begin
    case (rib_addr[4:0])
        4'h0: rib_data_o = {25'b0, drp_addr};     // 读取 DRP 地址
        4'h4: rib_data_o = {16'b0, drp_do};       // 读取 DRP 数据
        4'h8: rib_data_o = {30'b0, drp_rdy, pll_locked}; // 状态: bit[1]=drp_rdy, bit[0]=pll_locked
        4'hC: rib_data_o = {31'b0, pll_rst};      // 读取 PLL 复位状态
        default: rib_data_o = 32'h0;
    endcase
end

endmodule
```

C程序

```
//-----------------------------------------------
// 主函数: 动态配置 PLL 频率
//-----------------------------------------------
void pll_dynamic_config(uint16_t M, uint16_t D, uint16_t O) {
    // Step 1: 拉起 PLL 复位
    pll_reset(1);

    // Step 2: 写入 PowerReg (DRP地址0x28) 全1
    drp_write(0x28, 0xFFFF);  // 正确操作: 先设地址0x28, 再写数据0xFF
    // Step 5: 配置 DIVCLK Register (0x16)
    drp_write(0x16, 0x01);      // 写入分频值 D
    // Step 3: 配置 CLKFBOUT Register 1 (0x14)
    drp_write(0x14, 0x0514);       // 写入倍频值 M
    // Step 4: 配置 CLKFBOUT Register 2 (0x15)
    drp_write(0x15, 0x0000); // 相位/延时配置 (根据需求调整)
    drp_write(0x08, 0x0104);     // 分频值 O (高8位可为相位)
    drp_write(0x09, 0x0000); // 相位/边沿配置 (根据需求调整)

    // Step 8: 释放 PLL 复位
    pll_reset(0);

    // Step 9: 等待 PLL 锁定
    while ((PLL_REG_READ(REG_STATUS) & STATUS_PLL_LOCKED) == 0);
}

//-----------------------------------------------
// 示例: 将 PLL 配置为 M=64, D=1, O=8
//-----------------------------------------------
int main() {
    // 假设 M/D/O 已根据 DRP 格式转换 (需参考 PLL 手册)
    uint16_t M = 0x0040;  // CLKFBOUT1 寄存器值 (对应M=64)
    uint16_t D = 0x0001;  // DIVCLK 寄存器值 (对应D=1)
    uint16_t O = 0x0008;  // ClkReg1 寄存器值 (对应O=8)

    // 执行配置
    pll_dynamic_config(M, D, O);

    return 0;
}
```



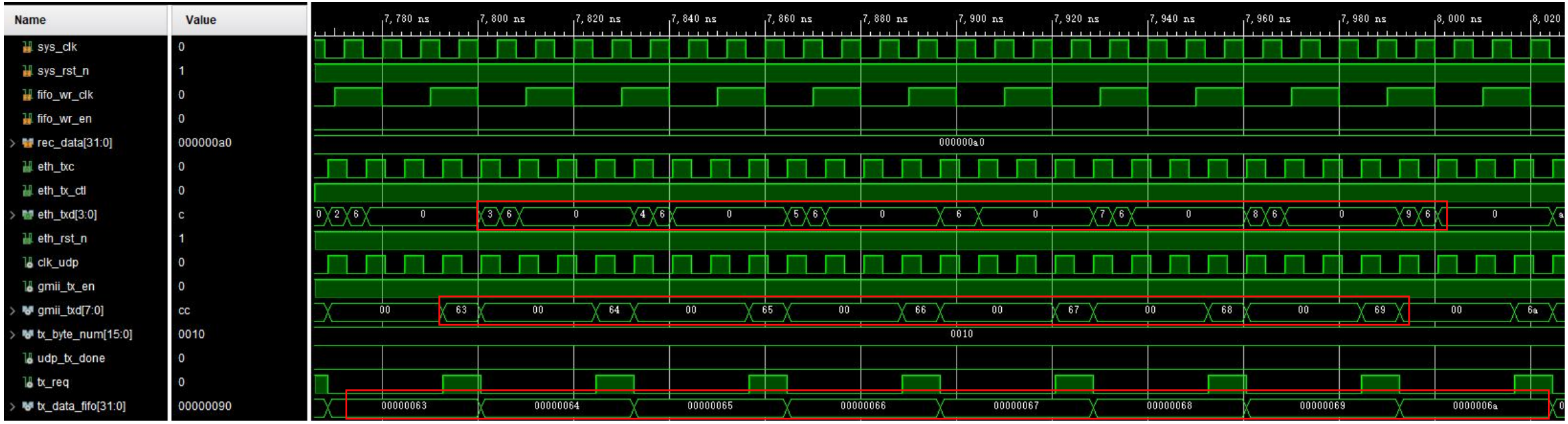
读出测试板：UDP传输模块

- 通过GMII/RGMII千兆以太网接口实现高速通信
  - GMII：单边沿采样，信号线较多
  - RGMII：双边沿采样，信号线较少，目前选用此方案

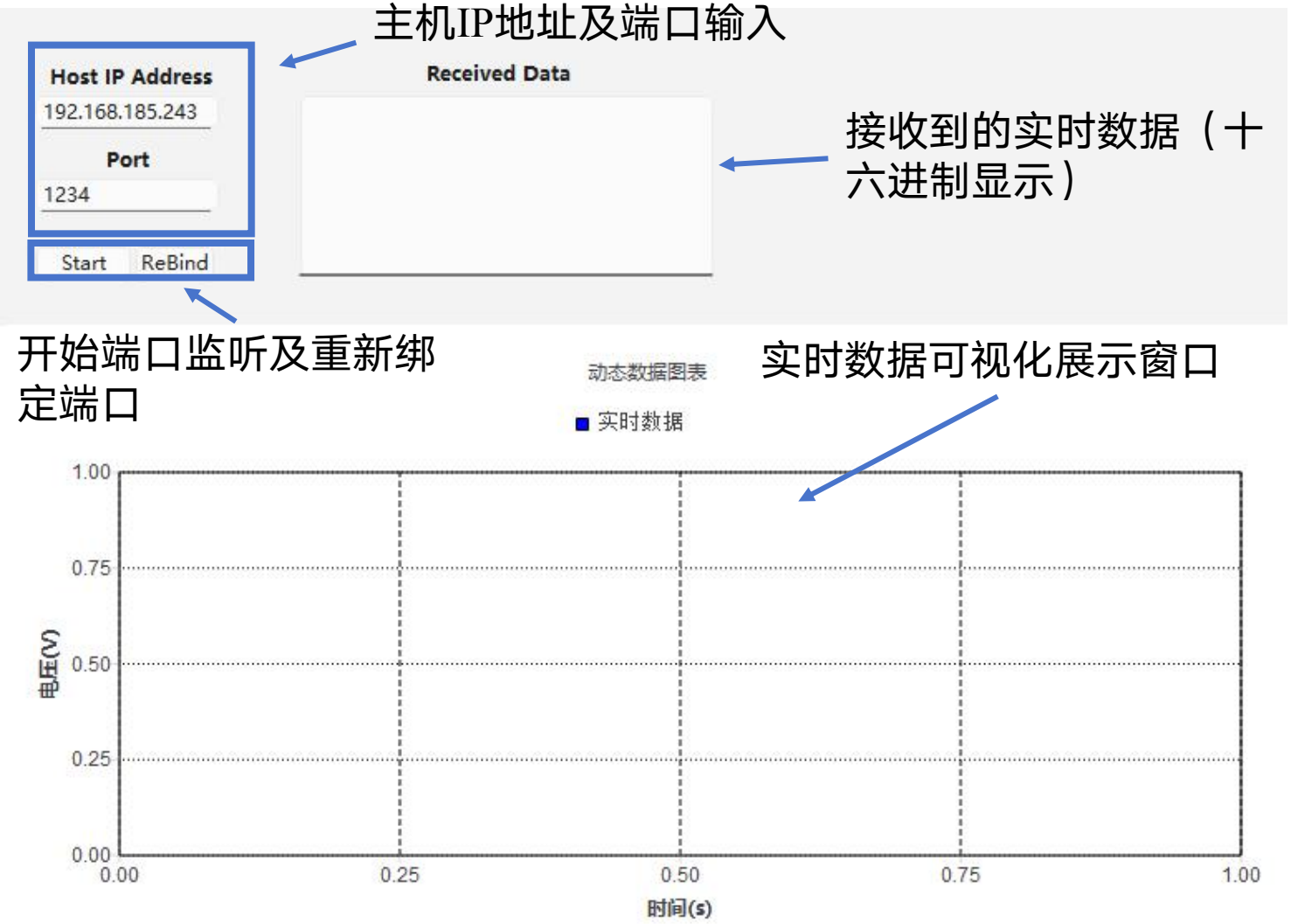
- 将数据打包后，按照IEEE标准进行传输。目前需要做的就是将按照标准进行打包后，控制开发板上的PHY（物理层）芯片对数据进行输出。

- 选择合适的协议进行传输，在网络层，使用ipv4完成主机间的路由选择和传递。在传输层，主要实现主机应用程序之间端到端的服务，主要有TCP和UDP两种协议可供选择，目前选用的是UDP协议

- 模块仿真结果

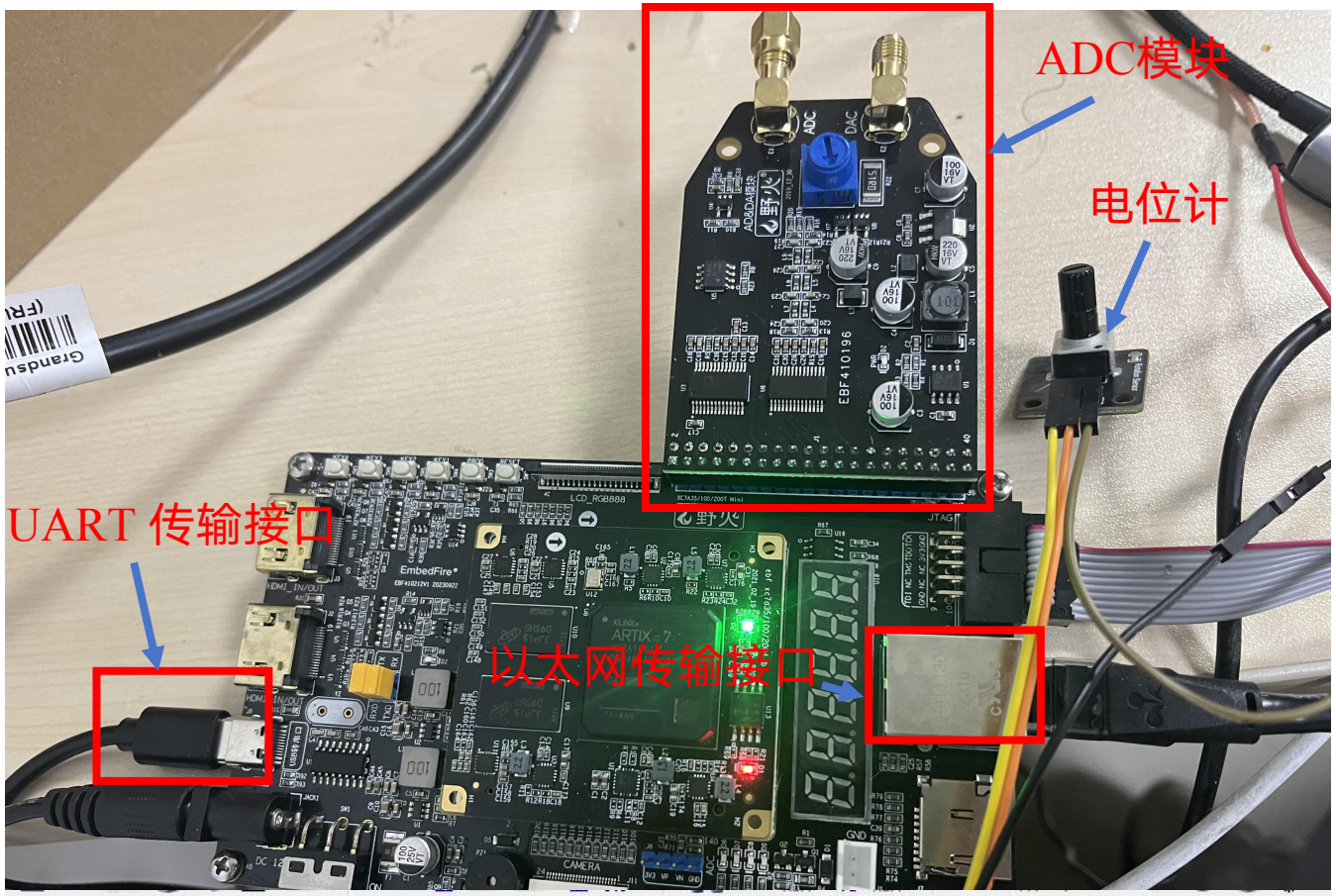


读出测试板：上位机



- 输入主机IP地址及端口后，将自动绑定相应端口。成功后输出对应的IP地址及端口，绑定失败显示“UPD socket 绑定失败”
- 如果绑定失败或者用户想要重新绑定，输入对应IP地址及端口后，点击ReBind按钮会自动解除绑定并重新开始绑定
- 绑定成功后，自动开始监听对应端口，并将二进制数据保存在指定的bin文件中
- 将二进制数据按需求显示在可视化窗口中

读出测试板：上板验证



- 使用AD9280 读取固定电压值，通过改变电位计，观察信号变化测试能否正常工作
- FIFO读取并写入ADC给出的数字信号
- 以太网传输模块读取存储在FIFO中的信号，并将信号传输到电脑上，频率125M
- 电脑端的上位机接收以太网传输模块输出的信号，并将二进制数据保存到bin文件中

开始运行后，使用C程序控制通过UART输出调试结果

```
── System Initialization ──

── Configuring Network Settings ──
Board IP set to: 192.168.185.111 (0xC0A8B96F)
Destination IP set to: 192.168.185.243 (0xC0A8B9F3)
Read back Board IP: 0xC0A8B96F
Read back Dest IP:  0xC0A8B9F3

── Enabling Data Acquisition and UDP Transmission ──
UDP Config Reg (0x10) written with: 0x00030400
UDP Config Reg (0x10) read back: 0x00030400
  TX Data Num: 1024
  UDP TX Enable: 1
  FIFO WR Enable: 1
```

上位机保存的数据文件

received_data.bin

[unregistered]
received_data.bin
D:\QTProject\UDPReceive\build\

文件大小: 13.7 MB
14,320,640 字节

缺省编辑模式
状态: 原始的
撤消级数: 0
反向撤消: 无

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13891488	00	00	00	9B	00	00	00	9C	00	00	00	9B	00	00	00	9B
13891504	00	00	00	9B	00	00	00	9B	00	00	00	9B	00	00	00	9C
13891520	00	00	00	9B	00	00	00	9C	00	00	00	9B	00	00	00	9B
13891536	00	00	00	9B	00	00	00	9B	00	00	00	9B	00	00	00	9B
13891552	00	00	00	9B	00	00	00	9B	00	00	00	9B	00	00	00	9B
13891568	00	00	00	9B	00	00	00	9B	00	00	00	9B	00	00	00	9B
13891584	00	00	00	9C	00	00	00	9C	00	00	00	9C	00	00	00	9C
13891600	00	00	00	9C	00	00	00	9C	00	00	00	9C	00	00	00	9C
13891616	00	00	00	9C	00	00	00	9C	00	00	00	9C	00	00	00	9C
13891632	00	00	00	9C	00	00	00	9C	00	00	00	9C	00	00	00	9C
13891648	00	00	00	9C	00	00	00	9C	00	00	00	9C	00	00	00	9C

后续计划：

- 使用真实硅探测器完成读出测试
- 更换到PULP综合平台
- 完善搭载linux系统、FMC等其他功能

一、System-on-Chip简介

二、当前研究工作

- 基于Risc-V的读出测试板
- 集成于ASIC的Risc-V 微处理器（MCU）研发

RISC-V soft-core Investigation Report

Investigation Objectives: Select the right RISC-V soft core for HEP experiments or possible data processing needs

Nine common, excellent and interesting RISC-V core:

- **PicoRV32**
- **SERV**
- **VexRiscv**
- **Ibex**
- **Hummingbird E203**
- **SweRV EH1 / EL2**
- **CVA6 (Ariane)**
- **Rocket Core**
- **Berkeley BOOM**

主要用于超低功耗控制器

中等性能和功耗，可用于数据处理

可用于高性能计算

Part 1

For all the soft core, we investigated:

- **Architecture**
- **FPGA Performance & Resources**
- **Use in HEP**
- **Feature & Use Cases**
- **…**

It also shows their **comparison** on:

- **Architectural Features of RISC-V Soft Cores** (pipeline design, ISA support, special features)
- **FPGA Resource Utilization and Performance of RISC-V Cores** (example targets: Xilinx 7-series and Lattice iCE40, with typical configurations)

Part 2

And the **Selection Guidance**.

Part 3

Comparison Table: SERV vs. PicoRV32

Metric	SERV	PicoRV32
LUT Usage	~125 LUTs (Artix-7) ~198 LUTs (Lattice iCE40)	~750 LUTs (minimal config) ~1070 LUTs (typical config)
FFs (Flip-Flops)	~164 FFs	~573 FFs (Xilinx 7-series)
DSP Usage	0 (no hardware multiplier)	Optional (1 DSP if M extension is enabled)
BRAM	0	0 (by default); external RAM required
Clock Frequency	High (>200 MHz), due to no combinational datapath	Medium-High (100–200 MHz)
Area Class	"UART-level" (smaller than a SPI controller)	Full-featured MCU-class core

- **SERV**: an ultra-minimalist design, intended for scenarios with extremely limited logic area. Its ALU is only 1-bit wide, requiring 32 clock cycles to complete a single 32-bit operation.
- **PicoRV32**: a minimal MCU-class design capable of running real bare-metal programs and RTOS, offering higher IPC (though still lower than pipelined cores).

minimal logic usage for *simple control tasks*

a more practical soft core that can run C code and respond more quickly

集成于ASIC的Risc-V MCU：Pico

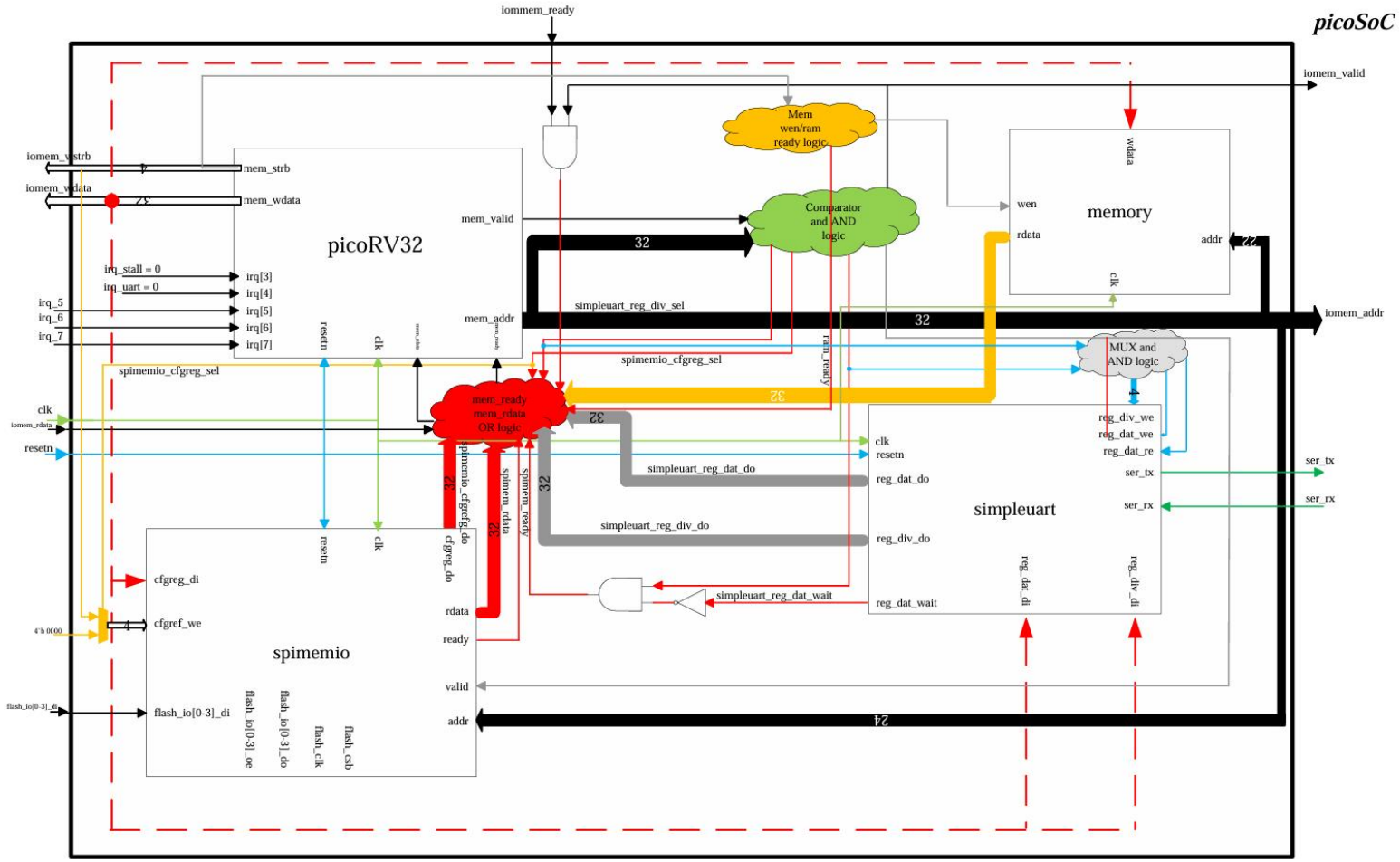
PicoSoC 是基于 PicoRV32 的最小系统级片上系统（SoC），PicoRV32是超轻量、可综合的开源 RISC-V 内核，具备高可配置性和极低资源占用，非常适合在 FPGA 和面积受限的 SoC 中实现灵活的嵌入式控制

Pico特点：

- 资源占用小：在 7 系列 Xilinx 架构中占用 750–2000 个查找表（LUTs）。
- 高频率：在 7 系列 Xilinx FPGA 上可实现 250–450 MHz 的最大频率（fmax）。
- 内存接口可选：支持 原生内存接口 或 AXI4-Lite 主设备接口。
- 可选中断支持：通过 简单自定义指令集架构（ISA） 实现中断请求（IRQ）功能。
- 可选协处理器接口：支持与协处理器的通信接口（如加速器或外设）。

当前进展：

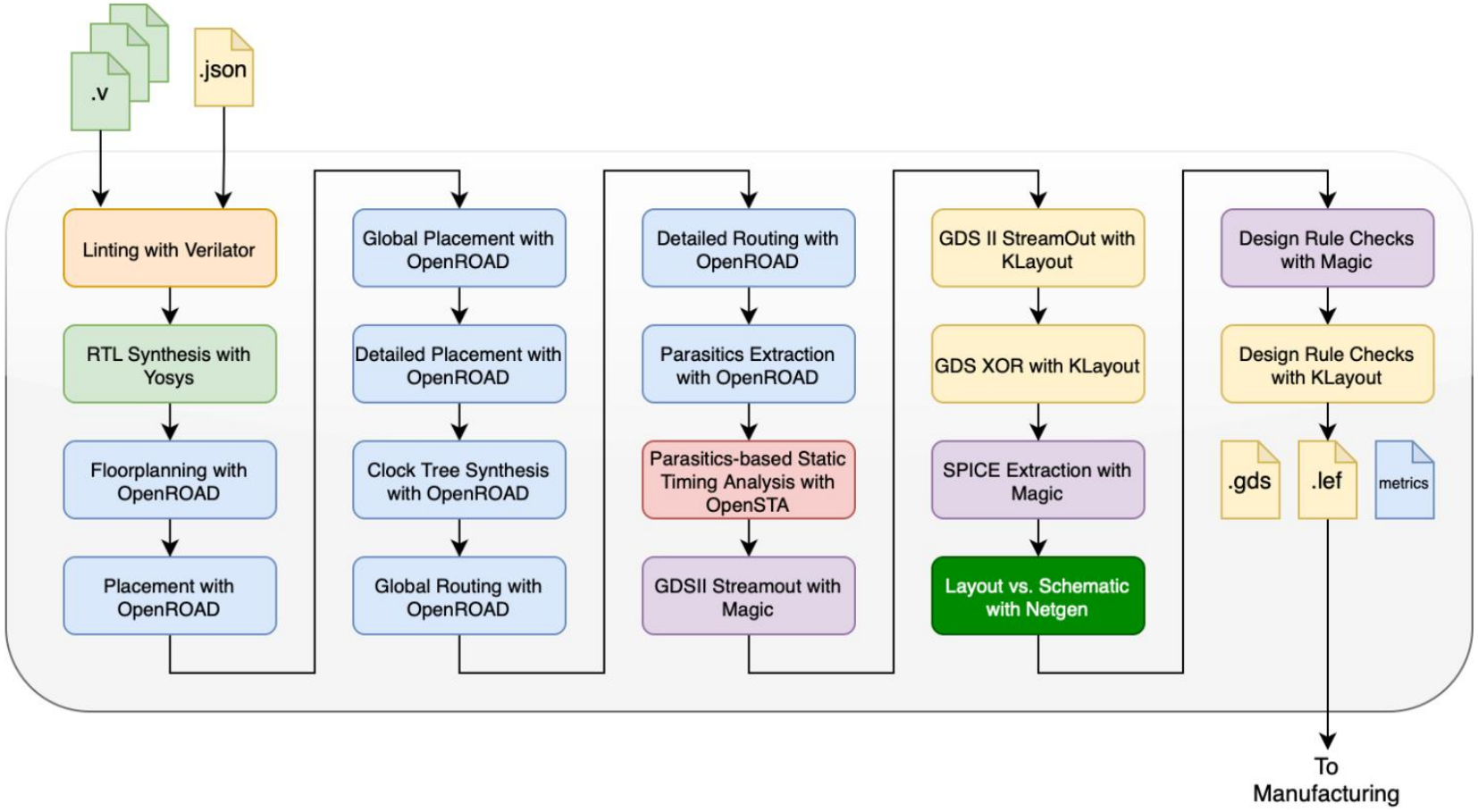
- ✓ Pico代码熟悉，根据源代码顶层模块实例化如右图所示
- ✓ Pico在FPGA上验证
- ☐ 数字IC设计流程准备



Pico：https://github.com/YosysHQ/picorv32

集成于ASIC的Risc-V MCU：数字IC开发工具—OpenLANE

- OpenLANE，一个可提供从RTL到GDSII的全自动流片流程的开源项目，它包含了几个重要的开源组件包括OpenROAD，Yosys，Magic，Netgen等负责流程中的各个部分
- 完成版图设计后，最终实现将CPU控制模块和硅探测器芯片结合



- 完全开源、透明度高、无供应商锁定
- “一键式”流程，用户只需提供RTL代码和一些配置参数，流程即可自动运行，减少了人工干预
- 结合开源PDKSkyWater PDK、GlobalFoundries Open PDK，社区活跃
- 对于非常大规模、高性能、高复杂度的尖端设计，其时序、面积、功耗可能仍无法与顶级的商业EDA工具媲美

秉持Risc-V的理念：开放、合作
欢迎一起合作、共同开发，包括以后共同的芯片研发

cite:https://github.com/The-OpenROAD-Project/OpenLane

一、System-on-Chip简介

二、当前研究工作

- 基于Risc-V的读出测试板
- 集成于ASIC的Risc-V 微处理器（MCU）研发

三、总结

总结

- 基本完成了读出测试板的功能，实现从ADC到二进制文件的流程，其他功能正在完善中
- 将Risc-V集成到硅探测器芯片中作为MCU，此工作处于前期规划中，包括：完善芯片功能需求调研、FPGA原型验证、芯片集成数字IC设计准备（目标：在硅探测器芯片上集成MCU模块）
- 从Chat-gpt3 到 DeepSeek 再到通义千问3、谷歌Gemini 2.5 pro，AI从自动代码补全、代码解读、生成，甚至是从事科研---AlphaEvolve，已充分证明AI作为新型生产力的核心价值，Risc-V架构开源、低功耗、可拓展（特别是RVV、Matrix等AI相关拓展），将基于Risc-V的AI与探测器结合，实现由“被动采集”转向“主动感知”，是一个有意思的方向：最终实现探测器芯片智能化

cite:AlphaEvolve: A coding agent for scientific and algorithmic discovery
Smart pixel：https://fastmachinelearning.org/smart-pixels/

back up

Not only RISCV Cores – many IPs for a Platform

RISC-V Cores

- RI5CY 32b
- Ibex 32b
- Snitch 32b
- Ariane + Ara 64b

Peripherals

- JTAG
- SPI
- UART
- I2S
- DMA
- GPIO

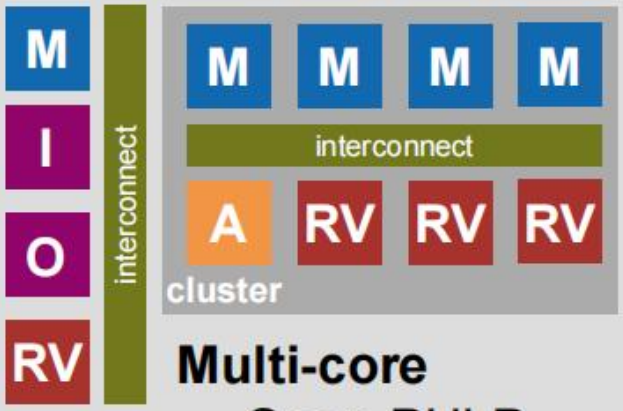
Interconnect

- Logarithmic interconnect
- APB – Peripheral Bus
- AXI4 – Interconnect

Platforms https://github.com/pulp-platform/

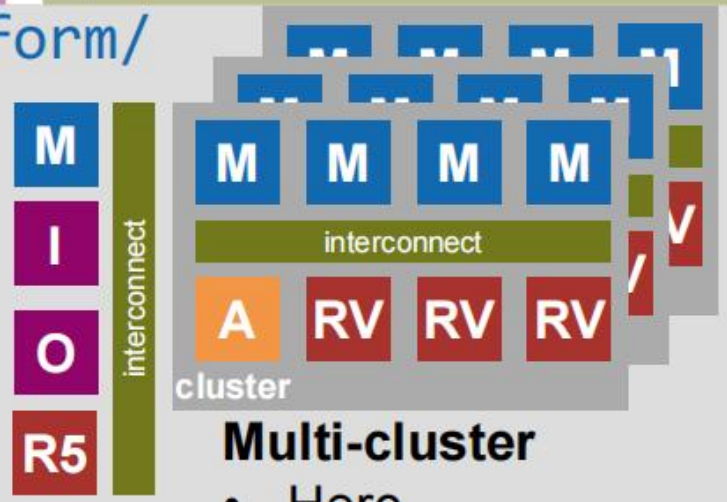
Single Core

- PULPino
- PULPissimo



Multi-core

- Open-PULP
- PULP-PM



Multi-cluster

- Hero
- Manticore

IOT → HPC

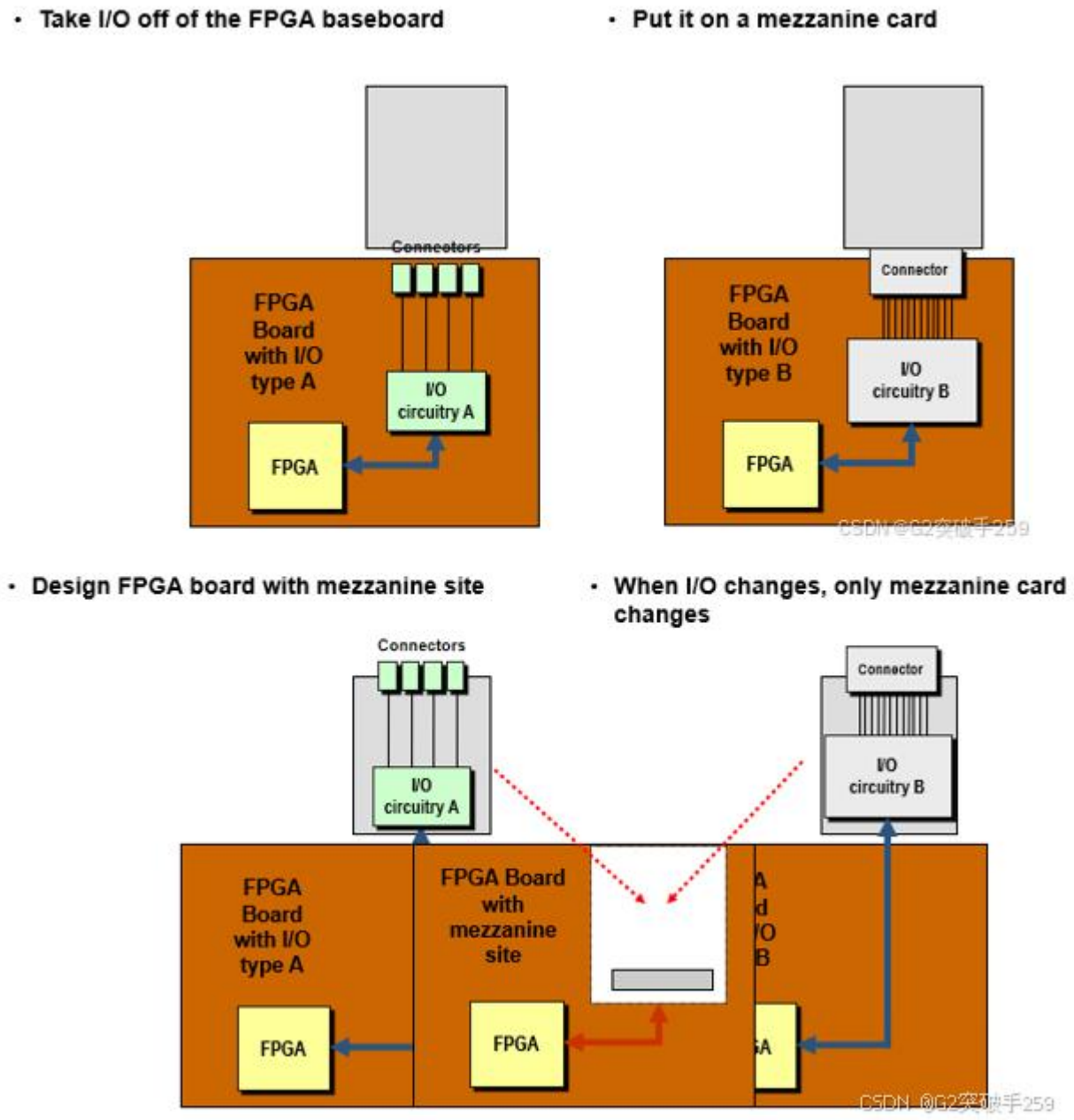
Accelerators

- HWCE (convolution)
- Neurostream (ML)
- HWCrypt (crypto)
- PULPO (1st ord. opt)

FMC（FPGA Mezzanine Card）

FMC接口允许用户通过一个标准化的方式将不同的硬件模块（例如高速ADC或DAC、传感器接口、通信模块等）连接到FPGA开发板上，从而极大地增强了FPGA系统的灵活性和可重用性。

使用FMC标准，开发者可以更容易地更换或升级附加硬件，而无需重新设计整个系统。这不仅节省了成本，也缩短了产品上市时间。



cite:https://blog.csdn.net/s1_mple/article/details/144675021

SERV Architecture

The world's smallest, award-winning bit-serial RISC-V CPU

https://github.com/olofk/serv

- Designed by Olof Kindgren for **extreme logic minimization**
- Executes 32-bit ops bit-by-bit across 32 cycles
- 1-bit ALU, all instructions processed serially
- FSM/microcoded engine; every instruction multi-cycle
- Implements RV32I ISA, passes RISC-V compliance tests
- Supports RISC-V debug interface and GDB debugging

- **Clock Frequency**
  - Up to >200 MHz (due to ultra-simple timing)
- **Performance Metrics**
  - DMIPS: ~0.1–0.2 / MHz
  - CoreMark: ~0.05 / MHz
  - Extremely low IPC; 32-cycle per ALU op typical
- **Resource Usage**
  - LUTs: ~125
  - FFs: ~164
  - BRAM: 0
  - DSP: 0
  - Smaller than many UART/SPI controllers

Use Cases in Extremely Logic-Constrained or Massively Parallel Systems

Suitable Scenarios:

- Slow control tasks at the scale of tens of thousands of channels (e.g., detector calibration)
- Replacing hard-wired finite state machines with small soft logic cores
- Control cores in radiation-hardened FPGAs where only single-digit LUTs are available

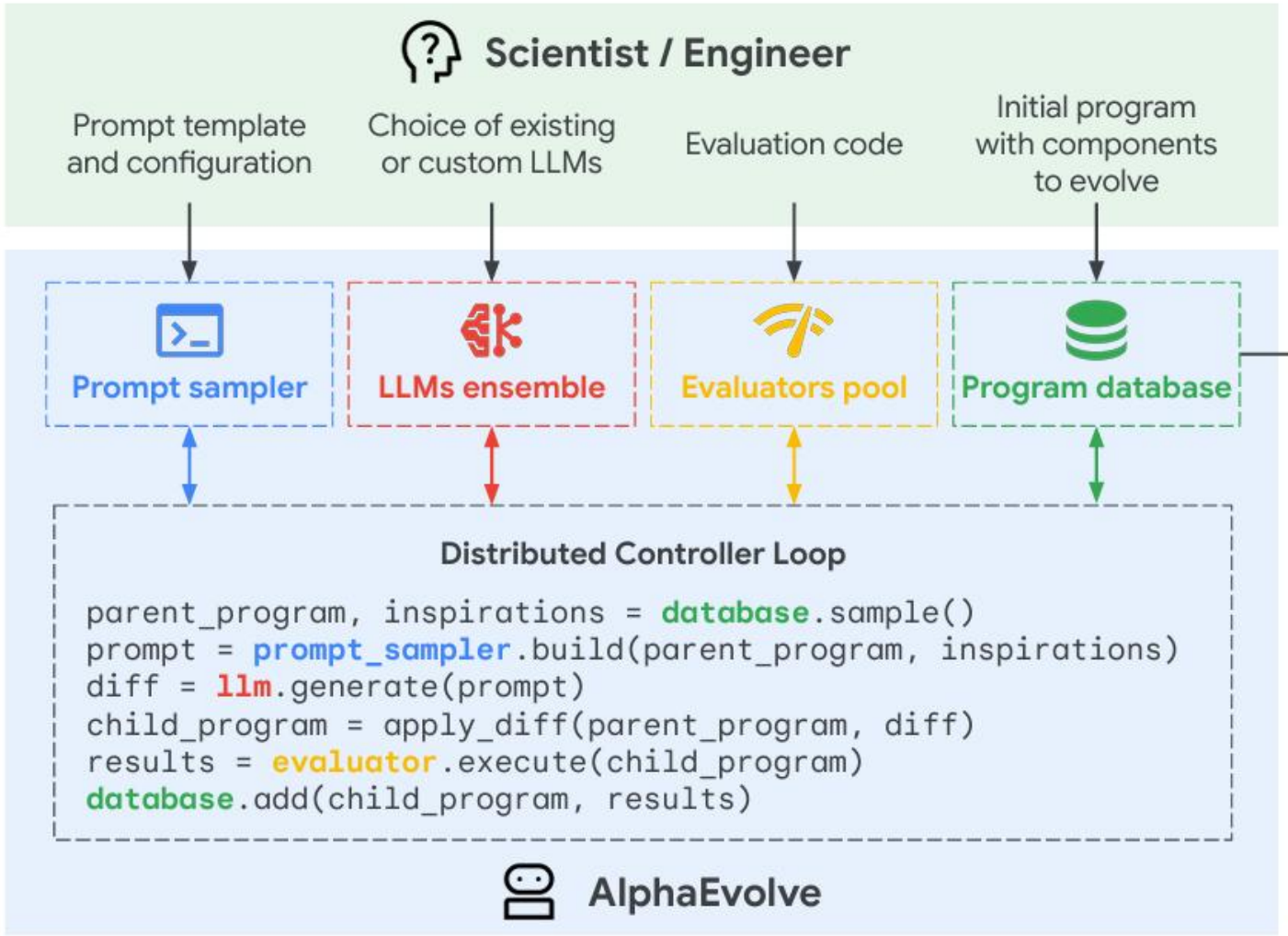
Advantages:

- Can be deployed in parallel as a “Core Swarm” system
- Each core supports single-step debugging (GDB supported)
- Negligible logic cost compared to mid-speed or high-performance cores

Limitations:

- Extremely low throughput, unsuitable for real-time data paths
- Not suitable for high-frequency processing such as DAQ trigger algorithms

通用科学AI--AlphaEvolve



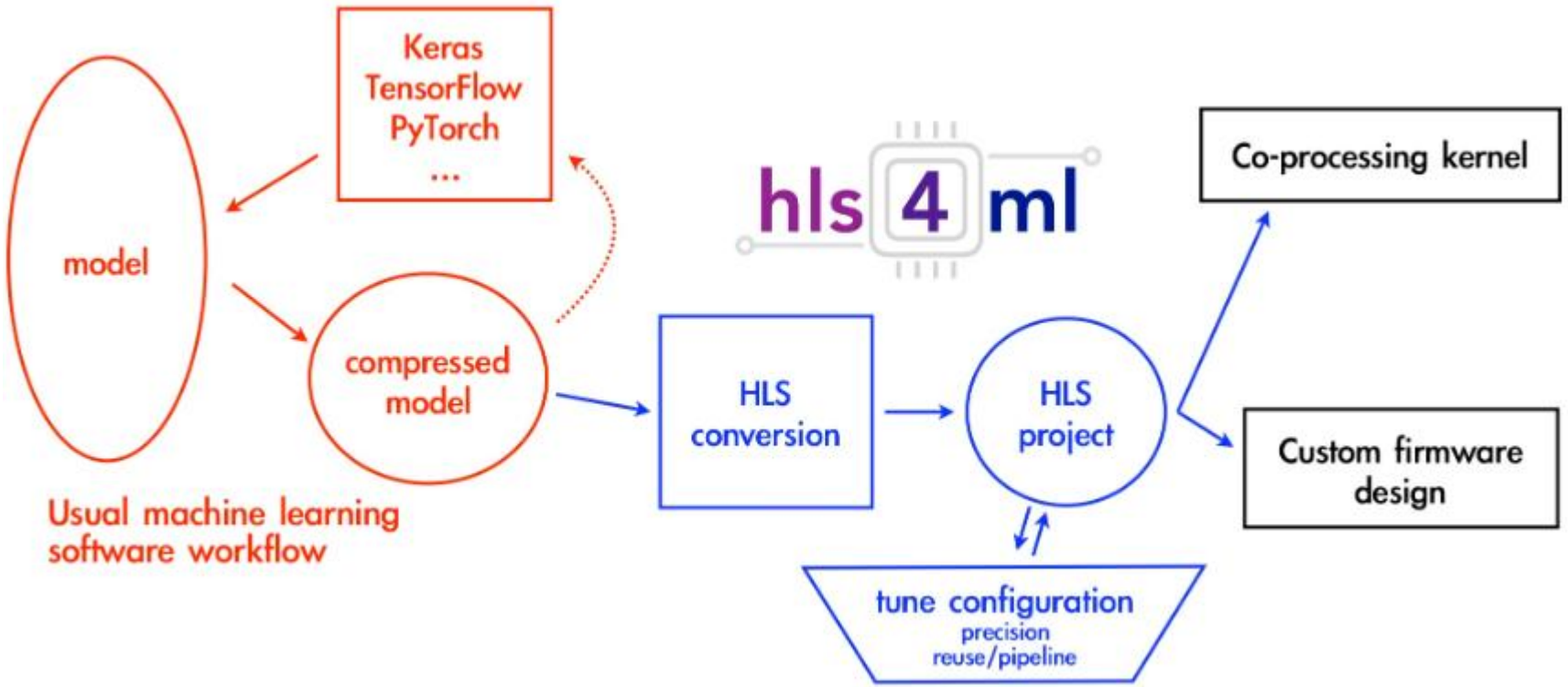
该系统基于 Gemini 系列大型语言模型构建，在用户输入问题及评估标准后，能够通过生成多种解决方案、评估筛选最优解并迭代生成新解的方式，逐步演化出更优的应对策略。

AlphaEvolve 还应用于超过 50 个数学领域的开放问题，并在近 75% 的案例中重新发现了最先进的解决方案，而在 20% 的案例中，它更是改进了已知的最佳方案，如提高了「亲吻数问题（kissing number problem）」的解决效率，这是一个困扰数学家超 300 年的难题。

source:https://deepmind.google/discover/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/

HLS4ML

灵感来自LHC， 机器学习通常是在 “离线” 环境中执行的。 然而，LHC 探测器的最大问题之一是每个Event会产生太多数据，以至于无法保存所有信息。 需要通过 “triggers” 的过滤器用于确定是否应保留给定事件。 机器学习算法可以在Sensor级别“实时”事例挑选，可以保留更多具有新物理场潜在迹象的事件以供分析。



该项目利用高层次综合（HLS）技术，将常见的机器学习模型转换为适应特定应用的硬件代码

source：https://fastmachinelearning.org/hls4ml/index.html