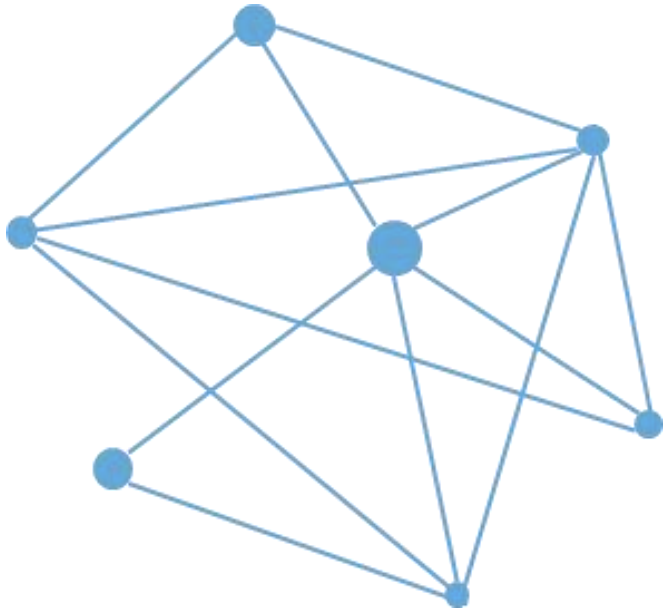


从设计到流片：
国内高能物理首款RISC-V芯片工程实现

报告人： 崔宇鑫

合作导师： 严琪 研究员

研发成员： 崔宇鑫、陈娇龙、骆首栋

测试团队： 王翰文、张奕晗

时　间： 2025-10-24

一、背景简介

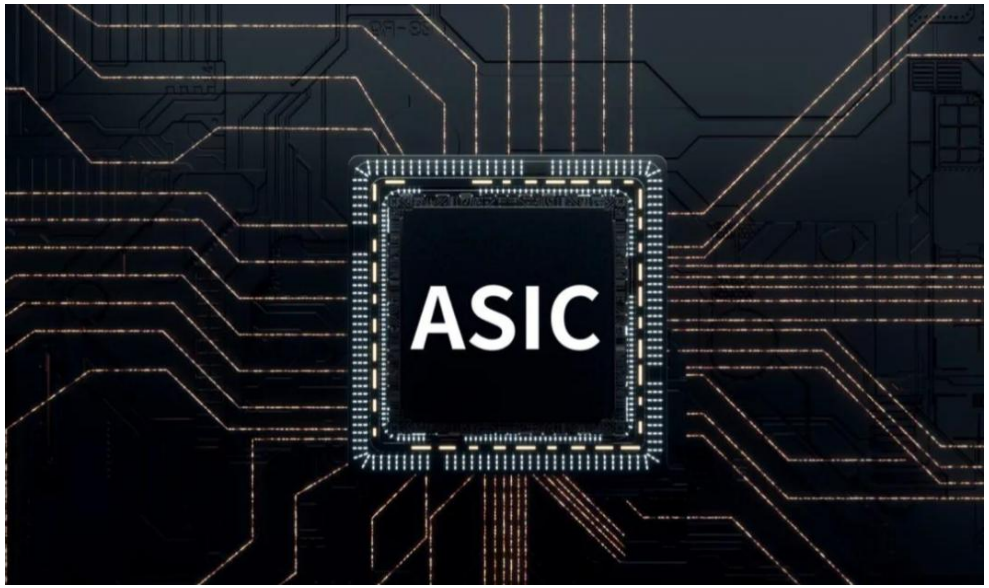
二、研发历程

三、后续规划

面向高能物理的可重构智能SoC（System on Chip）

一、背景与挑战（Why）

- 先进工艺成本高、设计复杂
  - 先进节点流片迭代价格昂贵
  - 要求首次设计成功，设计验证难度大
- 传统ASIC局限性明显
  - 功能固定、可配置性弱
  - 难以集成片上算法，更新与调试成本高
  - 无法满足高能物理实验的多样化与快速迭代需求



二、新的设计范式（How）

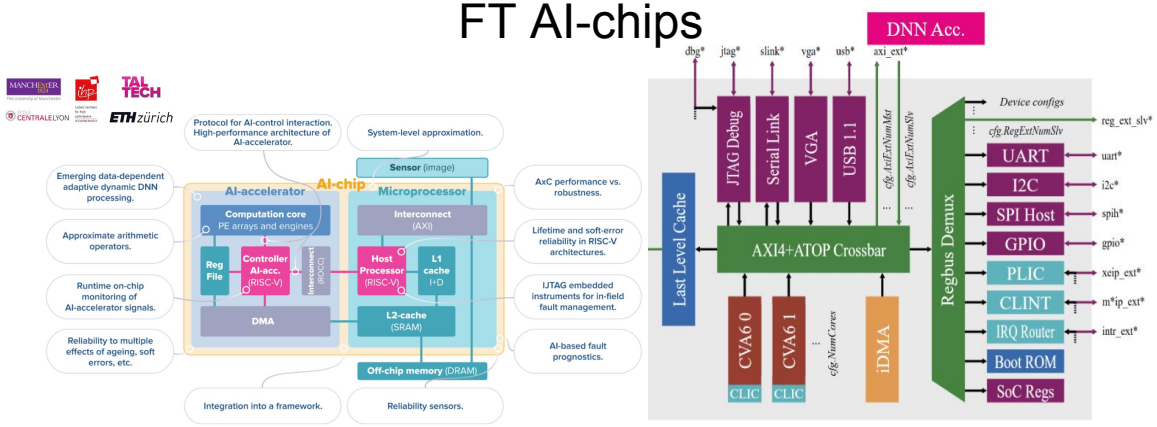
- 引入RISC-V的SoC架构，实现软硬件协同
  - 可编程性：通过软件配置实现多应用适配
  - 模块化与IP复用：提升设计协同与效率
  - 标准化互联：确保不同模块兼容性，构建开放生态
  - **智能芯片SoC是ASIC发展的未来趋势**

Risc-V在高能物理的应用(CERN DRD 7.2 相关Risc-V芯片)

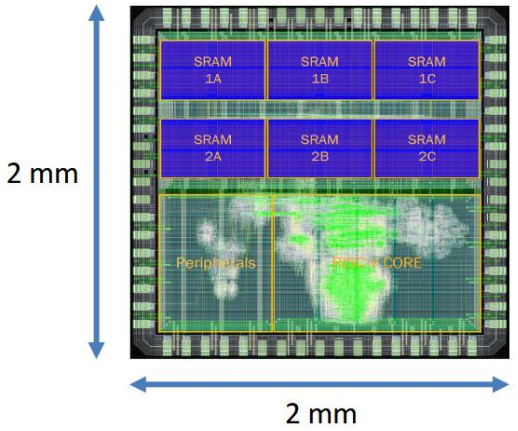
为了应对未来高能物理实验需求，CERN在Risc-V的应用主要聚焦于两个核心方向：

- 构建高可靠性的监控与控制系统。
- 发展智能化的片上数据处理。将Risc-V作为主控核心，结合AI硬件加速模块协同工作，可以在探测器前端直接对海量数据进行智能分析和事件筛选，提升发现新物理现象的效率。

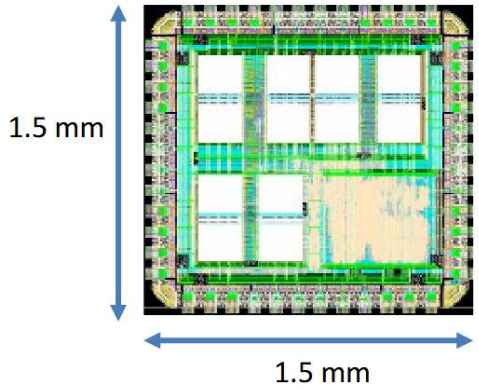
FT AI-chips



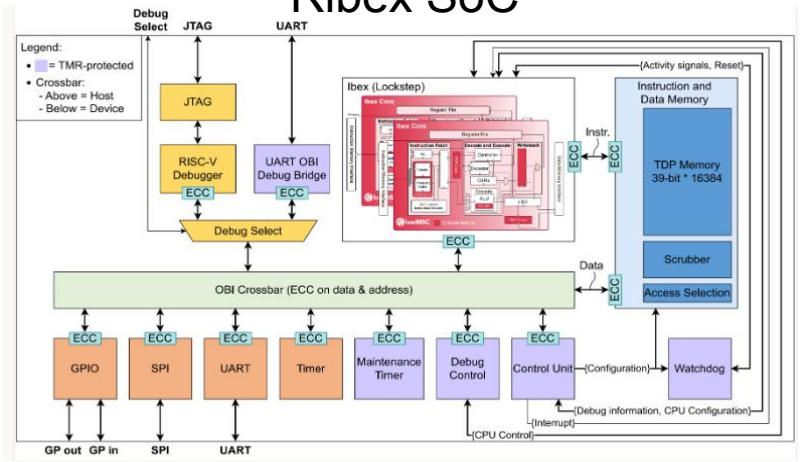
STRV-1 (65nm)



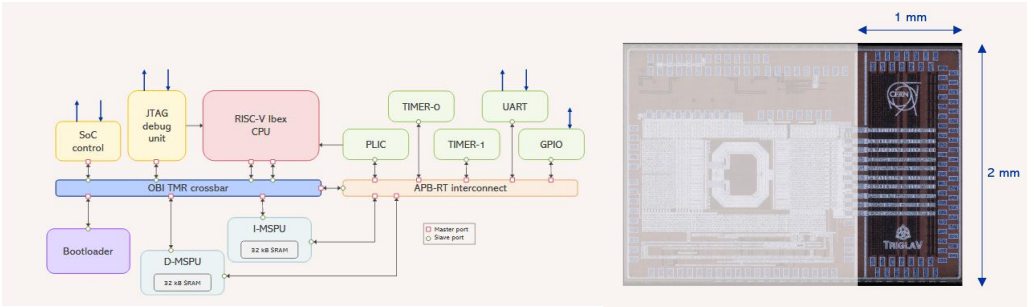
STRV-2 (28nm)



Ribex SoC

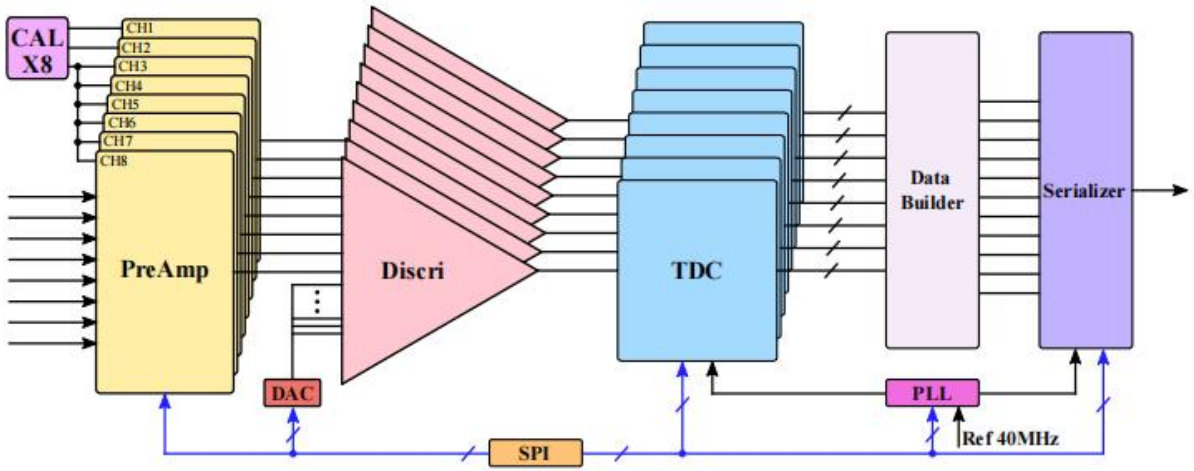


TriglaV

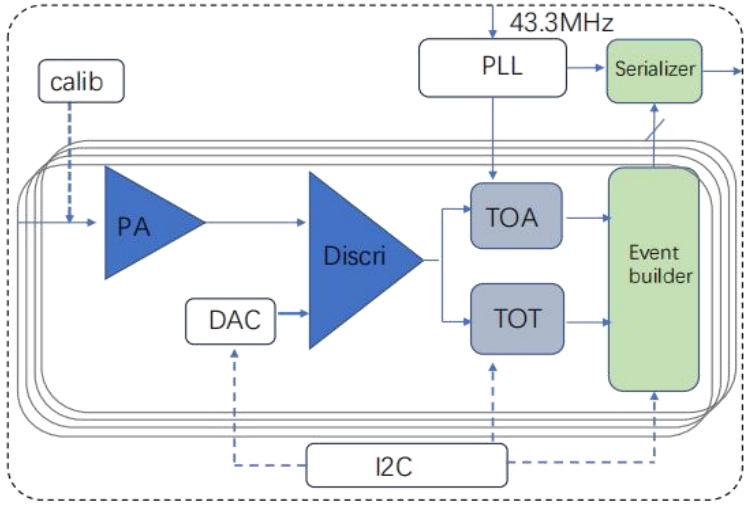


RISC-V应用于LATRIC

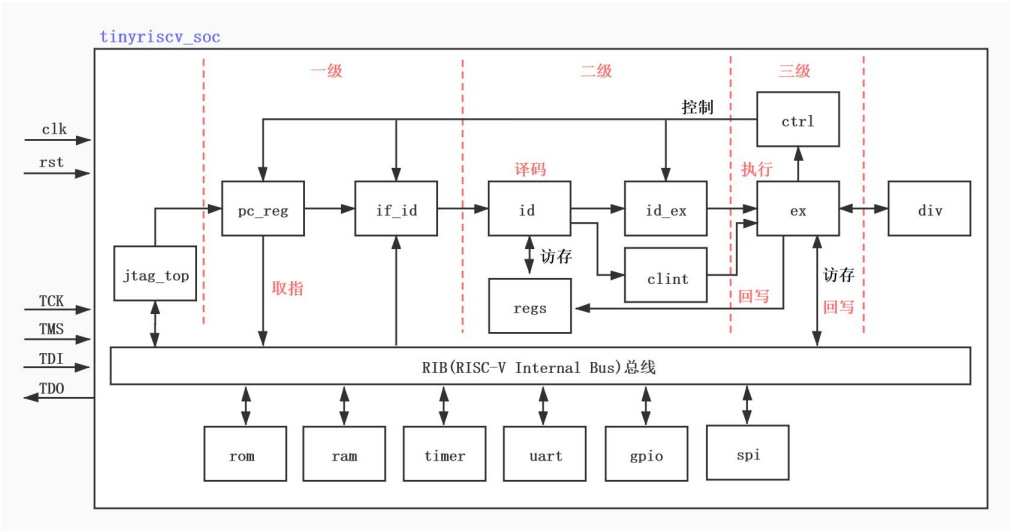
研究的起步阶段，我们选用入门友好的Tiny-Riscv软核，在OTK上实现一个动态设置的的DAC模块和其他算法



FPMROC示意图



LATRIC示意图



Tiny-RiscV架构图

- 支持RV32IM指令集；
- 采用三级流水线，即取指，译码，执行；
- 可以运行C语言程序；
- 支持JTAG，可以通过openocd在线更新程序；
- 支持中断；
- 支持总线；
- 支持FreeRTOS；
- 支持通过串口更新程序；
- 容易移植到任何FPGA平台；

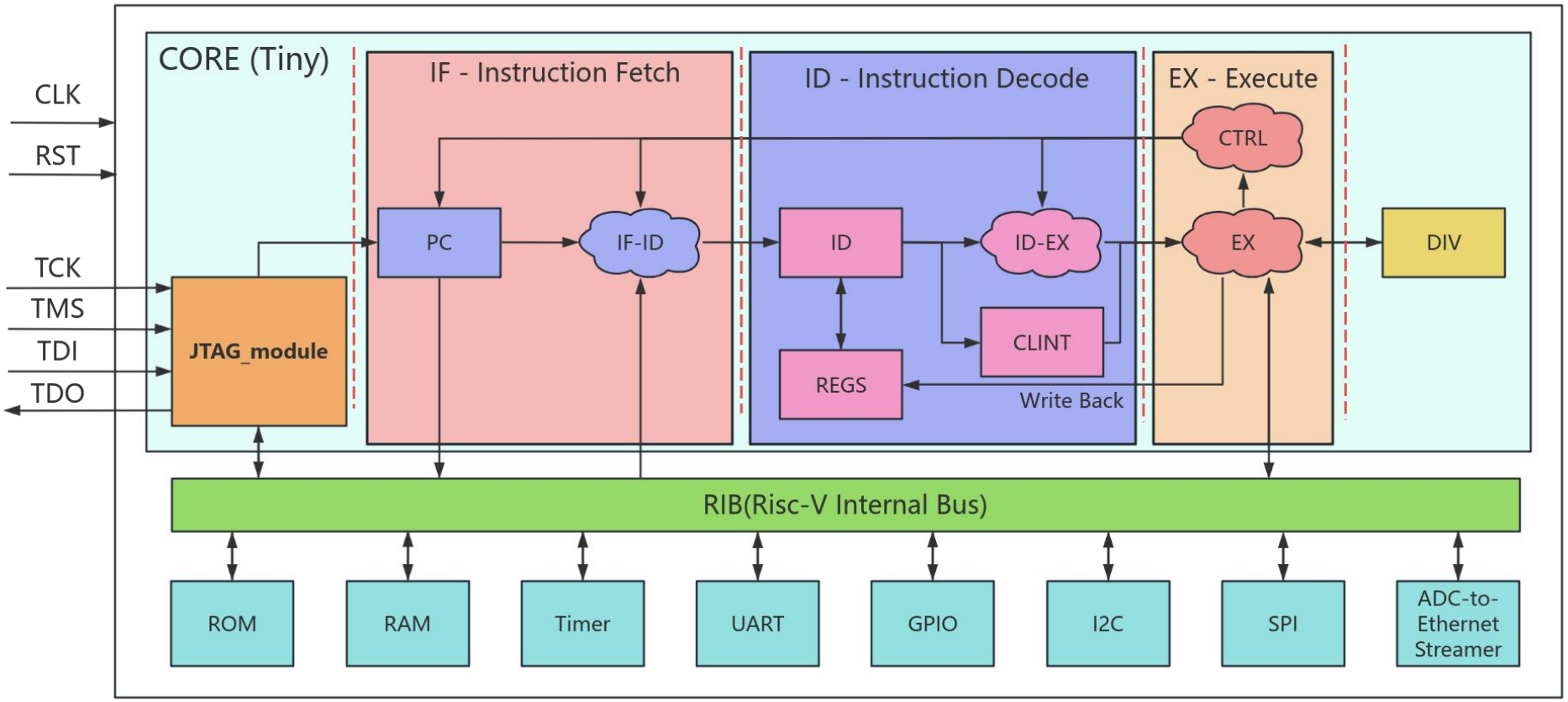
一、背景简介

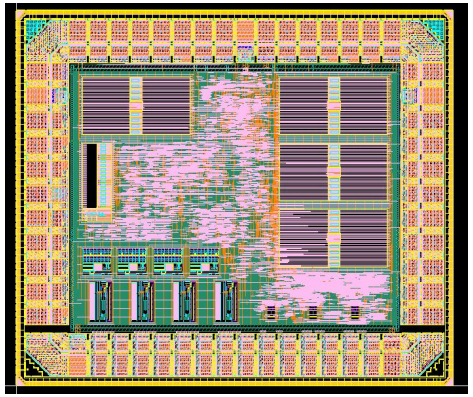
二、研发历程

三、后续规划

最终SOC架构

- 核心部分 (Tiny RiscV)
  - RV32IM 指令集
  - 三级流水线
  - CoreMark/MHz = 2.4
- JTAG 接口
  - 支持OpenOCD
  - 支持GDB 调试启动
- 外设
  - 4 KB ROM, 32 KB RAM
  - 支持I2C、UART、SPI
  - 集成ADC数据收集模块并通过UDP协议传输到上位机





- 55 nm 工艺制程
- 工作频率 50 MHz
- 面积 1020 x 1196 um
- 工作电压 1.2 V
- 10月份第一版设计验证完成并已经提交流片
- 正式芯片测试预计于2026年1月回片后开展

Risc-V程序运行示例

C程序源码

```
int a = 0;   // 结果
int b = 10;  // 操作数1
int c = 20;  // 操作数2

void main() {
    a = b + c; // 核心操作
}
```

编译

汇编指令	机器码	说明
...	...	...
la x5, a （伪指令）	(展开为两条指令)	la: 加载a的地址到x5寄存器 (作为数据段基址)
lw x6, 4(x5) (Load Word)	0x0042A303	从x5+4地址 (b的位置) 加载值(10)到x6寄存器
lw x7, 8(x5) (Load Word)	0x0082A383	从x5+8地址 (c的位置) 加载值(20)到x7寄存器
add x6, x6, x7 (Add)	0x00730333	将x6和x7的值相加，结果(30)存回x6
sw x6, 0(x5) (Store Word)	0x0062A023	将x6寄存器的值(30)存到x5+0地址 (a的位置)

取指 → 译码 → 执行

程序计数器(PC) →

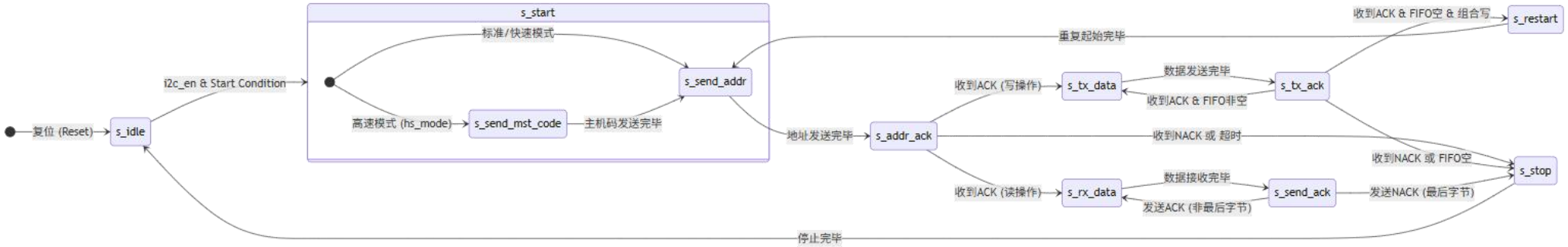
0x10008000:0x0042A303
0x10008004:0x00730333
0x10008008:0x00730333
0x1000800C:0x0062A023

字段	funct7	rs2	rs1	funct3	rd	opcode
位域	[31:25] (7位)	[24:20] (5位)	[19:15] (5位)	[14:12] (3位)	[11:7] (5位)	[6:0] (7位)
用途	功能码7 (区分指令)	源寄存器2	源寄存器1	功能码3 (区分指令)	目的寄存器	操作码 (指令类型)
	0000000	00111	00110	000	00110	0110011

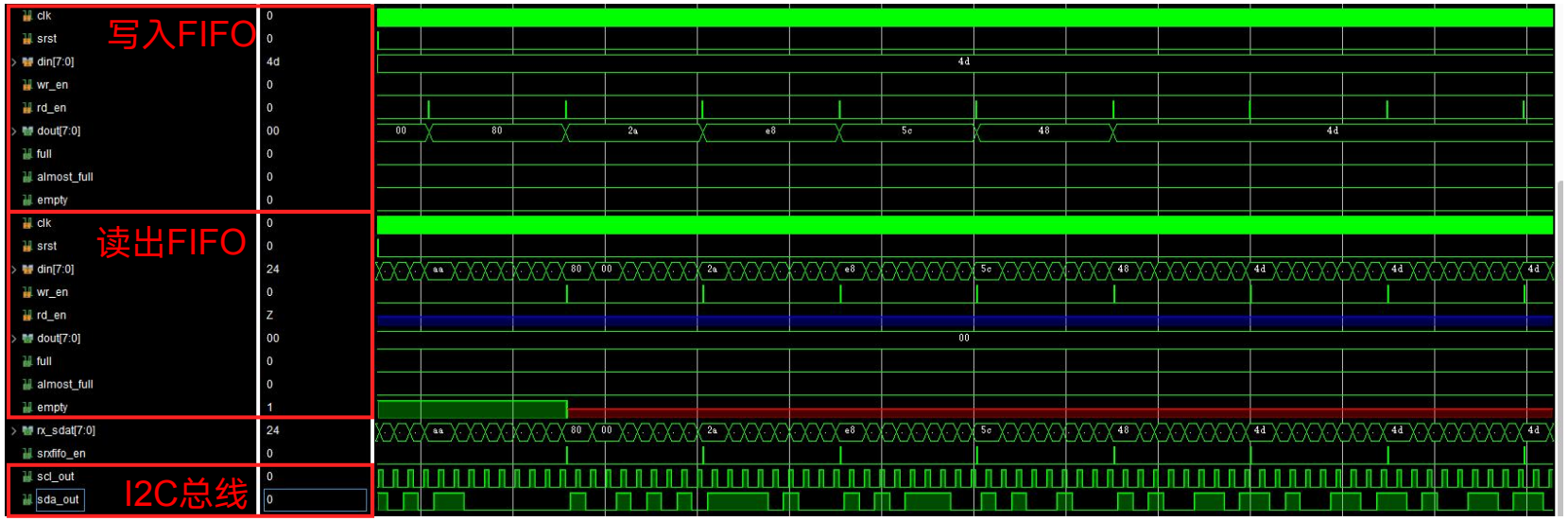
添加I2C主机模块

I2C主机选择开源网站Opencores (https://opencores.org/)提供的较为成熟版本，并对主机做一层封装挂载于总线

主机状态转移图



仿真程序图



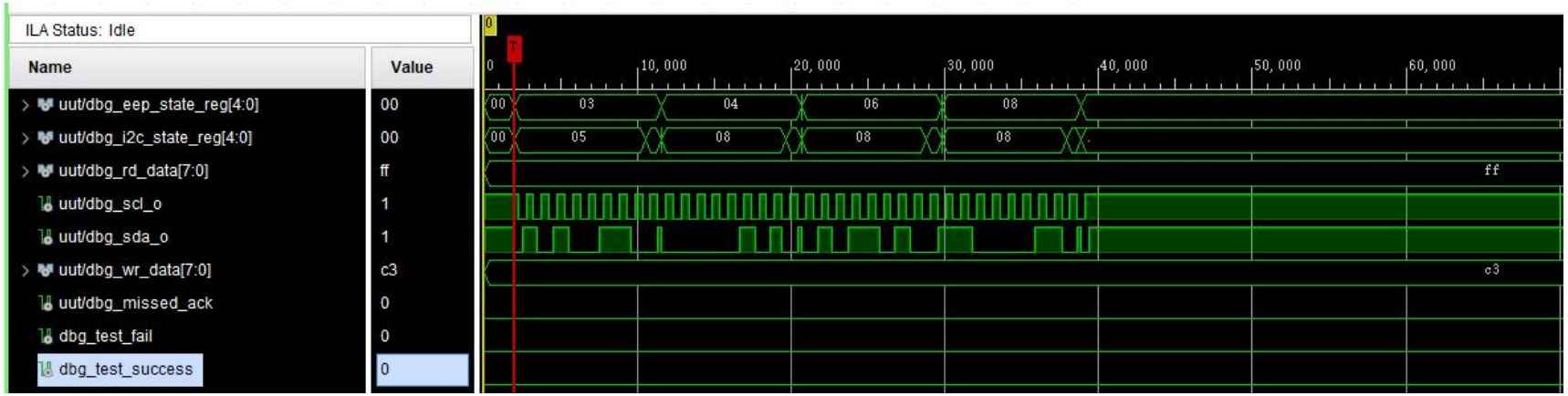
I2C模块FPGA验证

I2C使用开发板上自带的AT24C64 EEPROM 存储芯片验证模块功能

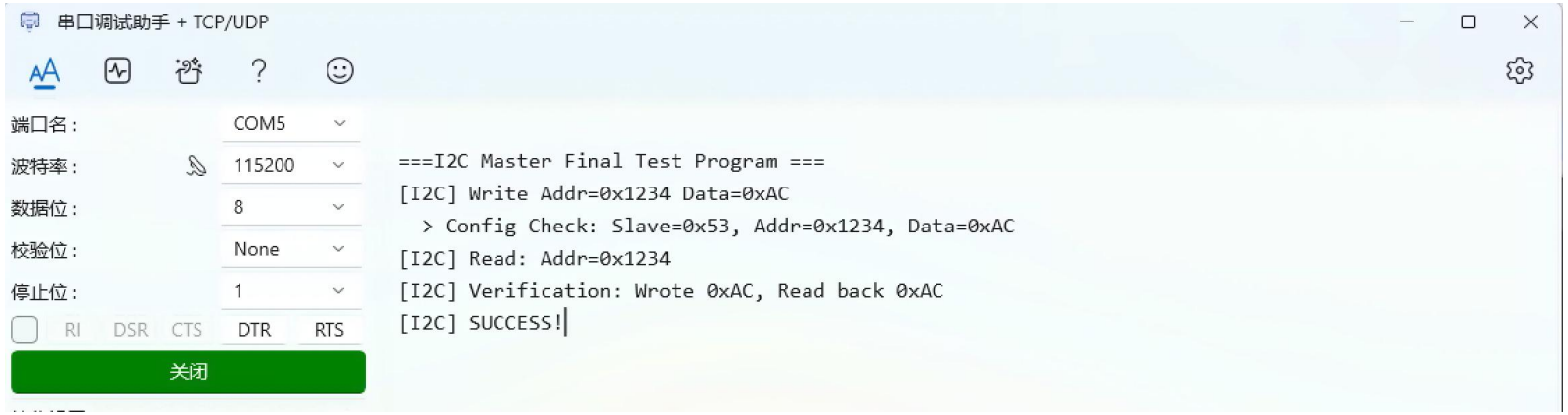
C程序

```
#include <stdint.h>
#include "../../bsp/include/uart.h"
#include "../../bsp/include/xprintf.h"

// ----------------------------------------------------------------------
// == THE CRITICAL FIX: Base address updated to the correct value ==
// ----------------------------------------------------------------------
#define I2C_MASTER_BASE_ADDR 0x50000000

#define I2C_REG_PRESCALE  0x00
#define I2C_REG_CMD       0x04
#define I2C_REG_SLAVE_ADDR 0x08
#define I2C_REG_MEM_ADDR  0x0C
#define I2C_REG_DATA      0x10
#define I2C_REG_STATUS    0x20

#define I2C_PRESCALE  (*(volatile uint32_t *)(I2C_MASTER_BASE_ADDR + I2C_REG_PRESCALE))
#define I2C_CMD       (*(volatile uint32_t *)(I2C_MASTER_BASE_ADDR + I2C_REG_CMD))
#define I2C_SLAVE_ADDR (*(volatile uint32_t *)(I2C_MASTER_BASE_ADDR + I2C_REG_SLAVE_ADDR))
#define I2C_MEM_ADDR  (*(volatile uint32_t *)(I2C_MASTER_BASE_ADDR + I2C_REG_MEM_ADDR))
#define I2C_DATA      (*(volatile uint32_t *)(I2C_MASTER_BASE_ADDR + I2C_REG_DATA))
#define I2C_STATUS    (*(volatile uint32_t *)(I2C_MASTER_BASE_ADDR + I2C_REG_STATUS))

#define CMD_GO    (1 << 31)
#define CMD_WRITE (1 << 30)
#define CMD_READ  (1 << 29)
#define STATUS_BUSY    (1 << 0)
#define STATUS_ACK_ERROR (1 << 1)

void delay_ms(volatile uint32_t ms) { for (uint32_t i = 0; i < ms; ++i) for (volatile uint32_t j = 0; j < 5000; ++j); }

void i2c_init(uint32_t f_clk, uint32_t f_scl) {
    while(I2C_STATUS & STATUS_BUSY);
    I2C_PRESCALE = f_clk / (4 * f_scl);
}

#define EEPROM_SLAVE_ADDR 0x53 // The address you confirmed works

void eeprom_write_byte(uint16_t mem_addr, uint8_t data) {
    xprintf("[I2C] Write: Addr=0x%04X Data=0x%02X\n", mem_addr, data);
    while(I2C_STATUS & STATUS_BUSY);

    // Phase 1: Configure
    I2C_SLAVE_ADDR = EEPROM_SLAVE_ADDR;
    I2C_MEM_ADDR  = mem_addr;
    I2C_DATA      = data;

    // Optional but recommended: A quick readback to confirm writes were received
    xprintf("  > Config Check: Slave=0x%X, Addr=0x%X, Data=0x%X\n", (uint8_t)I2C_SLAVE_ADDR, (uint16_t)I2C_MEM_ADDR, (uint8_t)I2C_DATA);

    // Phase 2: Execute
    I2C_CMD = CMD_GO | CMD_WRITE;

    while(I2C_STATUS & STATUS_BUSY);
    if (I2C_STATUS & STATUS_ACK_ERROR) {
        xprintf("[I2C] ERROR: NACK detected during write!\n");
    }
}

uint8_t eeprom_read_byte(uint16_t mem_addr) {
    xprintf("[I2C] Read: Addr=0x%04X\n", mem_addr);
    while(I2C_STATUS & STATUS_BUSY);

    // Phase 1: Configure
    I2C_SLAVE_ADDR = EEPROM_SLAVE_ADDR;
    I2C_MEM_ADDR  = mem_addr;

    // Phase 2: Execute
    I2C_CMD = CMD_GO | CMD_READ;

    while(I2C_STATUS & STATUS_BUSY);
    if (I2C_STATUS & STATUS_ACK_ERROR) {
        xprintf("[I2C] ERROR: NACK detected during read!\n");
        return 0xFF;
    }
    return (uint8_t)I2C_DATA;
}

int main() {
    uart_init();
    xprintf("\n=== I2C Master Final Test Program ===\n");
    i2c_init(50000000, 100000);

    uint16_t test_addr = 0x1234;
    uint8_t test_data  = 0xAC;

    eeprom_write_byte(test_addr, test_data);
    uint8_t read_back = eeprom_read_byte(test_addr);

    xprintf("[I2C] Verification: Wrote 0x%02X, Read back 0x%02X\n", test_data, read_back);
    if (read_back == test_data) {
        xprintf("[I2C] SUCCESS!\n");
    } else {
        xprintf("[I2C] FAILURE!\n");
    }

    while(1);
    return 0;
}
```

ILA调试结果



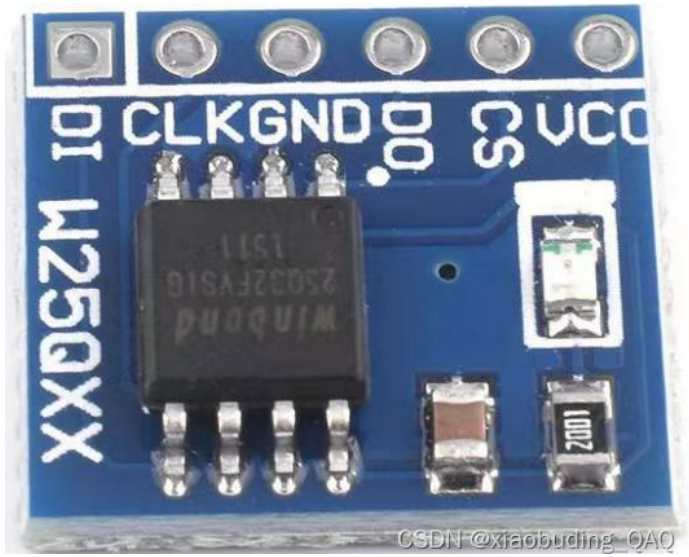
UART输出结果



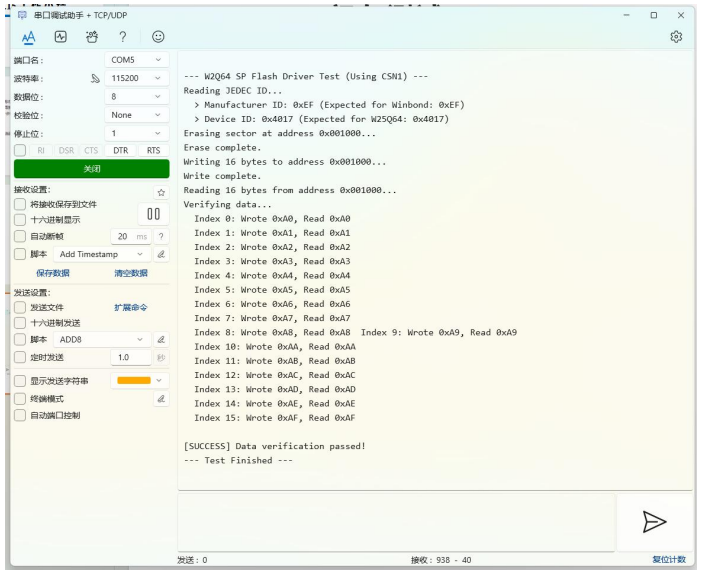
优化并测试SPI主机模块

更换原有SPI模块为更成熟的SPI主机模块 (https://github.com/nandland/spi-master), 并添加一层封装，使用基于SPI协议的W25Q64 存储芯片,在FPGA上成功验证模块功能

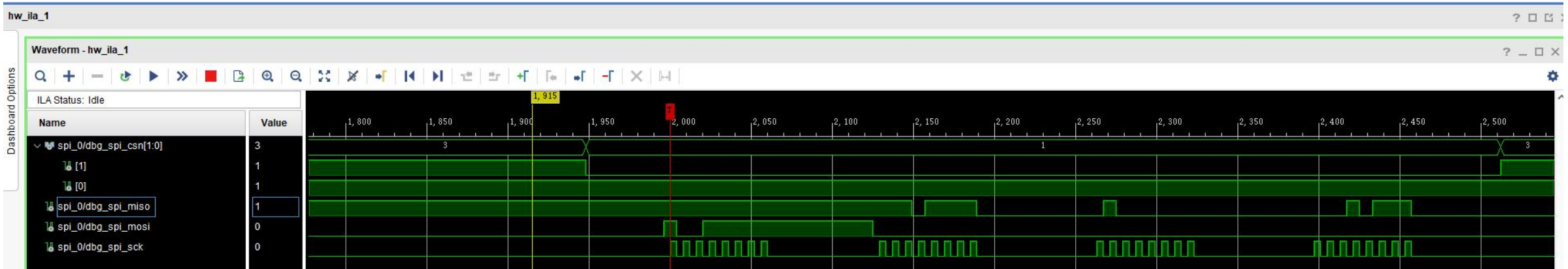
W25Q64



UART输出结果

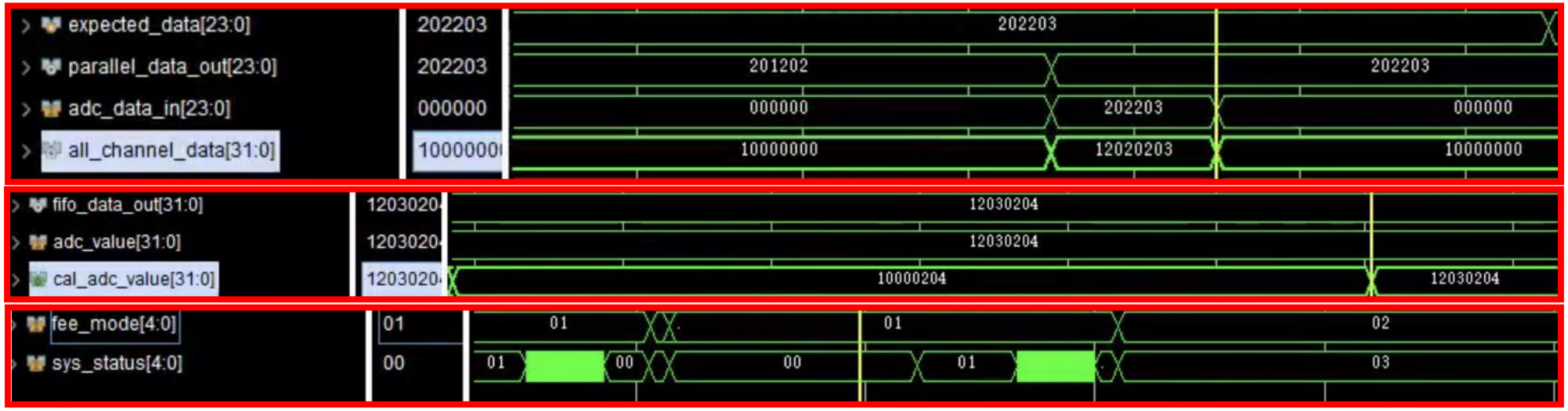


ILA结果



添加ADC-Eithernet 模块

接受串行输入数据并进行处理后通过UDP协议传输至上位机



ADC串行输入数据

串转并后的并行数据

ADC数据接收模块接收数据

FIFO输出数据

C程序输入数据

C程序反馈数据

工作模式切换

系统状态切换

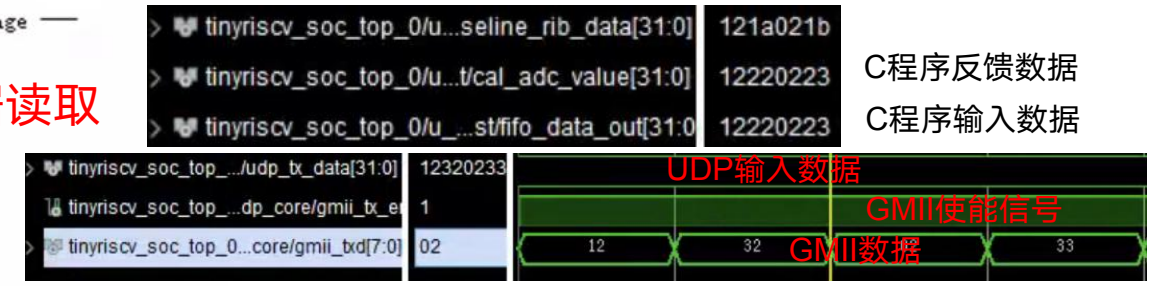
上板验证

初始化配置

```
── System Initialization ──
Board IP set to: 192.168.185.111 (0xC0A8B96F)
Board port set to: 1234
Destination IP set to: 192.168.185.43 (0xC0A8B9F3)
Board port set to: 1234
UDP Cofig Reg (0x10) written with: 0x00030400
UDP Config Reg (0x10) read back: 0x00030400
UDP Config Reg (0x10) written with: 0x00030400
```

```
── Measuring Baseline Voltage ──
Signal@CH0: 516
Signal@CH1: 515
Signal@CH0: 517
Signal@CH1: 516
Signal@CH0: 518
Signal@CH1: 517
Signal@CH0: 519
Signal@CH1: 518
Signal@CH0: 520
Signal@CH1: 519
```

数据读取



C程序反馈数据

C程序输入数据

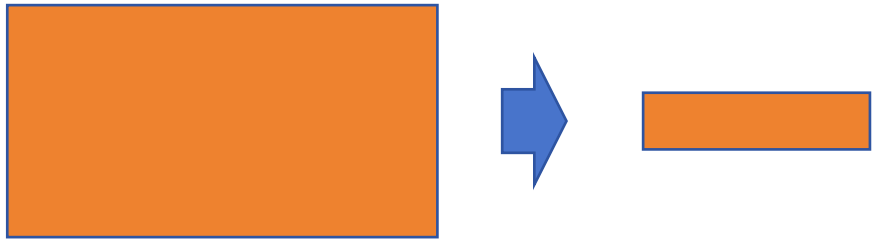
优化启动方式

Tiny-RiscV(Bram)设计于FPGA中使用，Rom模块实际为Block-RAM，大小为32 KB。只能通过JTAG启动：

- 缺少自启动的方式，可靠性、便利性不足；
- ROM面积开销过大；
- 优势在于无需外部存储器；

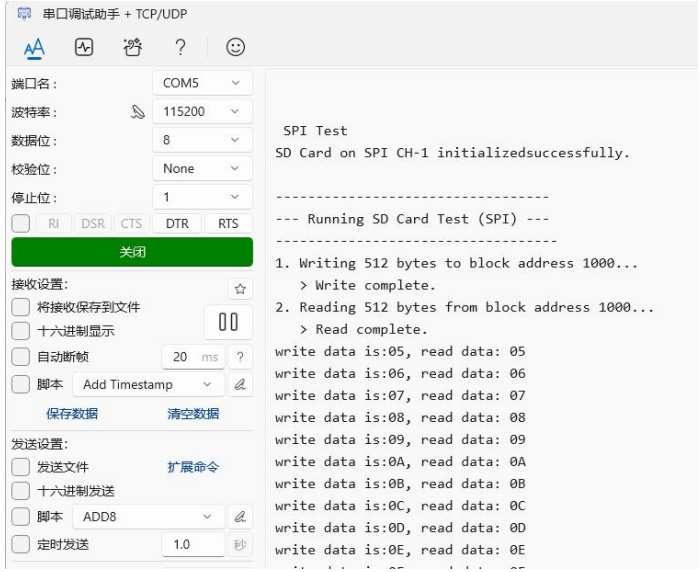
RAM 8192x32
569.26*307.045 um

ROM 512x32
75.845*300.46



考虑到器件的通用性和便携性，我们考虑基于SPI协议的SD卡作为外部存储器，ROM内为boot引导程序，上电后从SD卡中读取应用程序, 大小由32 KB 改为 4 KB（非调试版本2 KB）

SD卡测试输出结果



BOOT引导程序设计

Boot引导程序：当前RAM大小为16k，程序可用14k，最高位2k用于boot引导程序存放变量，加入UART输出信息用于调试

lds链接文件

```
/*
 * bootloader.lds (Corrected Final Version)
 * Defines separate RAM regions to guarantee isolation. This version uses
 * direct calculation within the MEMORY block for maximum compatibility.
 */
OUTPUT_ARCH( "riscv" )
ENTRY(_start_bootloader)

/* Define constants for easy modification */
RAM_TOTAL_LENGTH    = 16K;
BOOTLOADER_RAM_SIZE = 2K; /* Reserve 2KB at the top of RAM for the bootloader */

MEMORY
{
  rom (rx)          : ORIGIN = 0x00000000, LENGTH = 4K

  /* RAM available for the application. Its length is calculated directly. */
  app_ram (wxa!ri)  : ORIGIN = 0x10000000, LENGTH = RAM_TOTAL_LENGTH - BOOTLOADER_RAM_SIZE

  /* RAM reserved for bootloader's private use. Its origin is calculated directly. */
  boot_ram (wxa!ri) : ORIGIN = 0x10000000 + (RAM_TOTAL_LENGTH - BOOTLOADER_RAM_SIZE), LENGTH = BOOTLOADER_RAM_SIZE
}

SECTIONS
{
  /* Place code and read-only data in ROM */
  .text :
  {
    KEEP (*(SORT_NONE(.init)))
    *(.text .text.*)
    *(.rodata .rodata.*)
  } >rom

  /* Place the bootloader's .bss section into its reserved RAM region */
  .bss (NOLOAD) :
  {
    . = ALIGN(4);
    __bss_start = .;
    *(.bss .bss.*)
    *(COMMON)
    . = ALIGN(4);
    __bss_end = .;
  } >boot_ram

  /* The stack pointer is initialized to the top of the bootloader's RAM region */
  _stack_start = ORIGIN(boot_ram) + LENGTH(boot_ram);
}
```

start_s汇编

```
// bootloader/bootloader_start.S
.section .init
.globl _start_bootloader

_start_bootloader:
    // 1. Initialize the stack pointer to the top of the main RAM.
    //    The linker script provides the '_stack_start' symbol.
    la sp, _stack_start

    // Optional: Clear the .bss section for the bootloader.
    // For a minimal bootloader, this step can often be skipped if you know
    // your variables don't rely on being zero-initialized. But it's good practice.
    la t0, __bss_start
    la t1, __bss_end
.L_clear_bss_boot:
    beq t0, t1, .L_clear_bss_boot_done
    sw zero, (t0)
    addi t0, t0, 4
    j .L_clear_bss_boot
.L_clear_bss_boot_done:

    // 2. Jump to the C entry point of the bootloader
    jal ra, boot_main_c

.L_hang:
    // If boot_main_c returns, hang here.
    j .L_hang
```

boot主程序

```

// ... (配置宏定义保持不变) ...
#define APP_START_BLOCK_ON_SD_CARD   0 // 使用方案A，从块0读取
#define APP_RUN_ADDR_ON_CHIP_RAM   0x10000000
#define ON_CHIP_RAM_SIZE_BYTES     (14 * 1024)
#define MY_SD_CARD_SPI_CHANNEL     1

typedef void (*app_entry_t)(void);
const app_entry_t app_entry = (app_entry_t)APP_RUN_ADDR_ON_CHIP_RAM;

Explain code | Complete comments | Locate code bugs | Generate unit tests | Code Review | Close
void boot_main_c() {
    // STEP 1: 初始化UART，这是我们唯一的调试窗口
    // 注意：这里的uart_init()必须能够独立工作，不依赖于主程序中的任何东西。
    uart_init();

    // 发送一个启动信号，如果能在串口看到这个，说明Bootloader至少启动了
    DEBUG_PRINTF("\n\n--- RISC-V Bootloader Started ---\n");

    // STEP 2: 初始化SD卡
    DEBUG_PRINTF("Initializing SD card on SPI channel %d...\n", MY_SD_CARD_SPI_CHANNEL);
    if (sd_card_init(MY_SD_CARD_SPI_CHANNEL) != 0) {
        DEBUG_PRINTF("!!! FATAL: SD Card initialization failed. Halting. !!!\n");
        while(1); // 如果失败，打印信息并死循环
    }
    DEBUG_PRINTF("SD Card initialized successfully.\n");

    // STEP 3: 计算需要读取的块数
    uint32_t num_blocks_to_read = (ON_CHIP_RAM_SIZE_BYTES + SD_BLOCK_SIZE - 1) / SD_BLOCK_SIZE;
    DEBUG_PRINTF("Target RAM size: %d bytes. Need to read %d blocks from SD card.\n", ON_CHIP_RAM_SIZE_BYTES, num_blocks_to_read);

    // STEP 4: 循环拷贝
    DEBUG_PRINTF("Starting to copy data from SD:@0x%X to RAM:@0x%X...\n", APP_START_BLOCK_ON_SD_CARD, APP_RUN_ADDR_ON_CHIP_RAM);
    for (uint32_t i = 0; i < num_blocks_to_read; i++) {
        uint32_t source_block = APP_START_BLOCK_ON_SD_CARD + i;
        uint8_t* destination_address = (uint8_t*)APP_RUN_ADDR_ON_CHIP_RAM + (i * SD_BLOCK_SIZE);

        // 可以在这里加一个简单的进度指示
        if ((i % 4) == 0) { // 每4个块打印一个点
            DEBUG_PRINTF(".");
        }

        if (sd_card_read_block(source_block, destination_address) != 0) {
            DEBUG_PRINTF("\n!!! FATAL: Failed to read block %d. Halting. !!!\n", source_block);
            while(1); // 读取失败，打印信息并死循环
        }
    }
    DEBUG_PRINTF("\nData copy complete.\n");

    // STEP 5: 准备跳转
    DEBUG_PRINTF("Bootloader finished. Jumping to application at 0x%X...\n", app_entry);

    // 在跳转前可以加一个极短的延时，确保串口信息都发出去了
    for(volatile int i=0; i<10000; ++i);

    // 跳转
    app_entry();

    // 如果程序能执行到这里，说明跳转失败了
    DEBUG_PRINTF("!!! FATAL: Application returned to bootloader. Halting. !!!\n");
    while (1);
}
```

Boot引导程序相关硬件代码修改

1、取指模块，由 rst_n, flush_i, stall 这几个内部信号决定。没有任何来自外部调试模块的直接输入。

```
always @ (posedge clk or negedge rst_n) begin
    // 复位
    if (!rst_n) begin
        pc <= `CPU_RESET_ADDR;
        pc_prev <= 32'h0;
    // 冲刷
    end else if (flush_i) begin
        pc <= flush_addr_i;
    // 暂停，取上一条指令
    end else if (stall) begin
        pc <= pc_prev;
    // 取下一条指令
    end else begin
        pc <= pc + 32'h4;
        pc_prev <= pc;
    end
end
```

2、JTAG模块输入：在jtag_dm（debug module），根据RISC-V调试规范，这通常是通过写一个特殊的CSR寄存器，叫做 dpc (Debug Program Counter)。新增发出一个“PC写请求”信号 和新的PC值

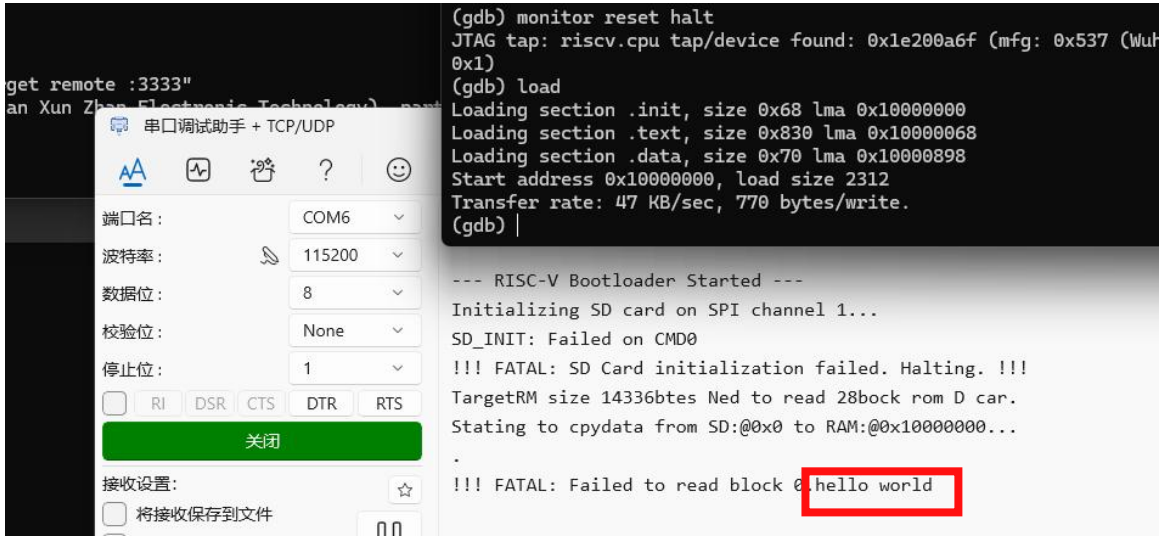
```
end else begin
    // when write dpc, we reset cpu here
    if (data[15:0] == DPC) begin
        //dm_reset_req <= 1'b1;
        jtag_pc_we <= 1'b1;       // <<-- 新增：发起PC写请求
        jtag_pc_wdata <= data0;  // <<-- 新增：要写入的PC值来自DATA0寄存器
        dm_halt_req <= 1'b0;
        dmstatus <= {dmstatus[31:12], 4'hc, dmstatus[7:0]};
    end else if (data[15:0] < 16'h1020) begin
        dm_reg_we <= 1'b1;
        dm_reg_wdata <= data0;
    end
```

当JTAG尝试写DPC时，当前硬件的反应不是去更新CPU的PC，而是拉高了一个复位请求信号

3、复用一条现有的、功能相似的通路，(flush 的本意就是“冲刷流水线，并从一个新的地址重新开始取指)，让JTAG模块“借用” flush 机制

```
// 当JTAG写入PC时，使用JTAG提供的新地址；否则，使用正常的跳转地址。
assign flush_addr_o = jtag_pc_we_i ? jtag_pc_wdata_i : jump_addr_i;

// 当发生正常跳转、CLINT暂停或JTAG写入PC时，都需要冲刷流水线。
assign flush_o = jump_assert_i | stall_from_clint_i | jtag_pc_we_i;
```

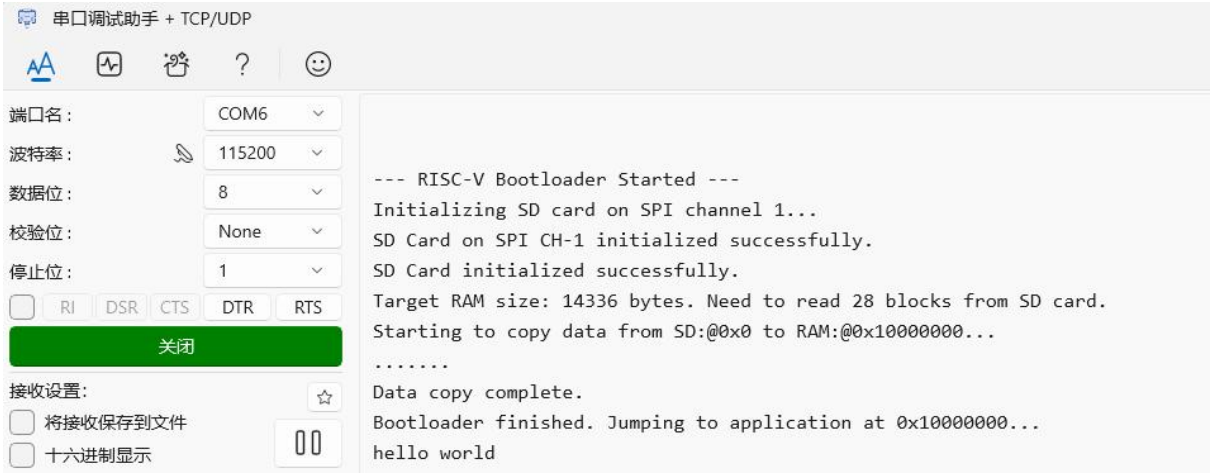
修改后正常通过GDB启动



C程序配套文件修改及FPGA上板验证

C程序根据引导程序的配置，使用配套的lds链接脚本和start_s脚本编译，并通过SD卡和GDB两种启动方式测试

方式一：
ROM+SD卡



ROM + SD卡正常启动输出的调试信息

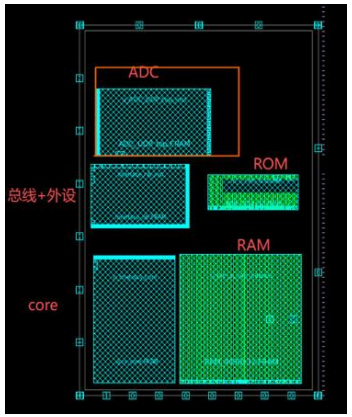
方式二：
GDB调试启动

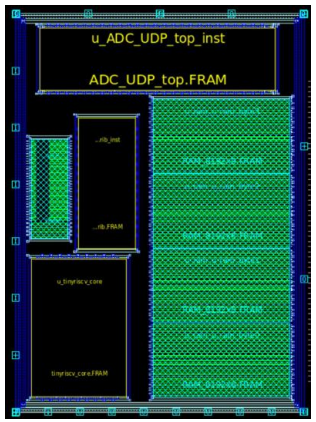
```
# file: debug.gdb
target remote localhost:3333
monitor reset halt
load
set $pc = 0x10000000
b main
c
```

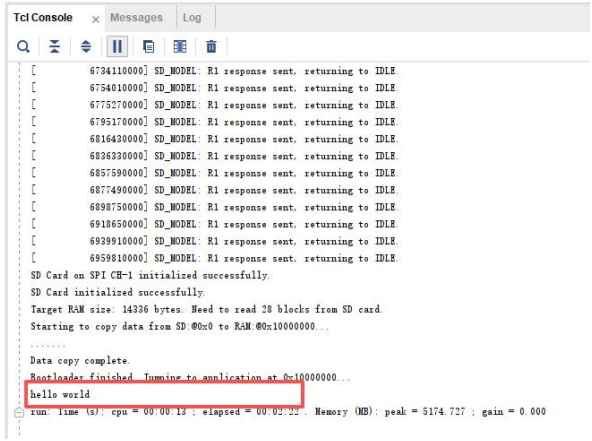
```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from .\gpio...
0x00000000 in ?? ()
JTAG tap: riscv.cpu tap/device found: 0x1e200a6f (mfg: 0x537 (Wuhan Xun Zhan Electronic Technology), part: 0xe200, ver:
0x1)
Loading section .init, size 0x6c lma 0x10000000
Loading section .text, size 0x260 lma 0x1000006c
Start address 0x10000000, load size 716
Transfer rate: 99 KB/sec, 358 bytes/write.
Breakpoint 1 at 0x100001a8: file main.c, line 9.
```

RAM修改

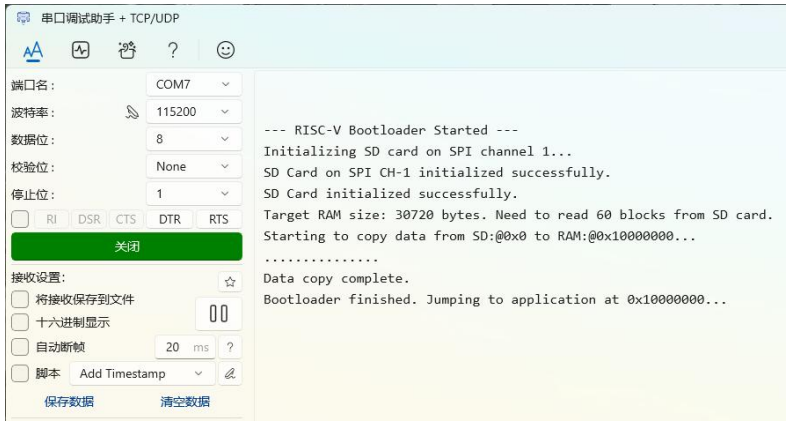
当前的架构需要实现字节选功能，即能只改变32位中的8位，而厂商提供的SRAM可能不支持该功能，会导致部分指令多一个周期，因此将32位RAM改为4个8位RAM组合，并新增SD仿真的Verilog模型用于前后仿验证



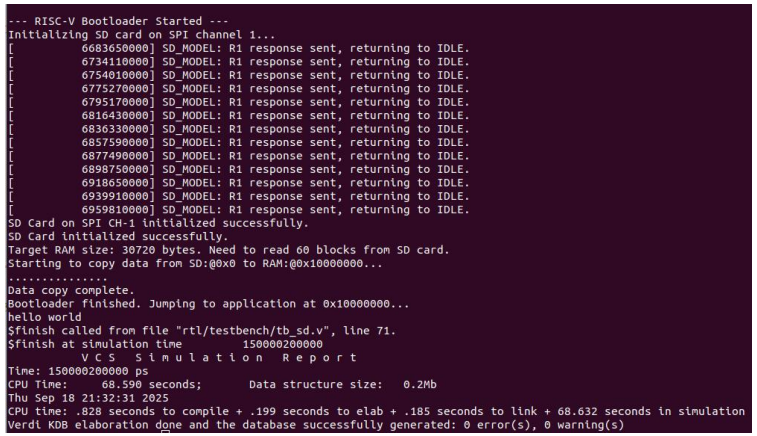




Vivado仿真

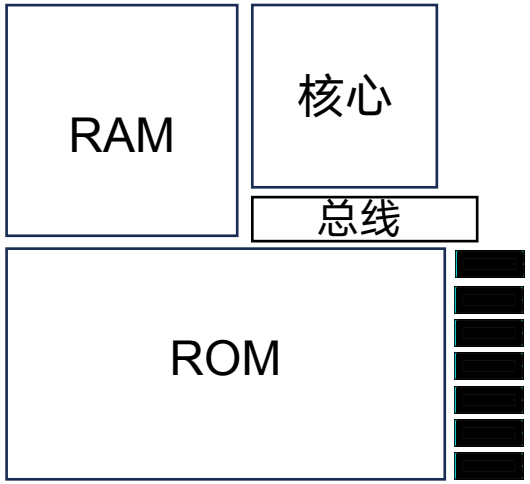


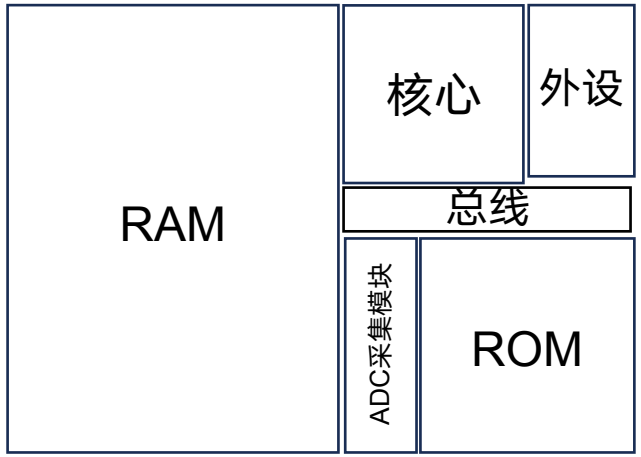
FPGA验证

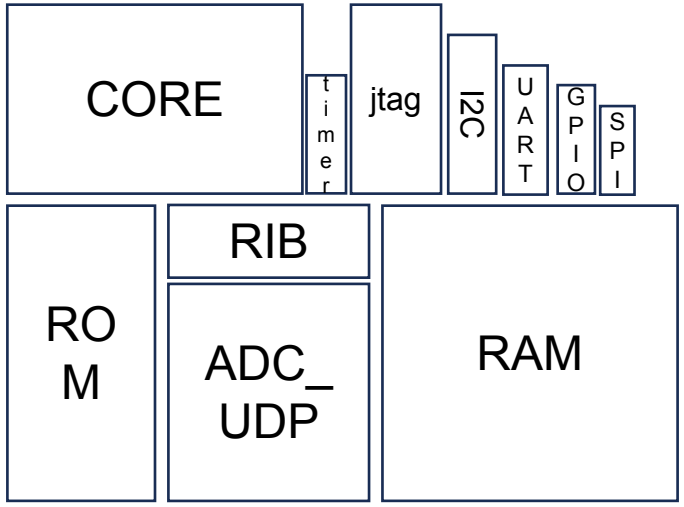


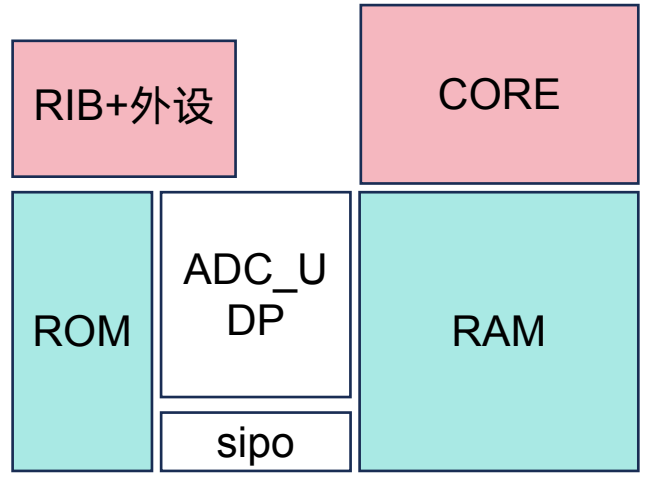
VCS仿真

数字IC设计过程



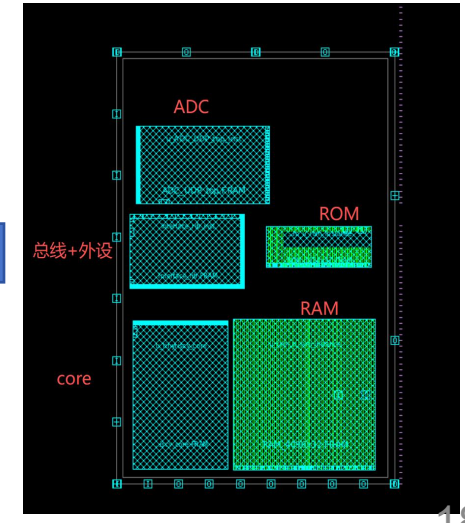


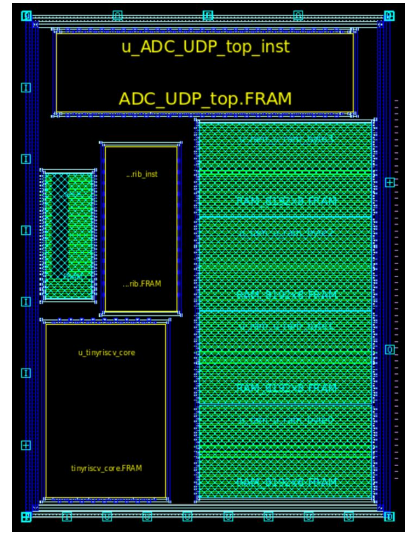


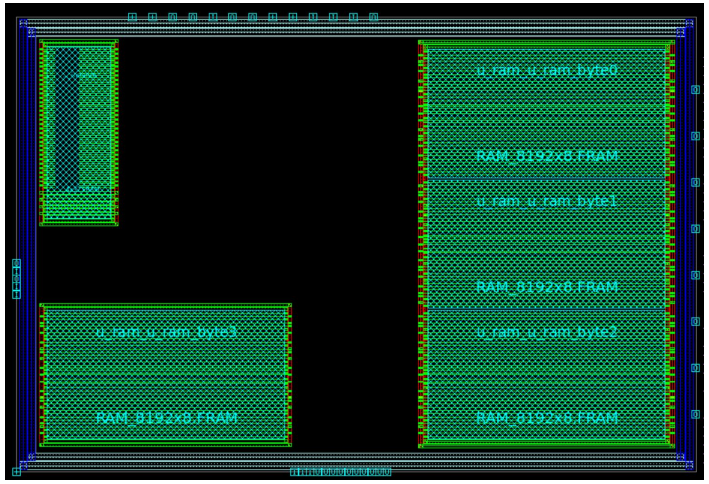


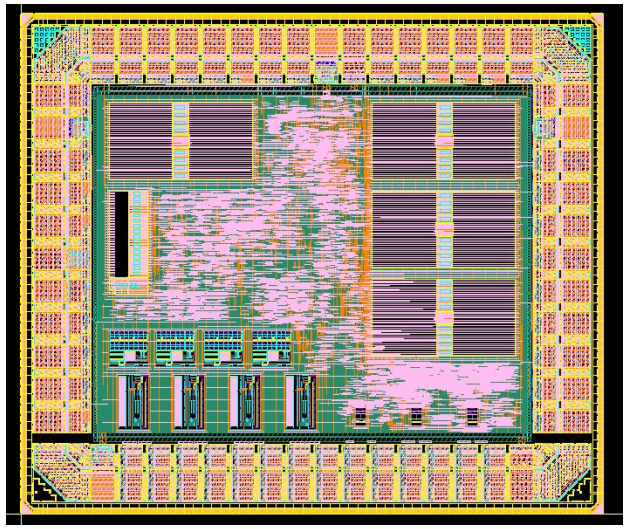
V1版图 700*700 um

.....



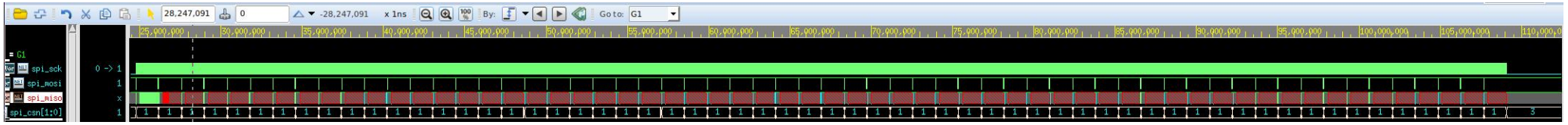




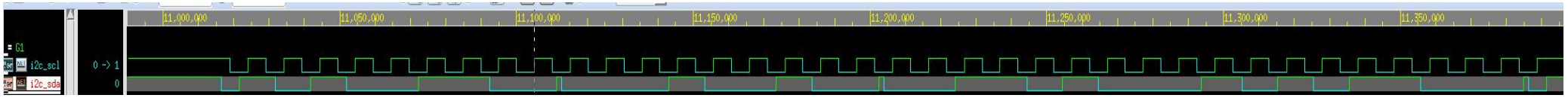


外设后仿

- SPI后仿正常，MISO出现x态因为SD程序只使用了32KB中的一部分，其余未使用部分会出现x



- I2C: 添加AT24C32仿真模型，后仿输出正常

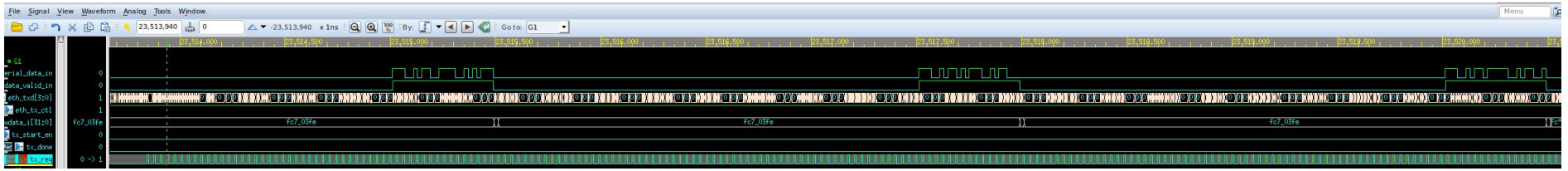


- Timer: 计时器后仿结果正常

```
===== Starting Timer Module Test Suite =====
Running Test: Basic Interrupt...
  -> PASS
Running Test: Register Read/Write...
  -> PASS
Running Test: Counter Increment...
  -> PASS
Running Test: Polling Mode...
  -> PASS
Running Test: Stop and Reset...
  -> PASS
Running Test: Byte Enables...
  -> WARNING: Byte-write to timer registers is not supported by hardware. Test skipped.
  -> PASS
Running Test: Boundary Value (VALUE=0)...
  -> PASS
===== Test Suite Finished =====
Result: 7 / 7 tests passed.
ALL TESTS PASSED!
```

- ADC传输模块仿真结果正常

```
*Verdi* : Create FSDB file 'tiny.fsdb'
*Verdi* : Begin traversing the scope (tinyriscv_soc_tb_sd), layer (0).
*Verdi* : End of traversing.
*Verdi* : fsdbDumpon - All FSDB files at 0 ns.
[                200] Test starting...
[                200] --- Starting Split RAM Fast Load ---
[                200] Using base file path: source/testbench/my_adc/my_adc
[                200] --- Split RAM Fast Load Complete ---
[                200] Reset released. CPU and automated ADC test start.
[                200] UART Monitor: Started. Watching tx_pin.
[                200] UART Monitor: CLK_FREQUENCY=    50000000, BAUD_RATE=     115200

--- System Initialization ---
Board IP set to: 192.168.185.110 (0xC0A8B96E)
Destination IP set to: 192.168.185.240 (0xC0A8B9F0)
UDP Config Reg (0x10) written with: 0x00030400
UDP Config Reg (0x10) read back: 0x00030400
UDP Config Reg (0x10) written with: 0x00030400
UDP Config Reg (0x10) written with: 0x00030400
UDP Config Reg (0x10) written with: 0x00030400
UDP Config Reg (0x10) written with: 0[              35000200] Simulation finished after sending some packets.
$finish called from file "source/testbench/Auto_test.v", line 207.
$finish at simulation time             35000200
           V C S   S i m u l a t i o n   R e p o r t
Time: 35000200 ns
CPU Time:   1314.340 seconds;       Data structure size:  25.5Mb
Sat Oct 11 18:52:39 2025
CPU time: 4.425 seconds to compile + .464 seconds to elab + .408 seconds to link + 1314.384 seconds in simulation
Verdi KDB elaboration done and the database successfully generated: 0 error(s), 0 warning(s)
```



核心后仿批量测试

批量测试流程：

1、修改编译链接文件，根据当前SOC设计重新将.S源码编译为.bin文件

2、将不同指令的.bin文件分割为4个RAM子文件

3、批量加载并运行程序

4、根据源码特定的地址索引寻找验证数据的起始和终止地址逐一验证结果

➢ Core后仿：使用Riscv官方测试集

初步测试大多数指令都通过了测试，部分失败，

部分失败源于程序结束后，RAM在输入存在x时会清空所有内容，

选择合适运行时间，结果正常如下

```
 200] Using base file path: source/testbench/core_test/I-DELAY_SLOTS-01.elf
 200] --- Split RAM Fast Load Complete ---
 200] Reset released. CPU starts execution from ROM.
 200] UART Monitor: Started. Watching tx_pin.
 200] UART Monitor: CLK_FREQUENCY=   50000000, BAUD_RATE=     115200
 300] CPU halted signal detected. Starting verification.
3310] CPU halted signal detected!
3510] Signature range found in RAM: Start Addr=0x10002000, End Addr=0x10002020
```

```
 200] Using base file path: source/testbench/core_test/I-ENDIANESS-01.elf
 200] --- Split RAM Fast Load Complete ---
 200] Reset released. CPU starts execution from ROM.
 200] UART Monitor: Started. Watching tx_pin.
 200] UART Monitor: CLK_FREQUENCY=   50000000, BAUD_RATE=     115200
 300] CPU halted signal detected. Starting verification.
2310] CPU halted signal detected!
2510] Signature range found in RAM: Start Addr=0x10002010, End Addr=0x10002030
```

```
 1 I-ADD-01: PASSED
 2 I-ADDI-01: PASSED
 3 I-AND-01: PASSED
 4 I-ANDI-01: PASSED
 5 I-AUIPC-01: PASSED
 6 I-BEQ-01: PASSED
 7 I-BGE-01: PASSED
 8 I-BGEU-01: PASSED
 9 I-BLT-01: PASSED
10 I-BLTU-01: PASSED
11 I-BNE-01: PASSED
12 I-DELAY_SLOTS-01: FAILED (Signature mismatch)
13 I-EBREAK-01: PASSED
14 I-ECALL-01: PASSED
15 I-ENDIANESS-01: FAILED (Signature mismatch)
16 I-IO-01: PASSED
17 I-JAL-01: PASSED
18 I-JALR-01: PASSED
19 I-LB-01: PASSED
20 I-LBU-01: PASSED
21 I-LH-01: PASSED
22 I-LHU-01: PASSED
23 I-LUI-01: PASSED
24 I-LW-01: PASSED
25 I-MISALIGN_JMP-01: FAILED (Signature mismatch)
26 I-MISALIGN_LDST-01: FAILED (Signature mismatch)
27 I-NOP-01: PASSED
28 I-OR-01: PASSED
29 I-ORI-01: PASSED
30 I-RF_size-01: PASSED
31 I-RF_width-01: PASSED
32 I-RF_x0-01: PASSED
33 I-SB-01: PASSED
34 I-SH-01: PASSED
35 I-SLL-01: PASSED
36 I-SLLI-01: PASSED
37 I-SLT-01: PASSED
38 I-SLTI-01: PASSED
39 I-SLTIU-01: PASSED
40 I-SLTU-01: PASSED
41 I-SRA-01: PASSED
42 I-SRAI-01: PASSED
43 I-SRL-01: PASSED
44 I-SRLI-01: PASSED
45 I-SUB-01: PASSED
46 I-SW-01: PASSED
47 I-XOR-01: PASSED
48 I-XORI-01: PASSED
```

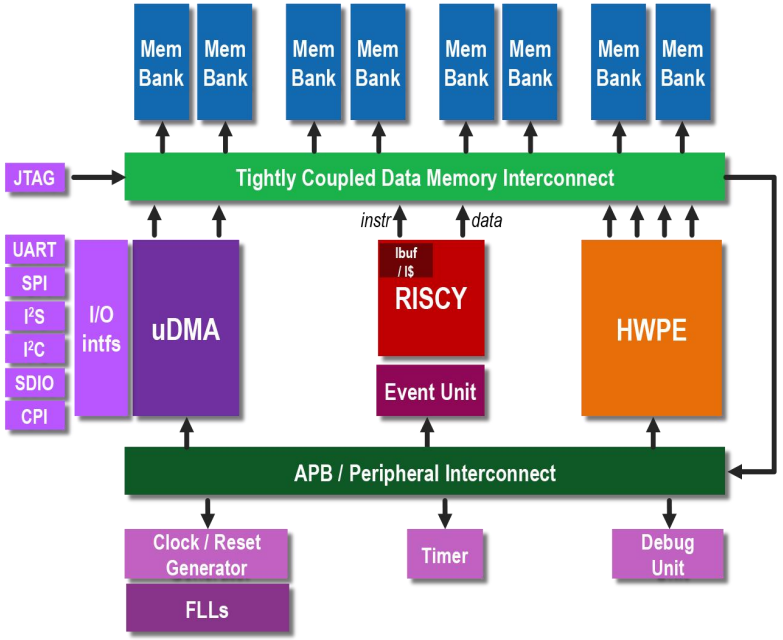
一、背景简介

二、研发历程

三、后续规划

后续规划

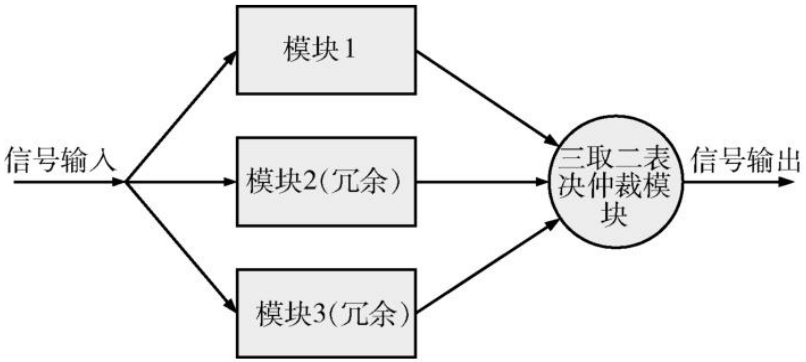
➢ 更换SOC架构，使用PULP平台的pulpissimo (已在FPGA上正常运行并通过部分程序测试)



- 可任意切换RI5CY和IBEX
- 自主输入/输出子系统（uDMA）
- 新型内存子系统
- 支持硬件处理引擎（HWPEs）
- 新型简单中断控制器
- 丰富的外设和配套的SDK
- 内外部多级总线配置

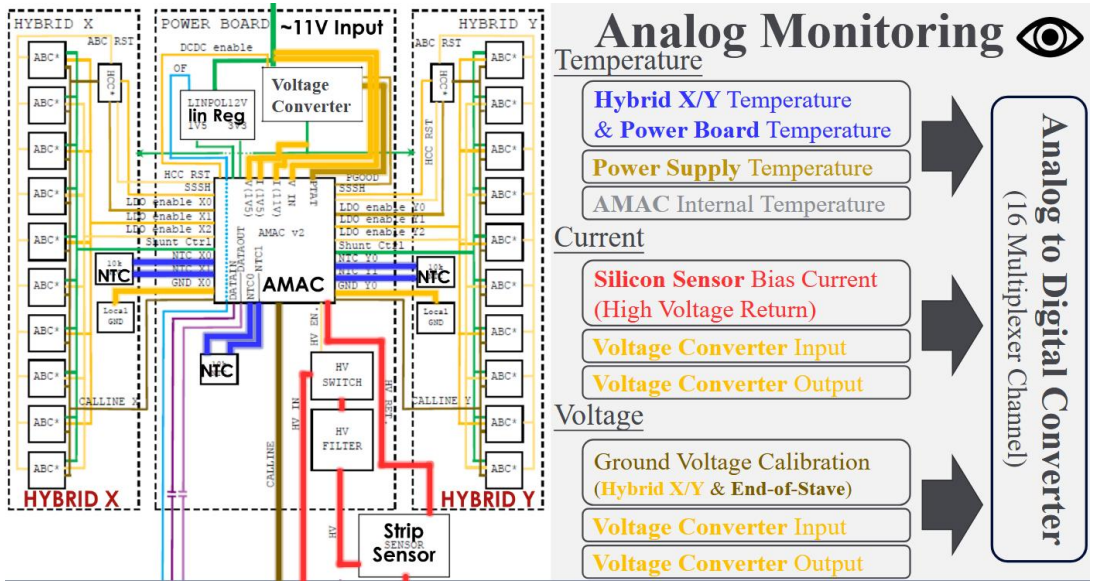
➢ 抗辐照设计

- 时间冗余 + 重要模块三模冗余 + ECC纠错
- LockStep + ECC
- 完整三模冗余



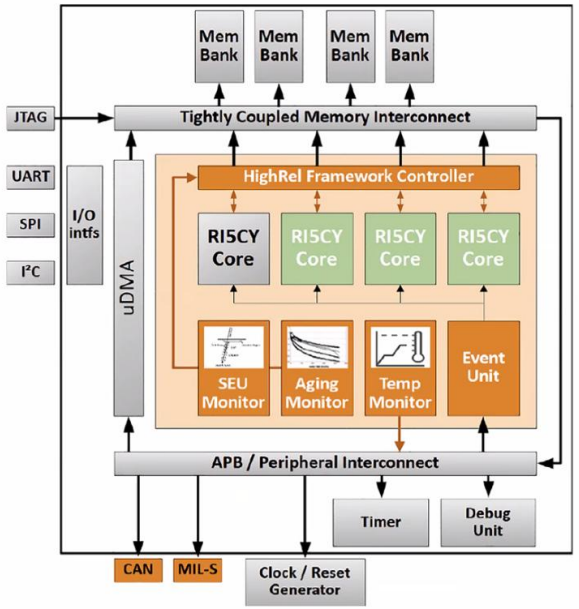
后续规划

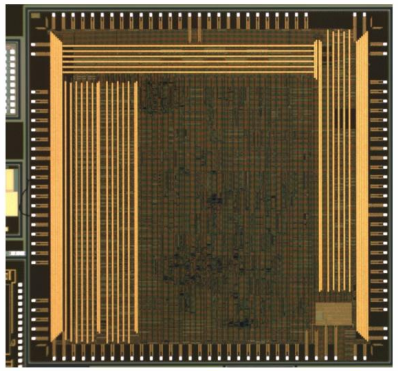
- 近期目标：设计测试版，等待回片，实现类似AMAC、TETRISC的温度、电压等监控，同时和LATRIC整合实现动态配置例如DAC



AMAC by ATLAS

https://doi.org/10.1088/1748-0221/20/08/P08003





Chip area	43,56 mm^2 (6.6*6.6mm)
Nominal clock frequency	30MHz
Power consumption	<1W (estimated)
Memory	4*8192*40 Bit SRAM
Pads	81 signal, 35 other

TETRISC SOC by IHP

https://doi.org/10.1016/j.microrel.2023.115173

- 远期目标：利用RISC-V的开放与可定制性，打造专为高能物理（HEP）实验设计的端侧智能数据处理平台。未来帮助高能所设计的每块ASIC集成一颗“智能大脑”