

# First High-Energy RISC-V Instrumentation SOC (HERIS) for CEPC Readout ASICs

---

**Reporter:** Cui Yuxin

**Team:** Yan Qi, Chen Jiaolong, Luo Shoudong, Wang Hanwen, Zhang Yihan

On behalf of the CEPC Silicon Tracker & the HERIS Group

➤ **Introduction**

➤ **R&D Process**

➤ **Future Plan**

# Reconfigurable Intelligent SoC (System On Chip)

## Challenges

- High cost and complexity of advanced process nodes
  - Expensive tape-out iterations for advanced nodes
  - Requires first-pass success, and the verification is also highly challenging.
- Limitations of traditional ASICs
  - Fixed functionality, weak configurability
  - Difficult to integrate on-chip algorithms



## New Design Paradigm

- Introducing RISC-V based SoC architecture for hardware-software co-design
  - Programmability: Adapts to multiple applications via software configuration
  - Modularity & IP Reuse: Enhances design collaboration and efficiency
  - Can be integrated into existing ASIC chips

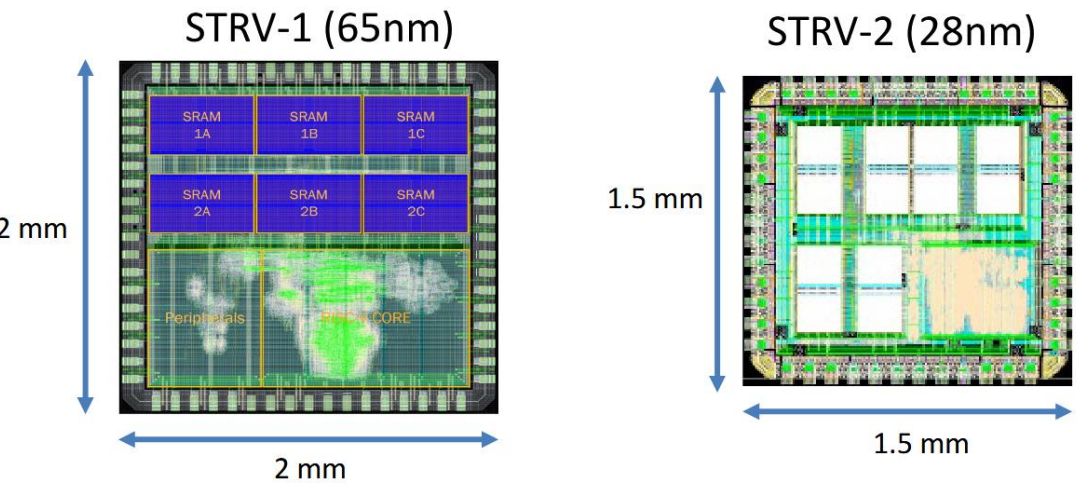
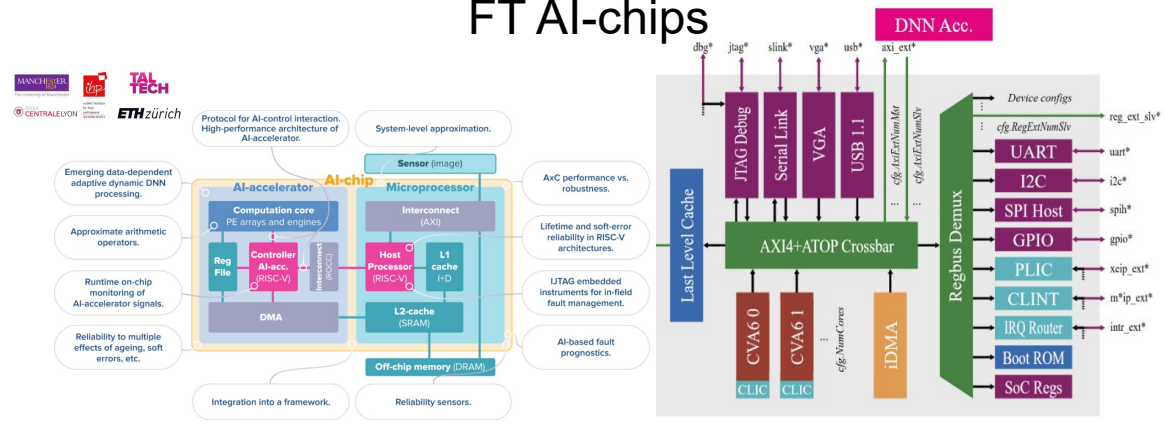
**Intelligent SoC is the future trend for ASIC development in HEP**

# RISC-V in High Energy Physics

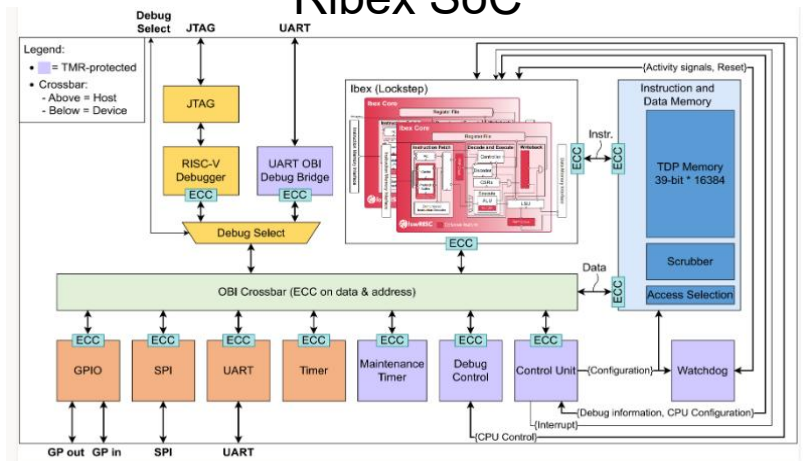
To address future HEP experiment needs, CERN's RISC-V application focuses on two core directions:

- Building highly reliable monitoring and control systems
- Developing intelligent on-chip data processing

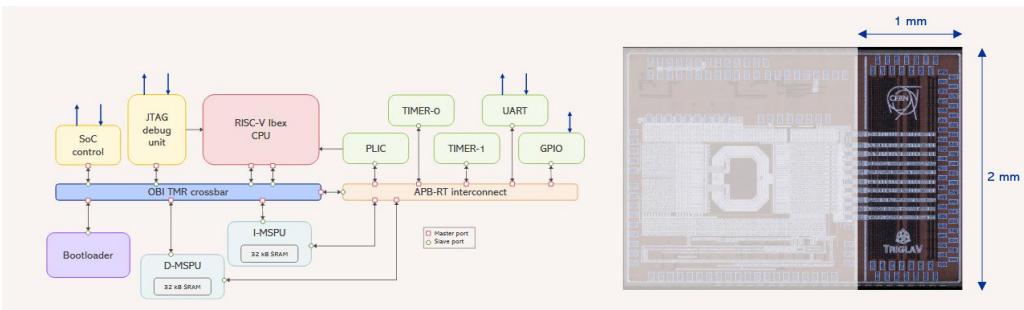
## FT AI-chips



## Ribex SoC

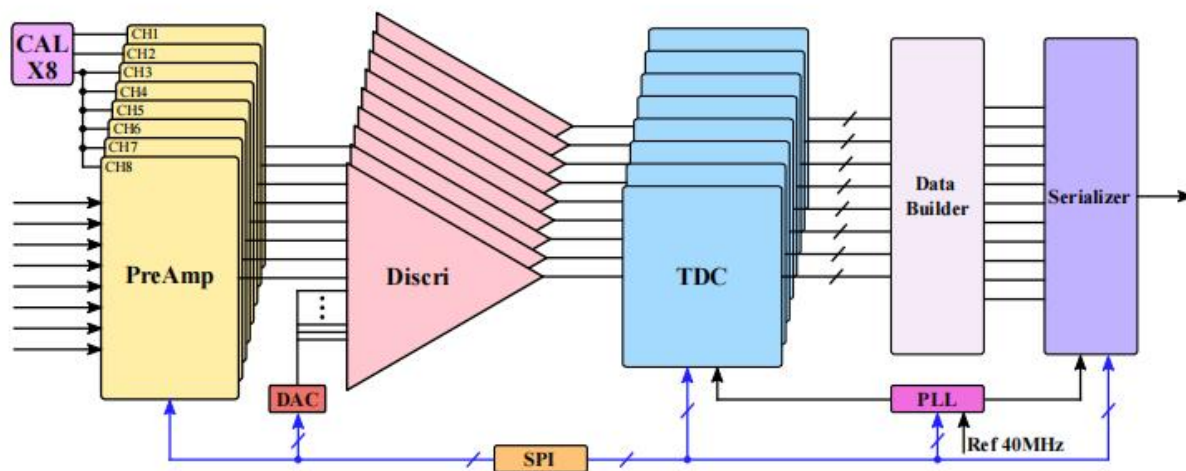


## TriglaV

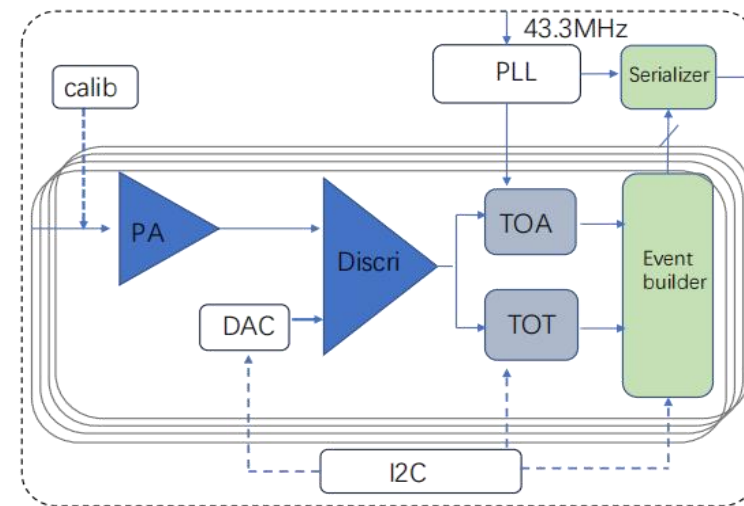


# RISC-V Applied to LATRIC

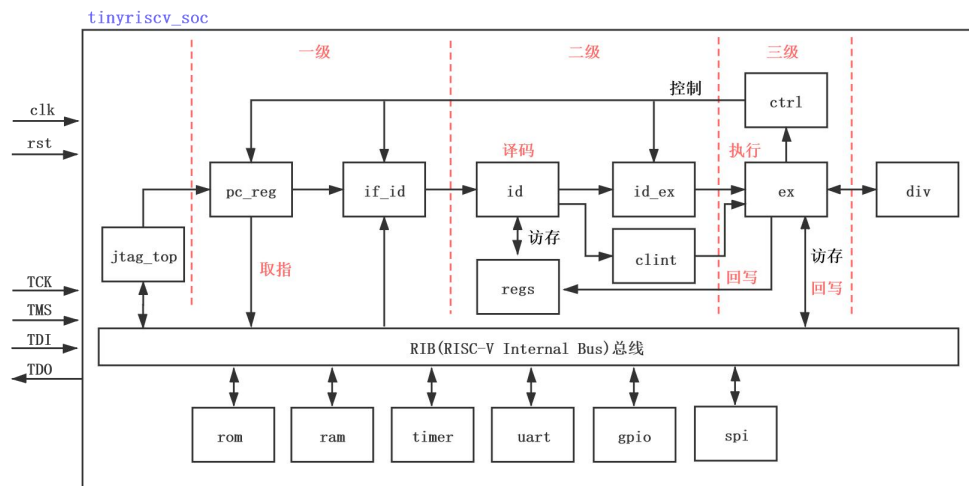
At the initial research stage, we selected the beginner-friendly Tiny-RiscV soft core, implementing a dynamically configurable DAC module and other algorithms on OTK.



Schematic FPMROC



Schematic LATRIC



Schematic Tiny-RiscV

- Supports RV32IM instruction set
- Employs a three-stage pipeline (Fetch, Decode, Execute).
- Capable of running C programs
- Supports JTAG for online program updates via OpenOCD.
- Supports interrupts、FreeRTOS
- Easily portable to any FPGA platform.

**一、 Background Introduction**

**二、 R&D Process**

**三、 Future Plan**



# Tiny-RiscV Architecture Diagram

## HERIS

### ➤ Core (Tiny RiscV)

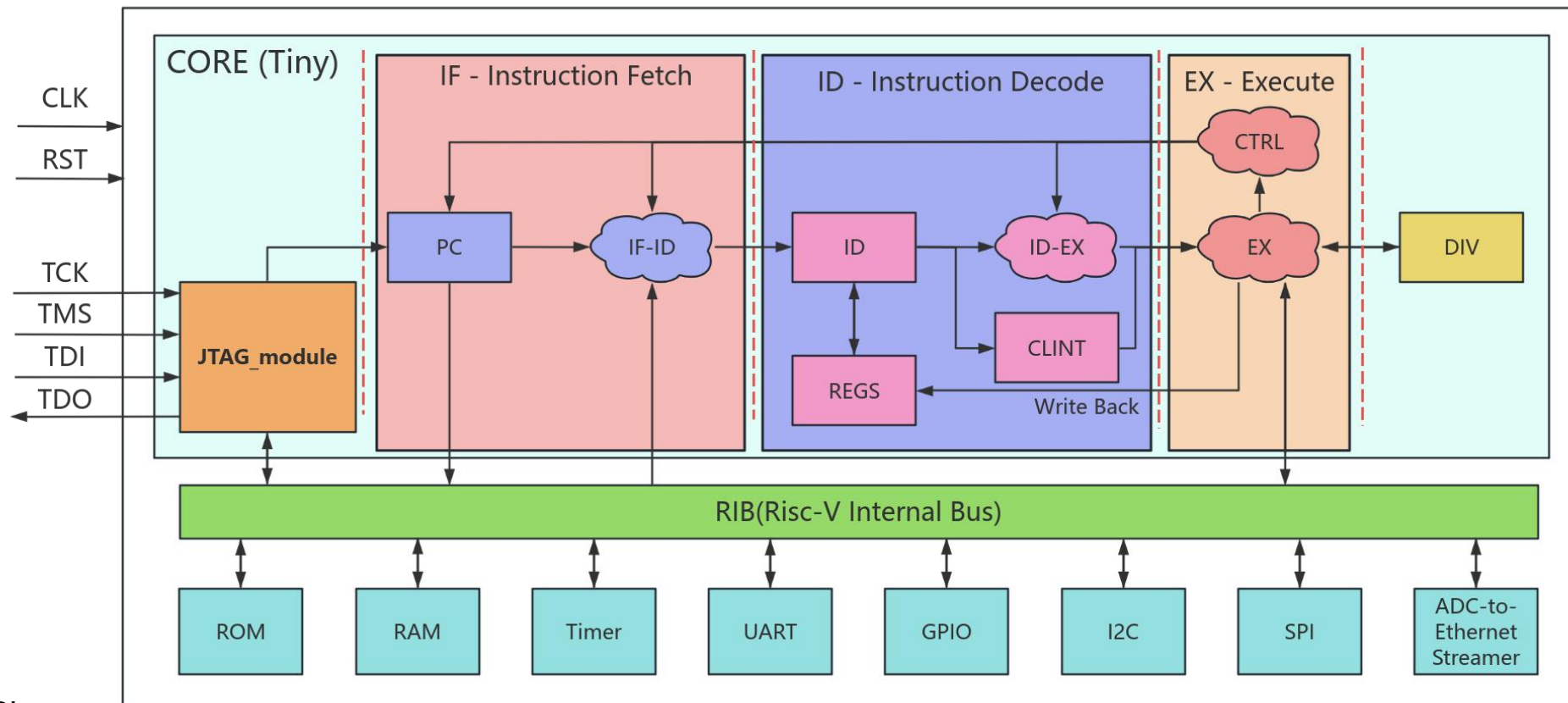
- RV32IM Core
- 3 stage pipeline
- CoreMark/MHz = 2.4

### ➤ JTAG Interface

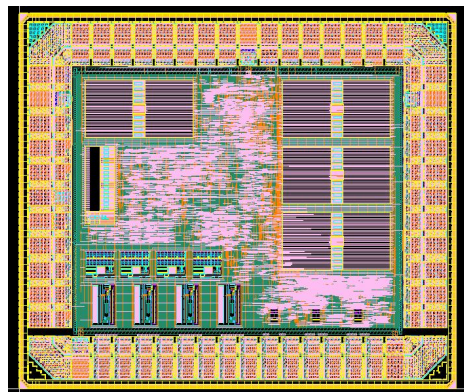
- OpenOCD support
- GDB debugging support

### ➤ Peripherals

- 4 KB ROM, 32 KB RAM
- Supports multiple serial communication protocols, such as I2C, UART, and SPI
- Integrated sensor data processing and UDP forwarding



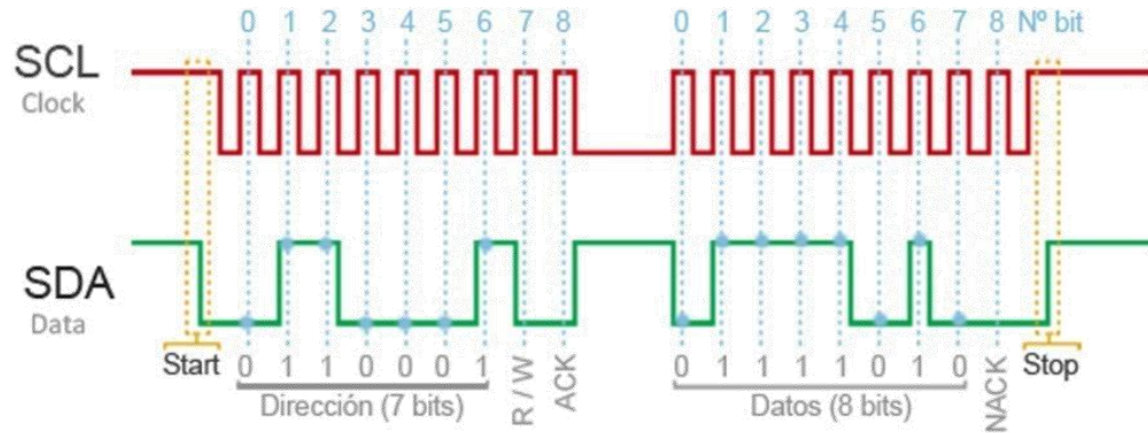
## HERIS-V1



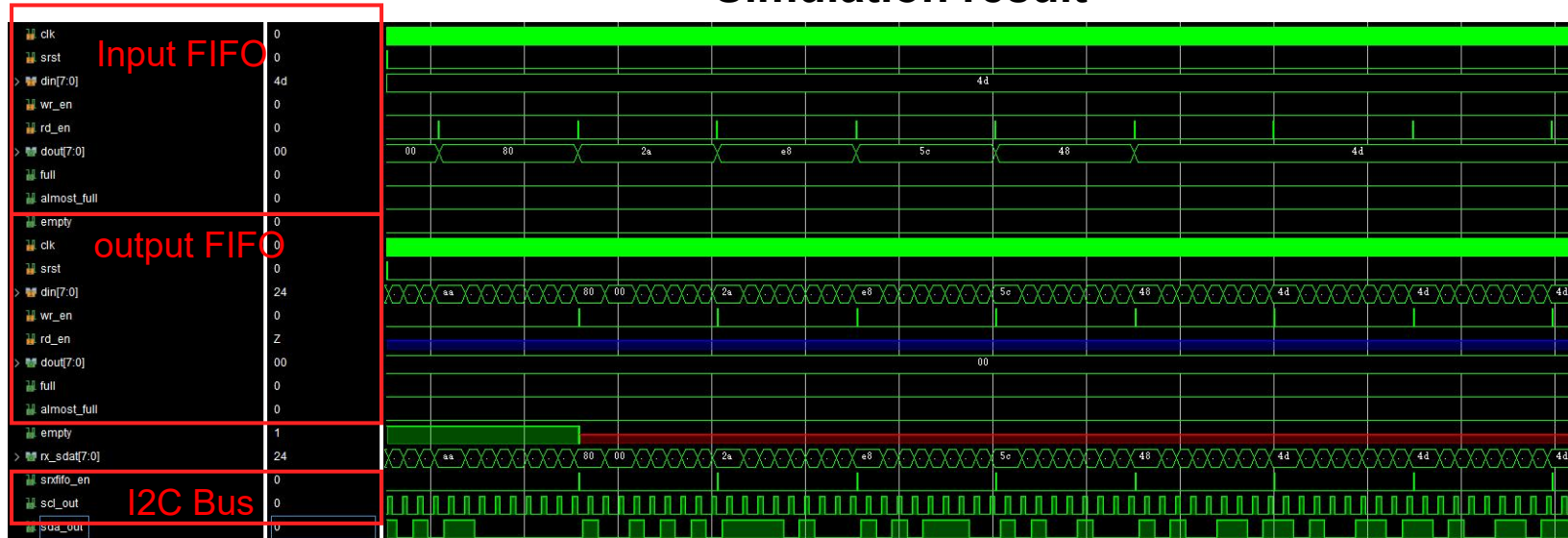
- 55 nm process technology
- Frequency 50 MHz
- Size 1020 x 1196 um
- Design verification completed and submitted for tape-out in October
- Testing is scheduled to begin after the chips return, expected in January 2026

# Adding I2C Master Module

- Selected a mature I2C host from Opencores (<https://opencores.org>).
- Wrapped the host with an interface layer and mounted it on the bus.



Simulation result





# I2C Module FPGA Verification

Used the onboard AT24C64 EEPROM memory chip for functional validation.

## C Program

```

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

// =====
// == The CRITICAL: Base address updated to the correct value ==
// =====
#define IC2_MASTER_BASE_ADDR 0x50000000

#define IC2_REG_PRESCALE 0x00
#define IC2_REG_CMD 0x04
#define IC2_REG_SLAVE_ADDR 0x08
#define IC2_REG_MEM_ADDR 0x0C
#define IC2_REG_DATA 0x10
#define IC2_REG_STATUS 0x20

#define IC2_PRESCALE (*(volatile uint32_t *)IC2_MASTER_BASE_ADDR + IC2_REG_PRESCALE)
#define IC2_CMD (*(volatile uint32_t *)IC2_MASTER_BASE_ADDR + IC2_REG_CMD)
#define IC2_SLAVE_ADDR (*(volatile uint32_t *)IC2_MASTER_BASE_ADDR + IC2_REG_SLAVE_ADDR)
#define IC2_MEM_ADDR (*(volatile uint32_t *)IC2_MASTER_BASE_ADDR + IC2_REG_MEM_ADDR)
#define IC2_DATA (*(volatile uint32_t *)IC2_MASTER_BASE_ADDR + IC2_REG_DATA)
#define IC2_STATUS (*(volatile uint32_t *)IC2_MASTER_BASE_ADDR + IC2_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 ic2_init(uint32_t fclk, uint32_t f_scl) {
    while(IC2_STATUS & STATUS_BUSY) {}
    IC2_PRESCALE = fclk / f_scl;
}

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

void eeprom_write_byte(uint16_t mem_addr, uint8_t data) {
    xprintf("[IC2] Write: Addr=0x%04X Data=0x%02X\n", mem_addr, data);
    while(IC2_STATUS & STATUS_BUSY) {}

    // Phase 1: Configure
    IC2_SLAVE_ADDR = EPROM_SLAVE_ADDR;
    IC2_MEM_ADDR = mem_addr;
    IC2_DATA = data;

    // Optional but recommended: A quick feedback to confirm writes were received
    xprintf("== Config Check: Slave=0x%04X, Addr=0x%04X, Data=0x%02X\n", (uint16_t)IC2_SLAVE_ADDR, (uint16_t)IC2_MEM_ADDR, (uint8_t)IC2_DATA);

    // Phase 2: Execute
    IC2_CMD = CMD_GO | CMD_WRITE;

    while(IC2_STATUS & STATUS_BUSY) {}
    if (IC2_STATUS & STATUS_ACK_ERROR) {
        xprintf("[IC2] ERROR: NACK detected during writing\n");
    }
}

uint16_t eeprom_read_byte(uint16_t mem_addr) {
    xprintf("[IC2] Read: Addr=0x%04X\n", mem_addr);
    while(IC2_STATUS & STATUS_BUSY) {}

    // Phase 1: Configure
    IC2_SLAVE_ADDR = EPROM_SLAVE_ADDR;
    IC2_MEM_ADDR = mem_addr;

    // Phase 2: Execute
    IC2_CMD = CMD_GO | CMD_READ;

    while(IC2_STATUS & STATUS_BUSY) {}
    if (IC2_STATUS & STATUS_ACK_ERROR) {
        xprintf("[IC2] ERROR: NACK detected during reading\n");
        return 0xFF;
    }
    return (uint16_t)IC2_DATA;
}

int main() {
    uart_init();
    xprintf("===== IC2 Master Final Test Program =====\n");
    IC2_init(50000000, 1000000);

    uint16_t test_addr = 0x1234;
    uint8_t test_data = 0xA5C;

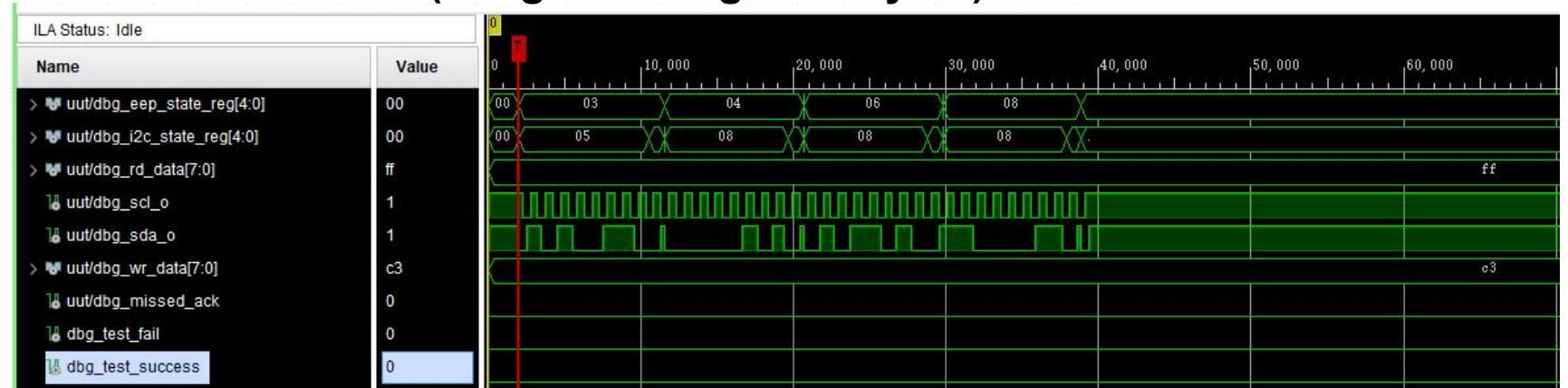
    eeprom_write_byte(test_addr, test_data);
    uint16_t read_back = eeprom_read_byte(test_addr);

    xprintf("[IC2] Verification: Wrote 0x%02X, Read back 0x%02X\n", test_data, read_back);
    if (read_back == test_data) {
        xprintf("[IC2] SUCCESS!\n");
    } else {
        xprintf("[IC2] FAILURE!\n");
    }

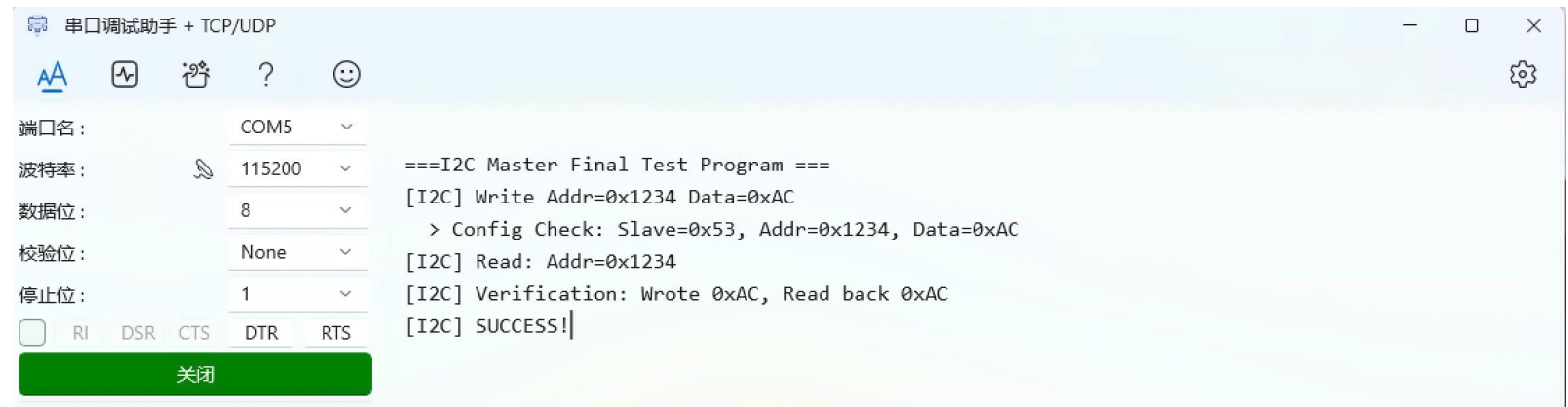
    while(1) {}
    return 0;
}

```

## ILA(Integrated Logic Analyzer) results



## UART output results



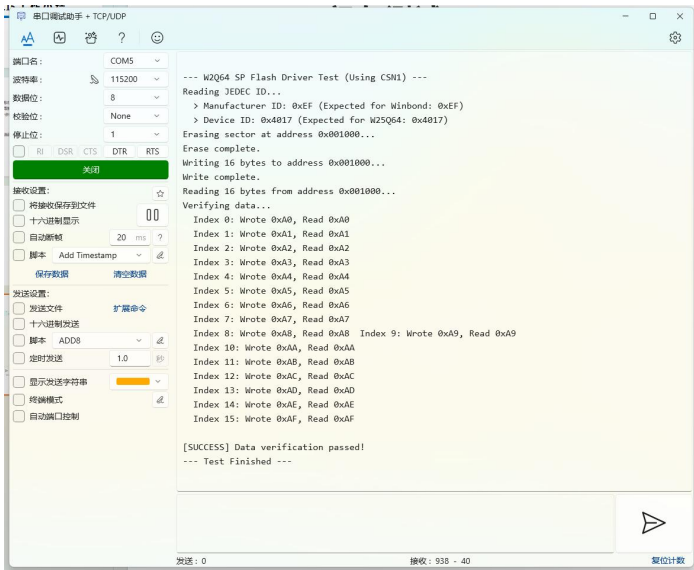
# Optimizing & Testing SPI Master Module

- Replaced the original SPI module with a more mature SPI Master, and added an interface wrapper
- Successfully verified module functionality on FPGA using the W25Q64 SPI Flash memory chip

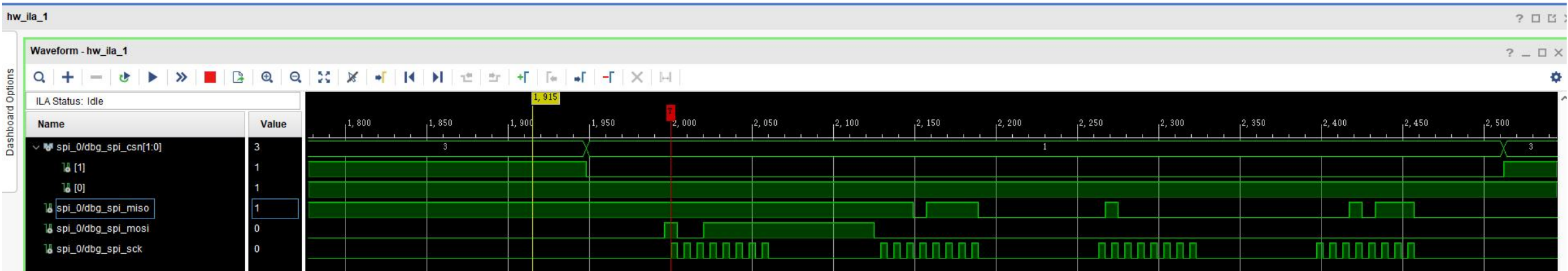
W25Q64



UART output results

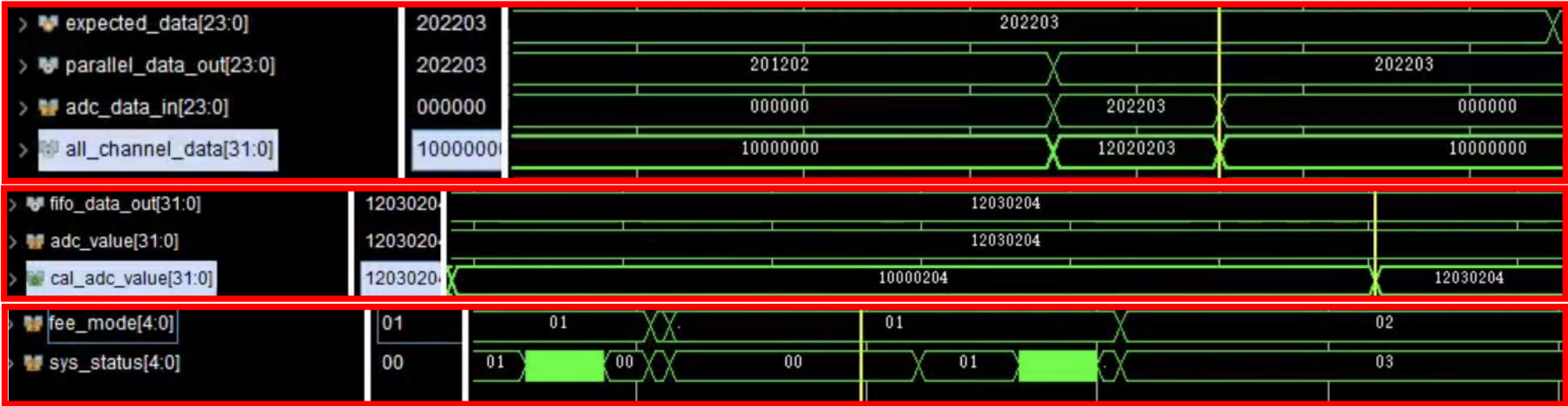


ILA(Integrated Logic Analyzer) results



# Adding ADC-Ethernet Module

Receives serial input data, processes it, and transmits it to the host PC via UDP protocol



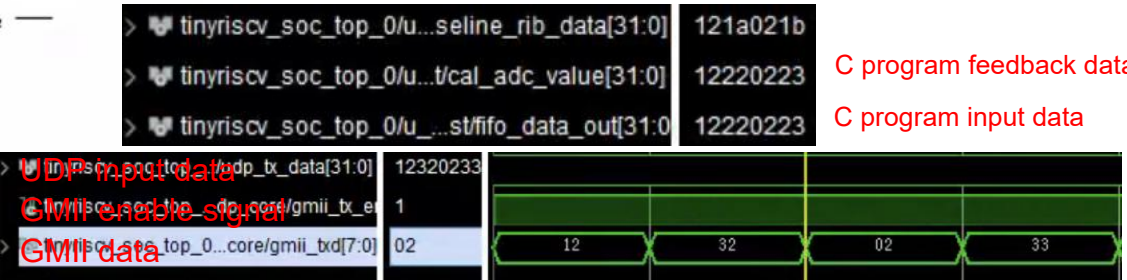
- ADC Serial Input Data
- Parallel Data
- ADC Received Data
- FIFO Output Data
- C Program Input Data
- C Program Feedback Data
- Operating Mode Switch
- System State Switch

## On-board Validation

```
— System Initialization —
Board IP set to: 192.168.185.111 (0xCOA8B96F)
Board port set to: 1234
Destination IP set to: 192.168.185.43 (0xCOA8B9F3)
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
```

## Data reading

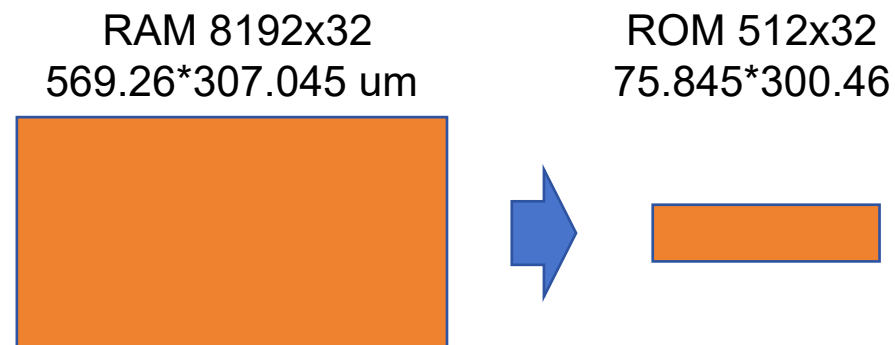
```
— Measuring Baseline Voltage —
Signal@CH0: 516
Signal@CH1: 515
Signal@CH0: 517
Signal@CH1: 516
Signal@CH0: 518
Signal@CH1: 517
Signal@CH0: 519
Signal@CH1: 518
Signal@CH0: 520
Signal@CH1: 519
```



# Optimizing Boot Method

Tiny-RiscV (original using Block RAM as 32KB ROM)

- Boot only via JTAG, Lacks self-start capability, poor reliability and convenience.
- Large ROM area overhead



Output from SD card test

New Solution: Use SPI-based SD Card as external storage

- ROM contains bootloader (reduced from 32KB to 4KB)
- On power-up, the bootloader loads the program from the SD card

```
串口调试助手 + TCP/UDP

SPI Test
SD Card on SPI CH-1 initialized successfully.

--- Running SD Card Test (SPI) ---

1. Writing 512 bytes to block address 1000...
  > Write complete.
2. Reading 512 bytes from block address 1000...
  > Read complete.

write data is:05, read data: 05
write data is:06, read data: 06
write data is:07, read data: 07
write data is:08, read data: 08
write data is:09, read data: 09
write data is:0A, read data: 0A
write data is:0B, read data: 0B
write data is:0C, read data: 0C
write data is:0D, read data: 0D
write data is:0E, read data: 0E
```



# Boot-Loader Design

- Current RAM size: 32KB (30KB available for programs, top 2KB reserved for bootloader variables).
- Added UART output for debugging.

## Linker Script

```
/*
 * 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 (wx!r!)   : ORIGIN = 0x10000000, LENGTH = RAM_TOTAL_LENGTH - BOOTLOADER_RAM_SIZE

    /* RAM reserved for bootloader's private use. Its origin is calculated directly. */
    boot_ram (wx!r!) : 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);
}
```

## Startup Assembly File

```
1 // bootloader/bootloader_start.S
2 .section .init
3 .globl _start_bootloader
4
5 _start_bootloader:
6     // 1. Initialize the stack pointer to the top of the main RAM.
7     // The linker script provides the '_stack_start' symbol.
8     la sp, _stack_start
9
10    // Optional: Clear the .bss section for the bootloader.
11    // For a minimal bootloader, this step can often be skipped if you know
12    // your variables don't rely on being zero-initialized. But it's good practice.
13    la t0, __bss_start
14    la t1, __bss_end
15    .L_clear_bss_boot:
16    beq t0, t1, .L_clear_bss_boot_done
17    sw zero, (t0)
18    addi t0, t0, 4
19    j .L_clear_bss_boot
20    .L_clear_bss_boot_done:
21
22    // 2. Jump to the C entry point of the bootloader
23    jal ra, boot_main_c
24
25    .L_hang:
26    // If boot_main_c returns, hang here.
27    j .L_hang
```

## Boot program

```
9 // ... (配置宏定义保持不变) ...
10 #define APP_START_BLOCK_ON_SD_CARD 0 // 使用方案A, 从块0读取
11 #define APP_RUN_ADDR_ON_CHIP_RAM 0x10000000
12 #define ON_CHIP_RAM_SIZE_BYTES (14 * 1024)
13 #define MY_SD_CARD_SPI_CHANNEL 1
14
15
16 typedef void (*app_entry_t)(void);
17 const app_entry_t app_entry = (app_entry_t)APP_RUN_ADDR_ON_CHIP_RAM;
18
19 void boot_main_c() {
20     // STEP 1: 初始化UART, 这是我们唯一的调试窗口
21     // 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。
22     uart_init();
23
24     // 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
25     DEBUG_PRINTF("\n\n--- RISC-V Bootloader Started ---\n");
26
27     // STEP 2: 初始化SD卡
28     DEBUG_PRINTF("Initializing SD card on SPI channel %d...\n", MY_SD_CARD_SPI_CHANNEL);
29     if (sd_card_init(MY_SD_CARD_SPI_CHANNEL) != 0) {
30         DEBUG_PRINTF("!!! FATAL: SD Card initialization failed. Halting. !!!\n");
31         while(1); // 如果失败, 打印信息并死循环
32     }
33     DEBUG_PRINTF("SD Card initialized successfully.\n");
34
35     // STEP 3: 计算需要读取的块数
36     uint32_t num_blocks_to_read = (ON_CHIP_RAM_SIZE_BYTES + SD_BLOCK_SIZE - 1) / SD_BLOCK_SIZE;
37     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);
38
39     // STEP 4: 循环拷贝
40     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);
41     for (uint32_t i = 0; i < num_blocks_to_read; i++) {
42         uint32_t source_block = APP_START_BLOCK_ON_SD_CARD + i;
43         uint8_t* destination_address = (uint8_t*)APP_RUN_ADDR_ON_CHIP_RAM + (i * SD_BLOCK_SIZE);
44
45         // 可以在这里加一个简单的进度指示
46         if ((i % 4) == 0) { // 每4个块打印一个点
47             DEBUG_PRINTF(".");
48         }
49
50         if (sd_card_read_block(source_block, destination_address) != 0) {
51             DEBUG_PRINTF("\n!!! FATAL: Failed to read block %d. Halting. !!!\n", source_block);
52             while(1); // 读取失败, 打印信息并死循环
53         }
54     }
55     DEBUG_PRINTF("\nData copy complete.\n");
56
57     // STEP 5: 准备跳转
58     DEBUG_PRINTF("Bootloader finished. Jumping to application at 0x%x...\n", app_entry);
59
60     // 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
61     for(volatile int i=0; i<10000; ++i);
62
63     // 跳转
64     app_entry();
65
66     // 如果程序能执行到这里, 说明跳转失败了
67     DEBUG_PRINTF("!!! FATAL: Application returned to bootloader. Halting. !!!\n");
68     while(1);
69 }
```



# Hardware Code Modifications for Bootloader

1、Fetch Module: Controlled by internal signals rst\_n, flush\_i, stall. No direct inputs from external debug modules

modules

```
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 Module Input: In jtag\_dm (Debug Module), based on RISC-V Debug Spec. Modified behavior when writing to dpc (Debug Program Counter) CSR.

```
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;
        jtag_pc_wdata <= data0;
        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
end
```

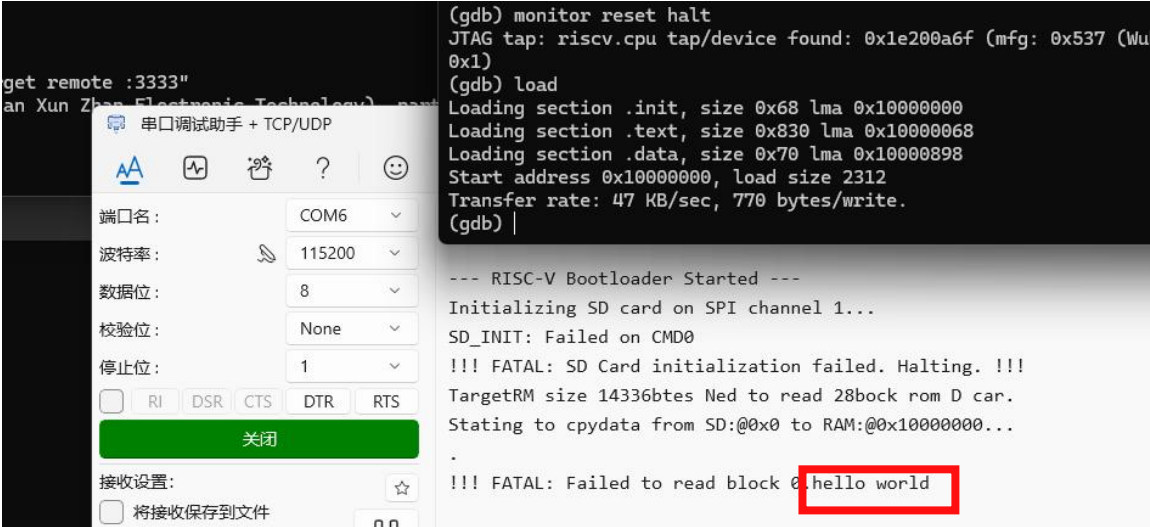
When JTAG attempts to write to DPC, the current hardware's response is not to update the CPU's PC, but rather to assert a reset request signal

3、Reused existing flush mechanism (which flushes pipeline and restarts fetch from new address). JTAG module "borrows" this mechanism to update PC.

```
// When JTAG writes to PC, use the new address provided by JTAG; otherwise, use the normal jump address.
assign flush_addr_o = jtag_pc_we_i ? jtag_pc_wdata_i : jump_addr_i;
```

```
// Flush the pipeline on a normal jump, a CLINT stall, or a JTAG PC write.
assign flush_o = jump_assert_i | stall_from_clint_i | jtag_pc_we_i;
```

Post-modification: Normal startup via GDB is successful



The screenshot displays a GDB terminal window on the right and a serial port configuration window on the left. The GDB terminal shows the following output:

```
(gdb) monitor reset halt
JTAG tap: riscv.cpu tap/device found: 0x1e200a6f (mfg: 0x537 (Wu
0x1)
(gdb) load
Loading section .init, size 0x68 lma 0x10000000
Loading section .text, size 0x830 lma 0x100000068
Loading section .data, size 0x70 lma 0x100000898
Start address 0x10000000, load size 2312
Transfer rate: 47 KB/sec, 770 bytes/write.
(gdb) |

--- RISC-V Bootloader Started ---
Initializing SD card on SPI channel 1...
SD_INIT: Failed on CMD0
!!! FATAL: SD Card initialization failed. Halting. !!!
TargetRM size 14336btes Ned to read 28bock rom D car.
Stating to cpydata from SD:@0x0 to RAM:@0x10000000...
.
!!! FATAL: Failed to read block 0,hello world
```

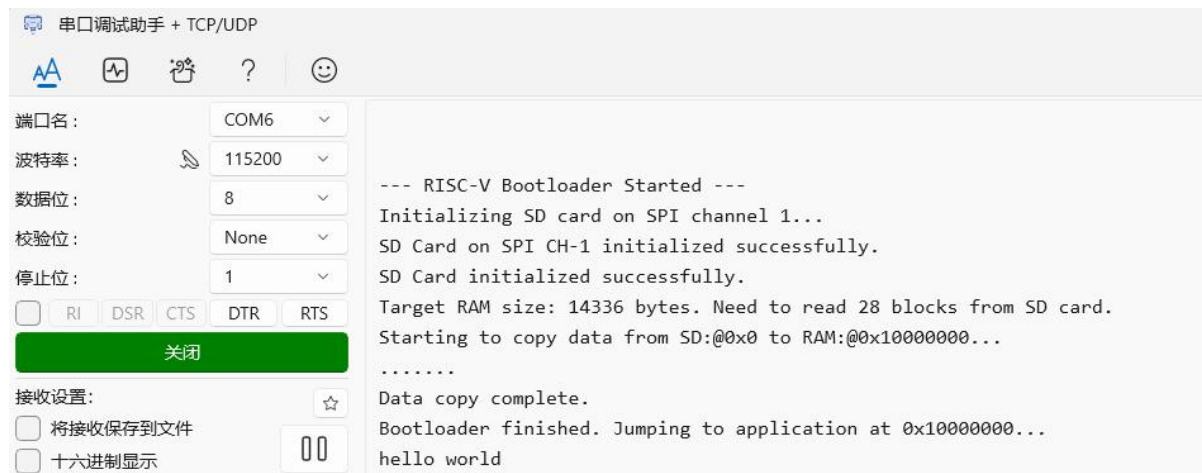
The serial port configuration window on the left shows the following settings:

- 端口名: COM6
- 波特率: 115200
- 数据位: 8
- 校验位: None
- 停止位: 1
- 接收设置: 将接收保存到文件

# C Program Modifications & FPGA Validation

C programs compiled using lds linker script and start\_s script, configured for the bootloader.

## Method 1: ROM + SD Card



Debug output messages from successful ROM + SD Card boot shown

## Method 2: GDB Debug Startup

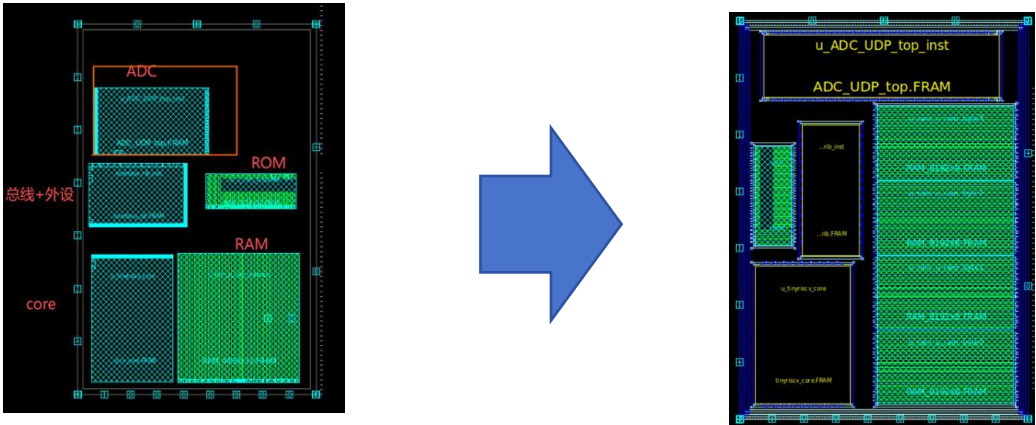
```
# file: debug.gdb
target remote localhost:3333
monitor reset halt
load
set $pc = 0x10000000
b main
c
```

```
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 Modification

Modified RAM architecture to implement byte-select functionality

- Changed 32-bit RAM to 4 combined 8-bit RAMs.
- Added a Verilog model for SD card simulation for pre- and post-layout verification.



```
Tcl Console x Messages Log
[ 6734110000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6754010000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6775270000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6795170000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6816430000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6836330000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6857590000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6877490000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6898750000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6918650000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6939910000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6959810000] SD_MODEL: R1 response sent, returning to IDLE.
SD Card on SPI CH-1 initialized successfully.
SD Card initialized successfully.
Target RAM size: 14336 bytes. Need to read 28 blocks from SD card.
Starting to copy data from SD:0x0 to RAM:0x10000000...
Data copy complete.
Bootloader finished. Jumping to application at 0x10000000...
hello world
run_time (s): cpu = 00:00:13 ; elapsed = 00:02:22 ; Memory (MB): peak = 5174.727 ; gain = 0.000
```

Pre-Simulation

```
串口调试助手 + TCP/UDP
端口名: COM6
波特率: 115200
数据位: 8
校验位: None
停止位: 1
[ ] RI [ ] DSR [ ] CTS [ ] DTR [ ] RTS
[ 关闭 ]
接收设置:
[ ] 将接收保存到文件
[ ] 十六进制显示
--- RISC-V Bootloader Started ---
Initializing SD card on SPI channel 1...
SD Card on SPI CH-1 initialized successfully.
SD Card initialized successfully.
Target RAM size: 14336 bytes. Need to read 28 blocks from SD card.
Starting to copy data from SD:0x0 to RAM:0x10000000...
Data copy complete.
Bootloader finished. Jumping to application at 0x10000000...
hello world
```

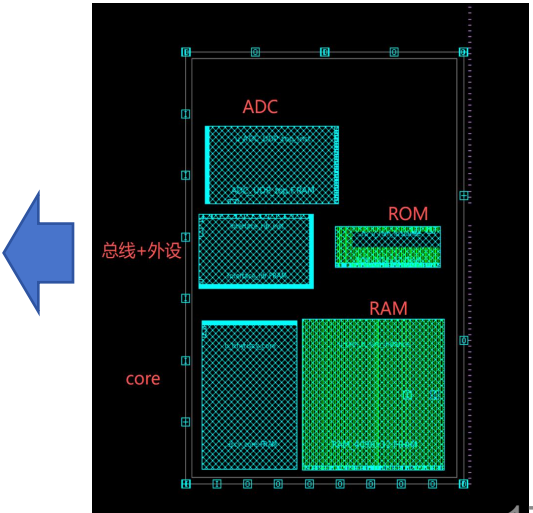
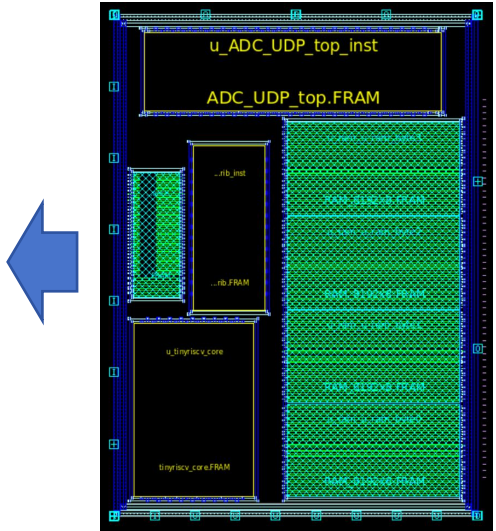
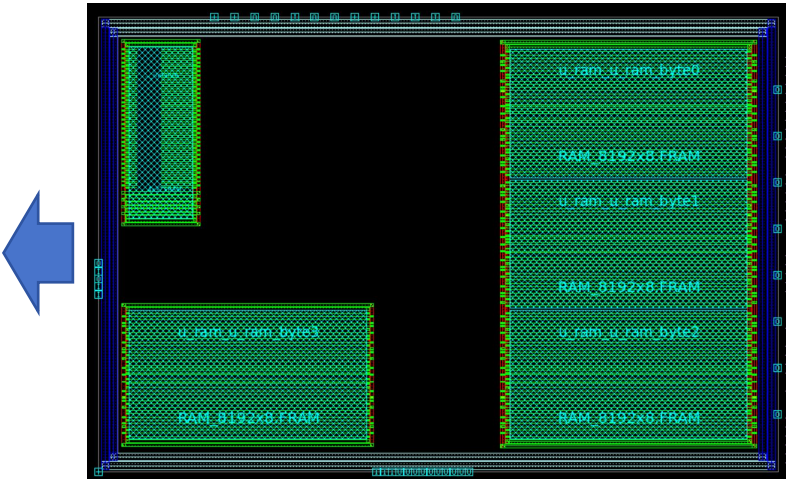
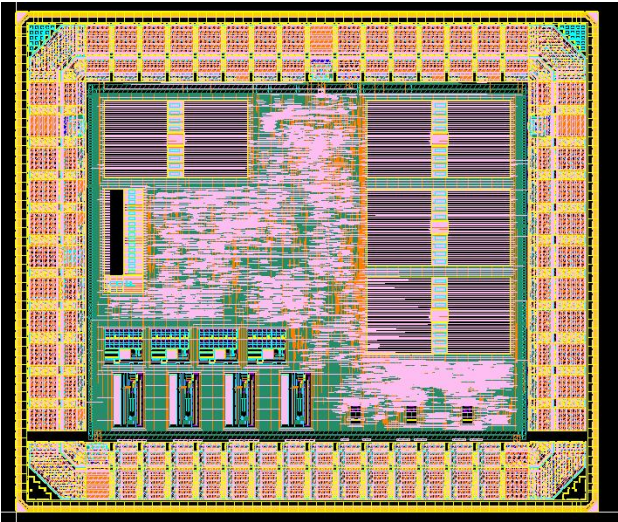
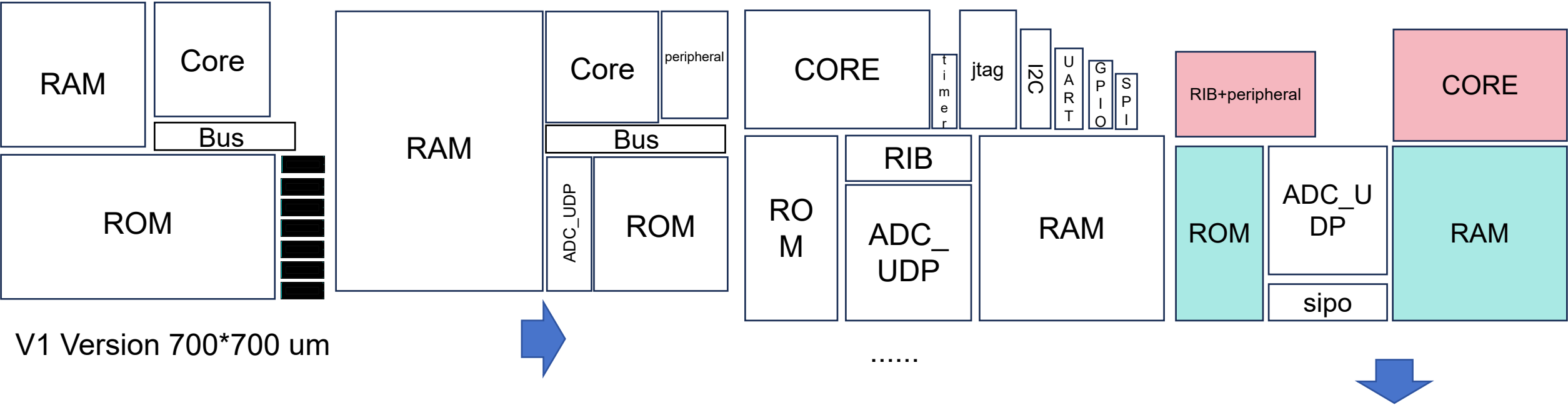
FPGA Validation

```
--- RISC-V Bootloader Started ---
Initializing SD card on SPI channel 1...
[ 6083650000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6734110000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6754010000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6775270000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6795170000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6816430000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6836330000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6857590000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6877490000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6898750000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6918650000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6939910000] SD_MODEL: R1 response sent, returning to IDLE.
[ 6959810000] SD_MODEL: R1 response sent, returning to IDLE.
SD Card on SPI CH-1 initialized successfully.
SD Card initialized successfully.
Target RAM size: 30720 bytes. Need to read 60 blocks from SD card.
Starting to copy data from SD:0x0 to RAM:0x10000000...
Data copy complete.
Bootloader finished. Jumping to application at 0x10000000...
hello world
$finish called from file "rtl/testbench/tb_sd.v", line 71.
$finish at simulation time 150000200000
VCS Simulation Report
Time: 150000200000 ps
CPU Time: 68.590 seconds; Data structure size: 0.2Mb
Thu Sep 18 21:32:31 2025
CPU time: 828 seconds to compile + .199 seconds to elab + .185 seconds to link + 68.632 seconds in simulation
Verdi KDB elaboration done and the database successfully generated: 0 error(s), 0 warning(s)
```

Post-Simulation

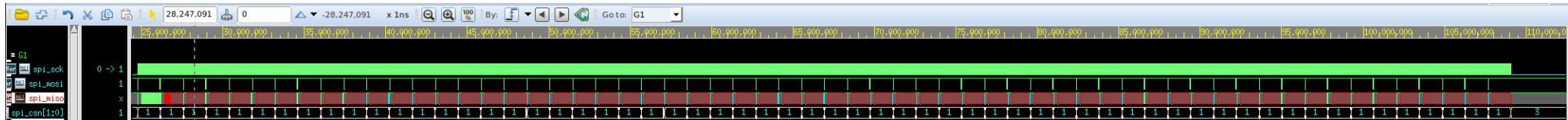


# Digital IC Design Process



# Peripheral Post-Simulation

- SPI Post-simulation is normal (The MISO signal shows 'X' (unknown state) because the SD program only uses a portion of the 32KB space; the remaining unused sections exhibit 'X')



- Added the AT24C32 simulation model. Post-simulation output is normal

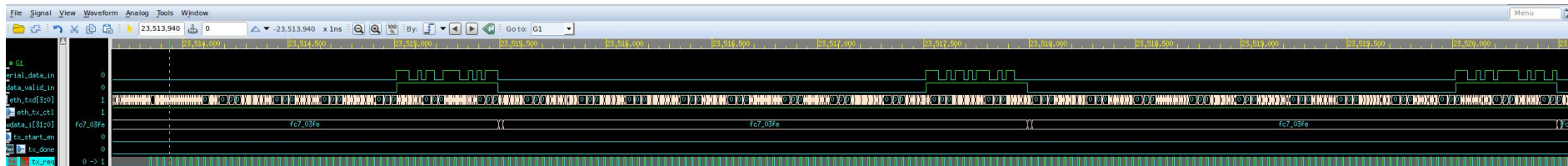


- Timer & ADC-Ethernet: the simulation results are normal

```
==== 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!
```

```
*Verdi* : Create FSDb file 'tiny.fsd'
*Verdi* : Begin traversing the scope (tinyriscv_soc_tb_sd), layer (0).
*Verdi* : End of traversing.
*Verdi* : fsdbDumpen - 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_pln.
[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
[35000200] Simulation finished after sending some packets.
$finish called from file "source/testbench/Auto_test.v", line 207.
$finish at simulation time 35000200
VCS Simulation Report
Time: 35000200 ns
CPU Time: 1314.340 seconds; Data structure size: 25.5Mb
Sat Oct 11 18:52:39 2025
CPU times: 4.425 seconds to compile + .464 seconds to elab + .480 seconds to link + 1314.384 seconds in simulation
Verdi KDB elaboration done and the database successfully generated: 0 error(s), 0 warning(s)
```





# Core Post-Simulation

- Used official RISC-V test suite, All instructions passed the tests, with some results shown below.

```
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
```

## Batch Testing Process:

- 1、 Modified compile/link files to generate .bin files compatible with current SoC.
- 2、 Split .bin files for different instructions into 4 sub-files for the RAMs.
- 3、 Batch loaded and ran programs.
- 4、 Verified data by checking specific memory addresses defined in source code.

Our SOC works as expected

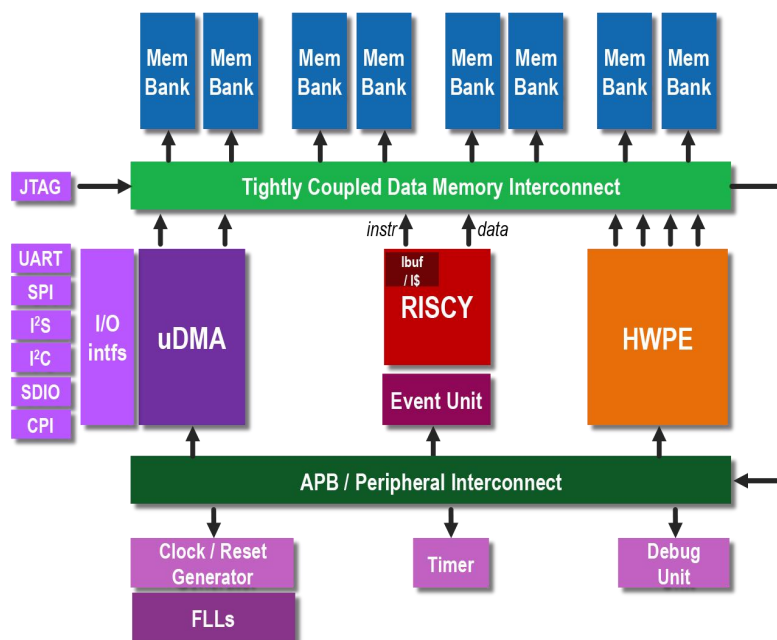
**一、 Background Introduction**

**二、 R&D Process**

**三、 Future Plan**

# Development Plan of HERIS-V2

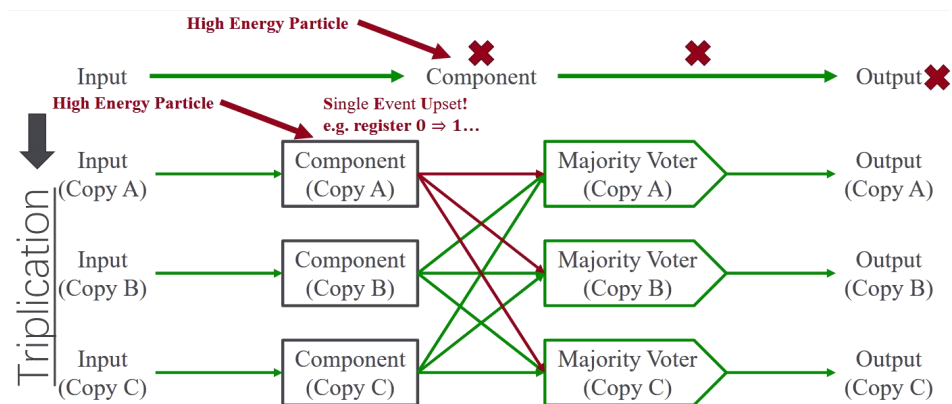
- Migrate to PULP platform's Pulpissimo SoC (Already running on FPGA, passed some program tests)



- Freely switchable cores: RI5CY and IBEX
- Autonomous I/O Subsystem (uDMA)
- New Memory Subsystem
- Support for Hardware Processing Engines
- New simple interrupt controller
- Rich peripherals
- Multi-level internal/external bus configuration

- Radiation Hardening Design

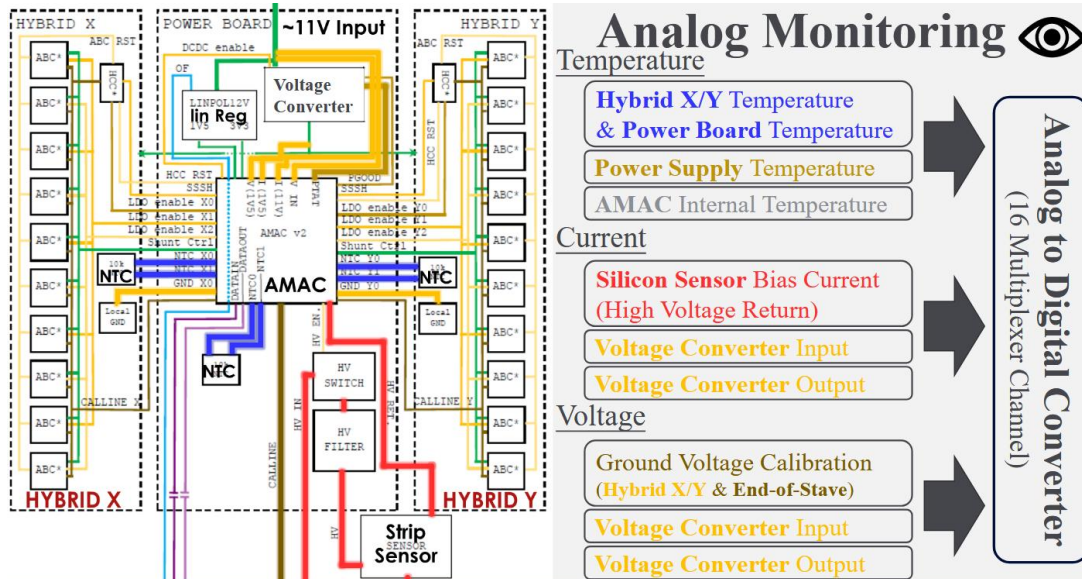
- ECC(Error Correction Code)
- LockStep + ECC
- Full TMR (Triple Modular Redundancy)



Cite by CERN AMAC ASIC

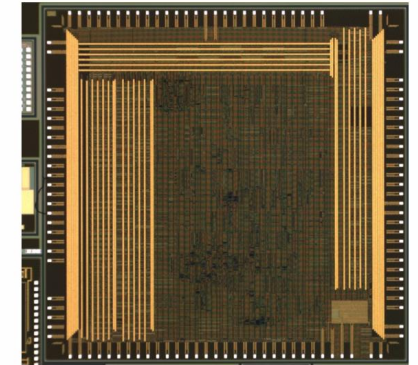
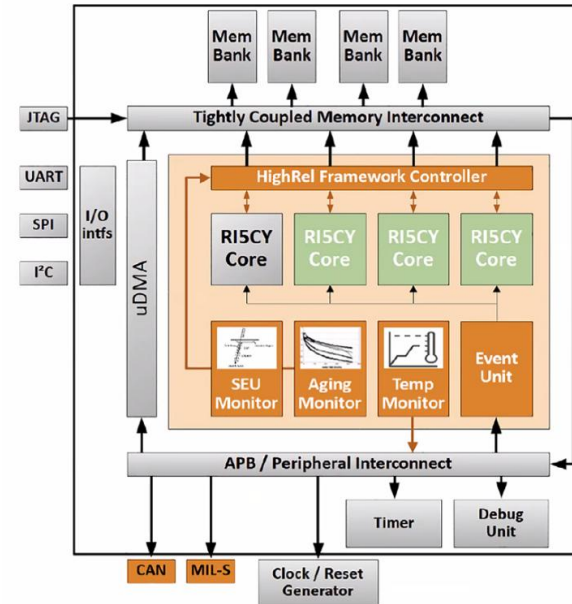
# Futruere Plan

- Near-term Goal: Design test board, await chip return. Implement monitoring functions (e.g., temperature, voltage) similar to AMAC/TETRISC. Integrate with LATRIC for dynamic configuration (e.g., DAC)



## AMAC by ATLAS

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



Chip area	43,56 mm <sup>2</sup> (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>

- Long-term Goal: Leverage RISC-V's openness and customizability to build an intelligent edge data processing platform specifically for High-Energy Physics (HEP) experiments
- Vision: Equip every CEPC ASIC with an "Intelligent Brain"