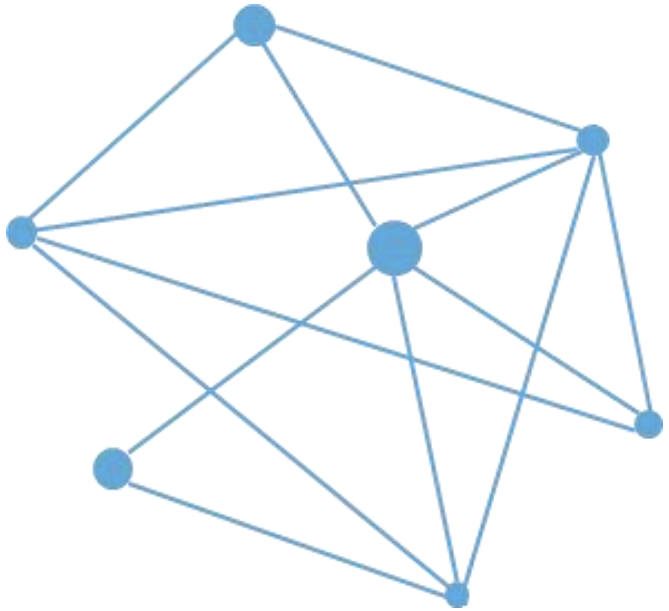


First RISC-V–Based SOC (System-on-Chip) for CEPC Readout ASICs

Reporter: Cui Yuxin

Co-Supervisor: Prof. Yan Qi

R&D Team Members: Cui Yuxin, Chen Jiaolong, Luo Shoudong

Testing Team Members: Wang Hanwen, Zhang Yihan

Date: 2025-10-24

一、Background Introduction

二、R&D Process

三、Future Plan

Reconfigurable Intelligent SoC for High Energy Physics

一、Background & Challenges（Why）

- 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

二、New Design Paradigm (How)

- 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 the chip

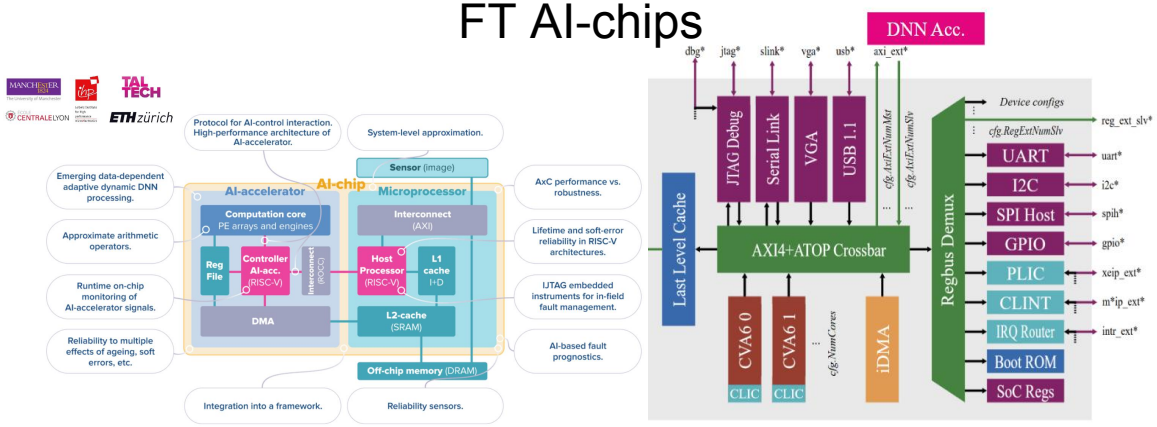
Intelligent SoC is the future trend for ASIC development.

RISC-V in High Energy Physics

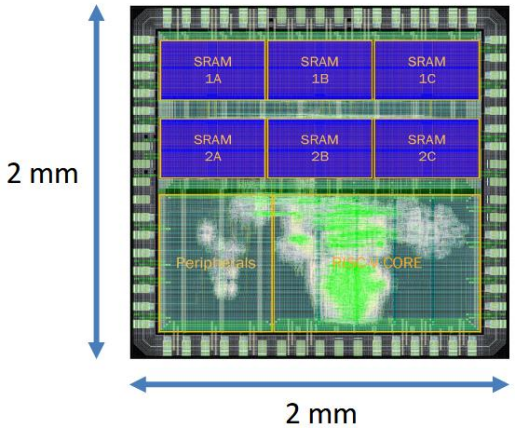
To address future HEP experiment needs, CERN's RISC-V application focuses on two core directions:

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

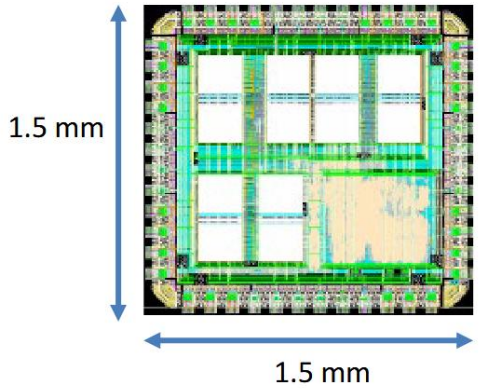
FT AI-chips



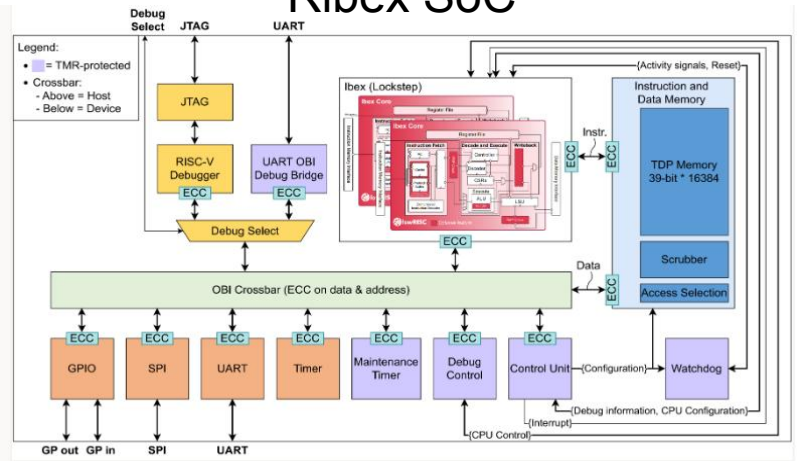
STRV-1 (65nm)



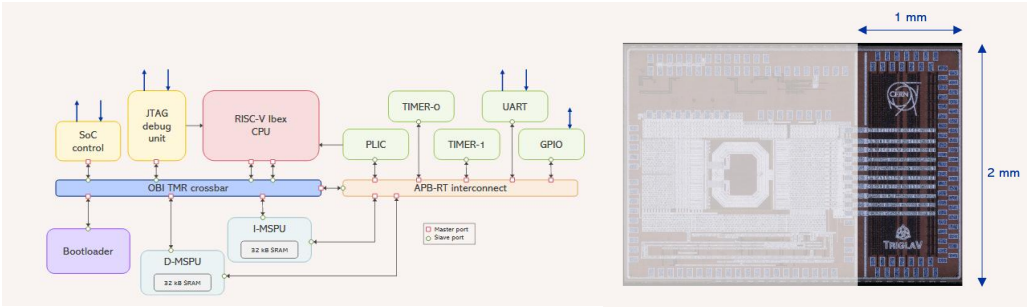
STRV-2 (28nm)



Ribex SoC

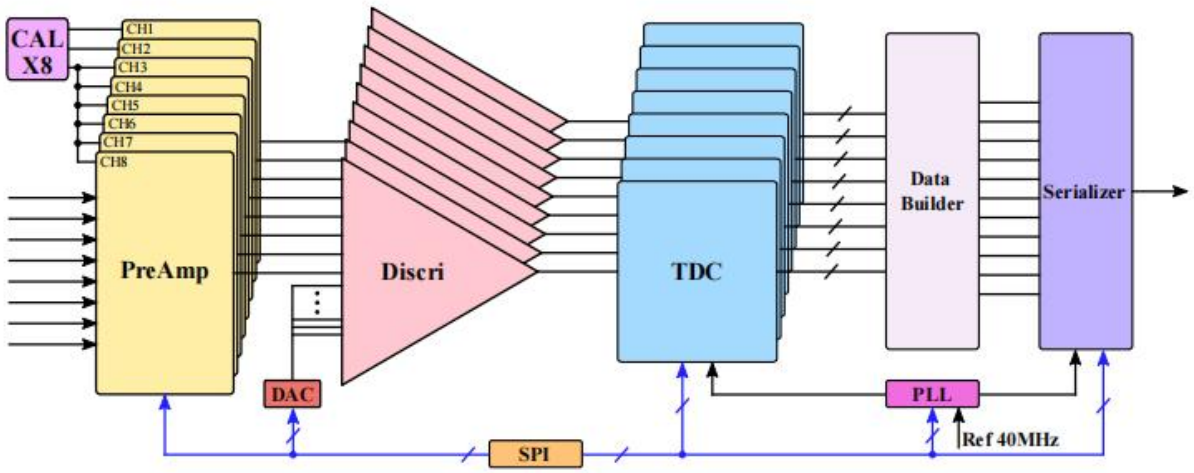


TriglaV

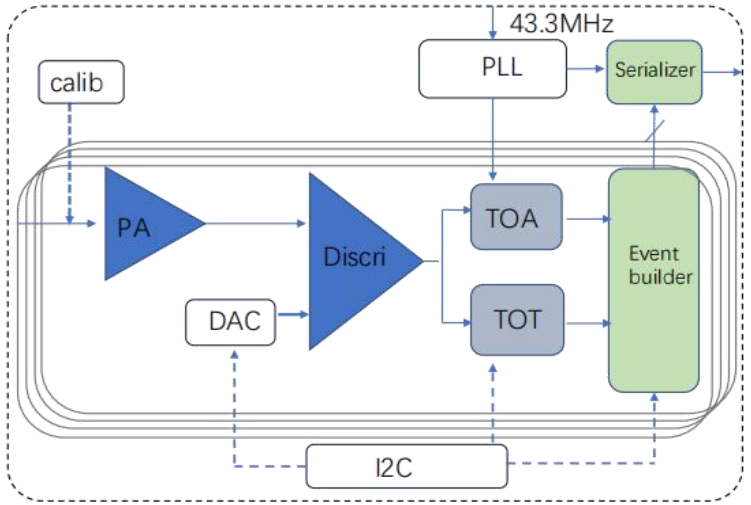


RISC-V Applied to LATRIC

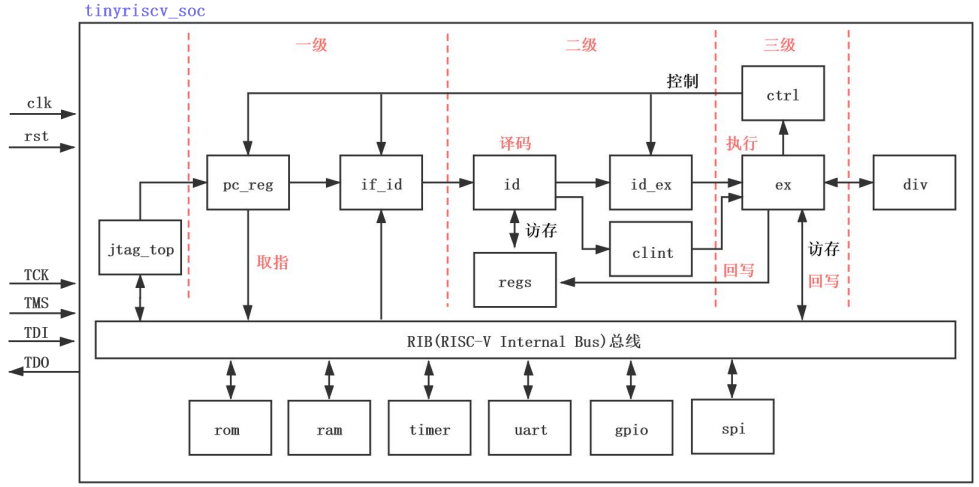
At the initial research stage, we selected the beginner-friendly Tiny-Riscv soft core, implementing a dynamically configurable DAC module and other algorithms on OTK.



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.

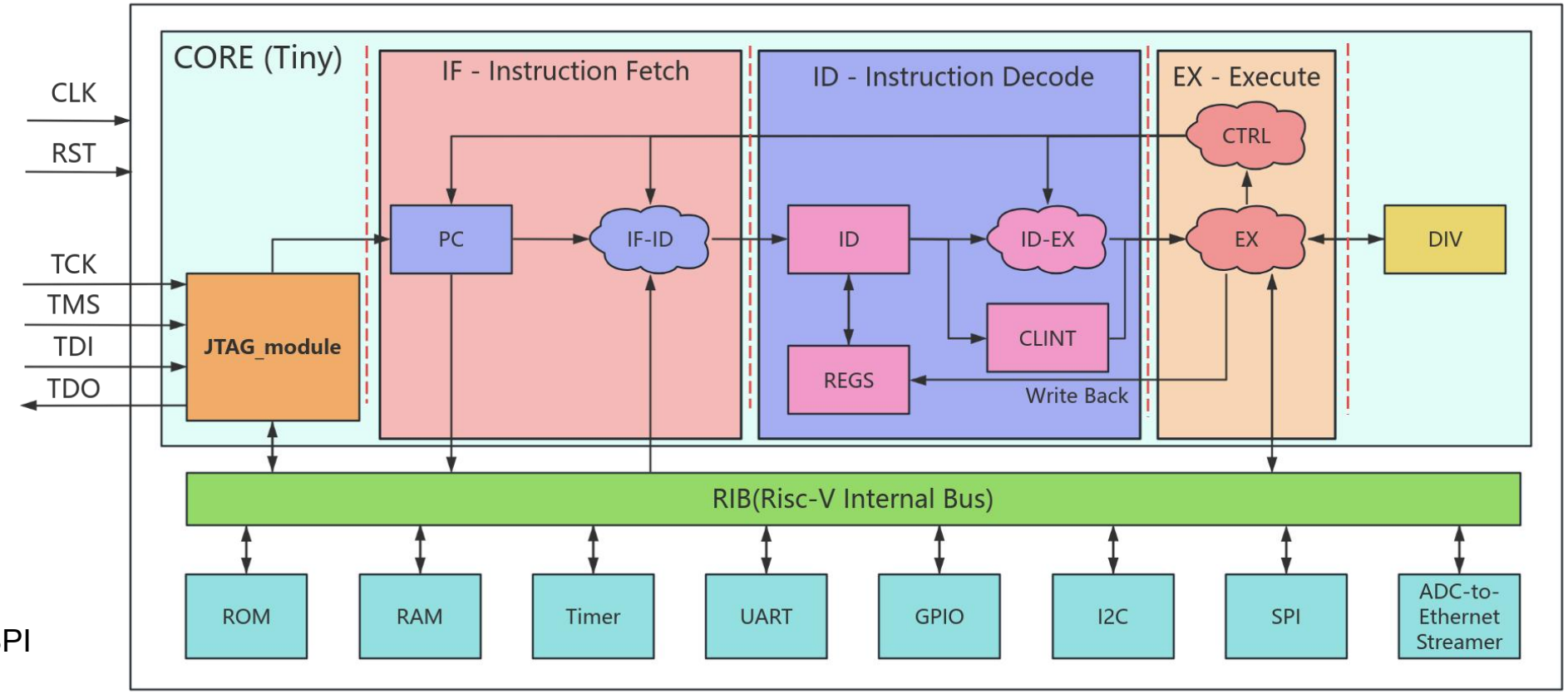
一、Background Introduction

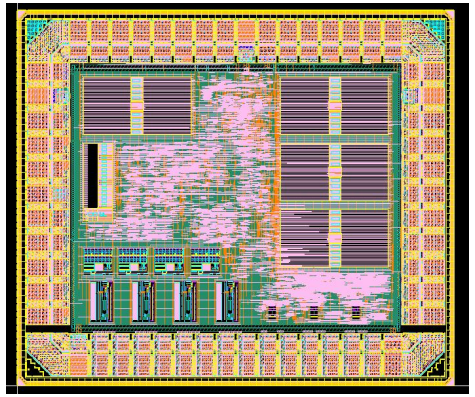
二、R&D Process

三、Future Plan

Tiny-RiscV Architecture Diagram

- 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

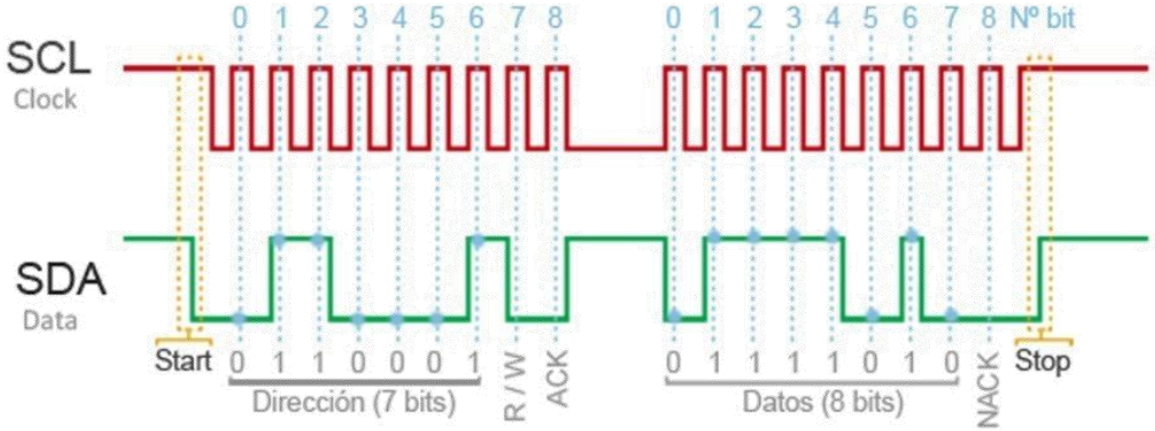




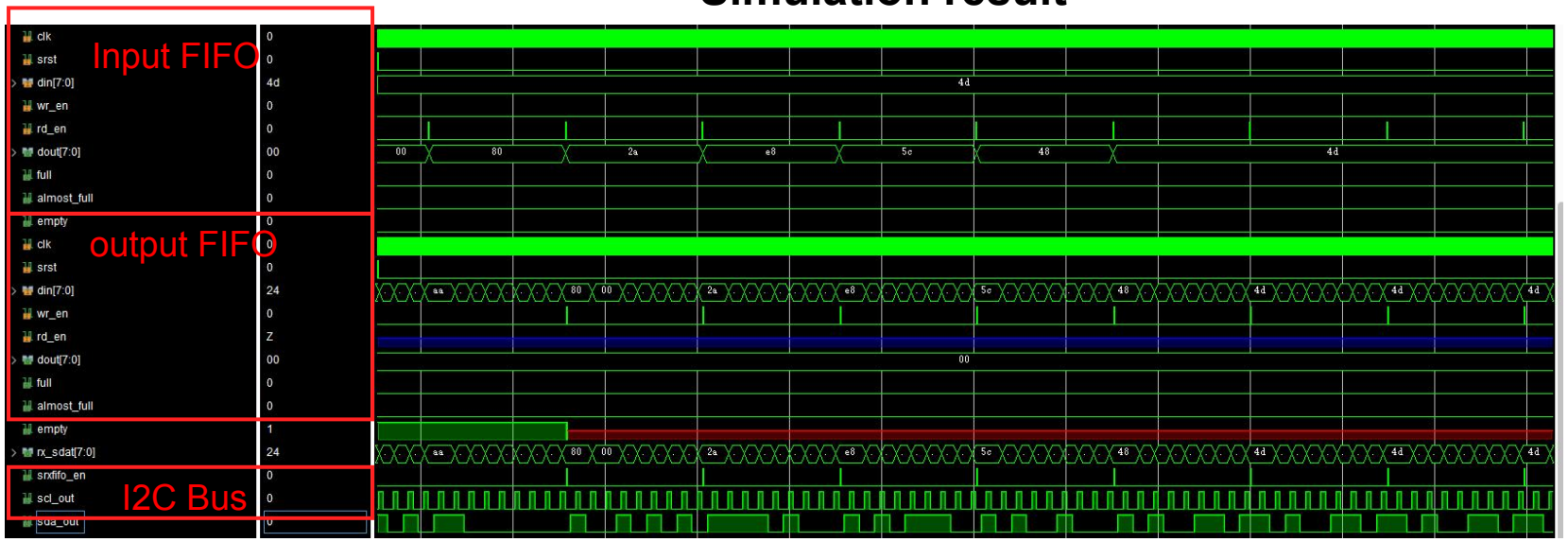
- 55 nm process technology
- Frequency 50 MHz
- Size 1020 x 1196 um
- Design verification completed in October; submitted for tape-out.
- Testing is scheduled to begin after the chips return, expected in January 2026

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



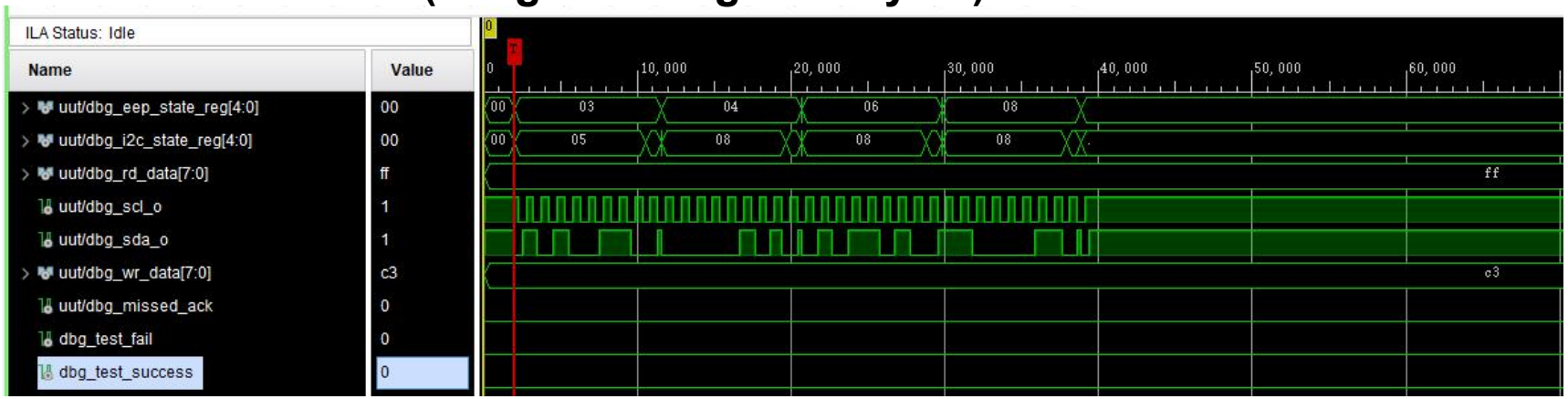
I2C Module FPGA Verification

Used the onboard AT24C64 EEPROM memory chip for functional validation.

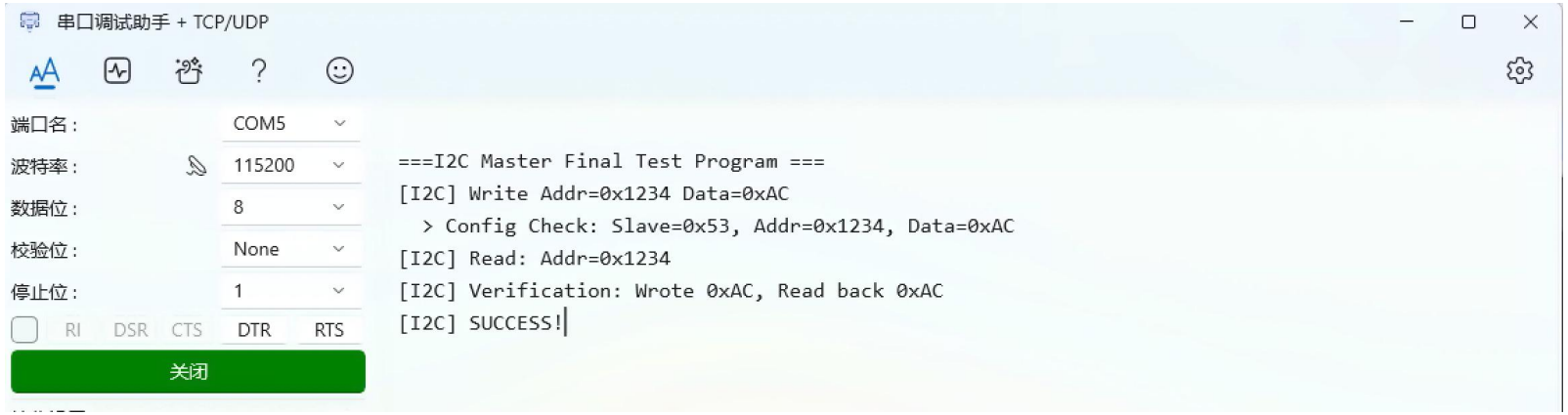
C Program

```c
#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(Integrated Logic Analyzer) results



UART output results

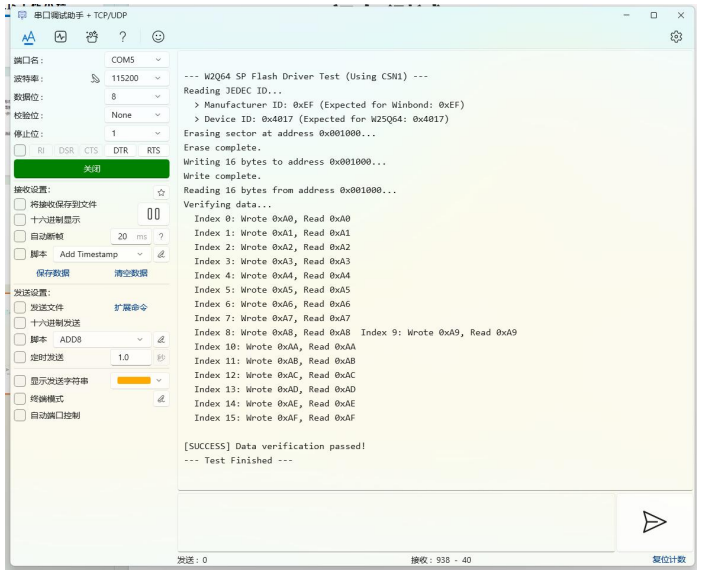


Optimizing & Testing SPI Master Module

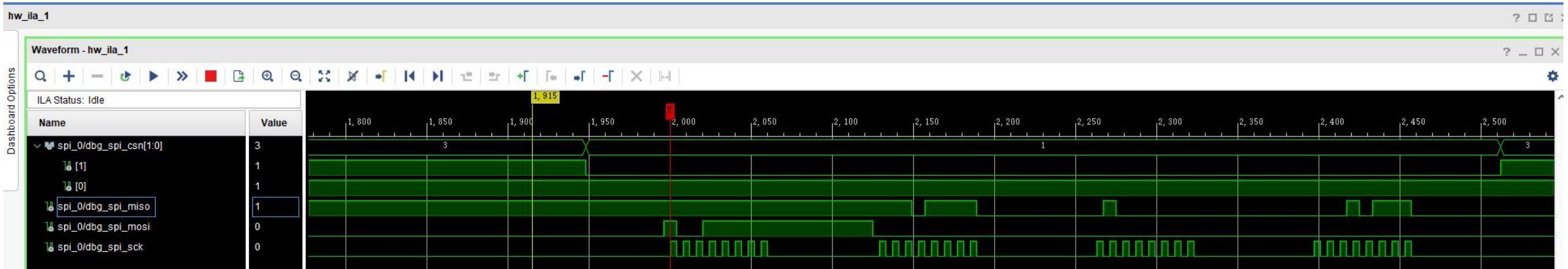
- Replaced the original SPI module with a more mature SPI Master, and added an interface wrapper
- Successfully verified module functionality on FPGA using the W25Q64 SPI Flash memory chip

W25Q64

UART output results

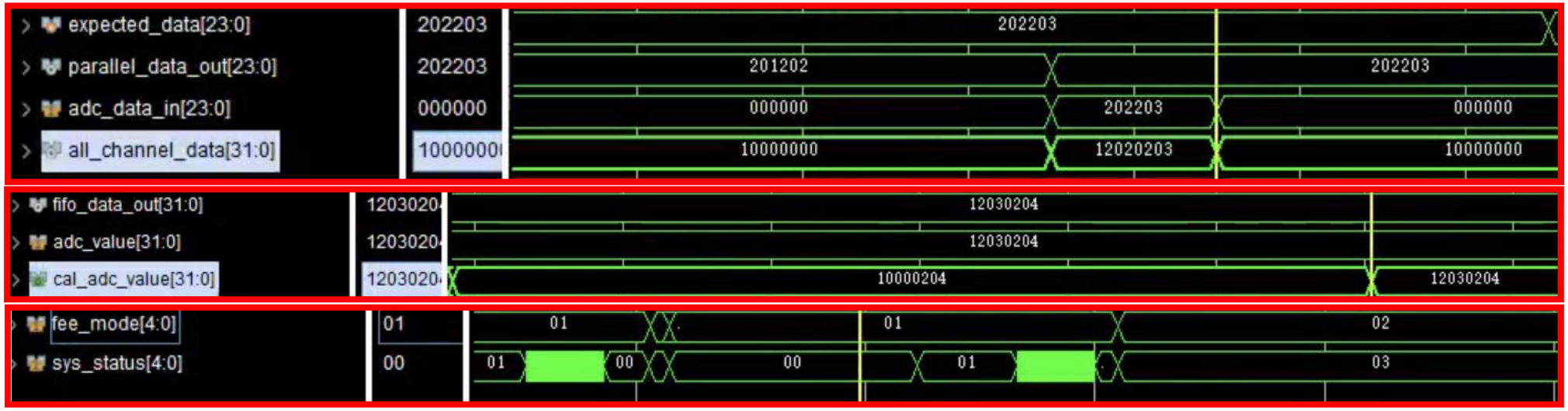


ILA(Integrated Logic Analyzer) results



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 (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
```

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
```

C program feedback data

C program input data

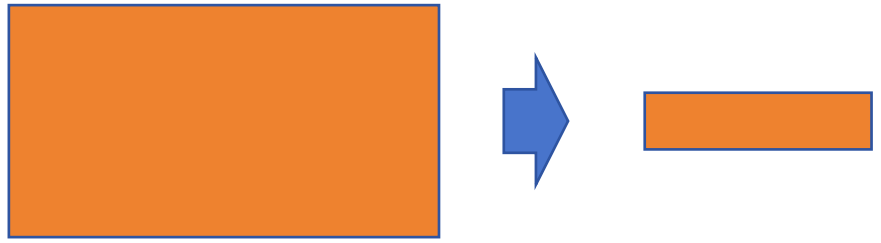
Optimizing Boot Method

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

RAM 8192x32
569.26*307.045 um

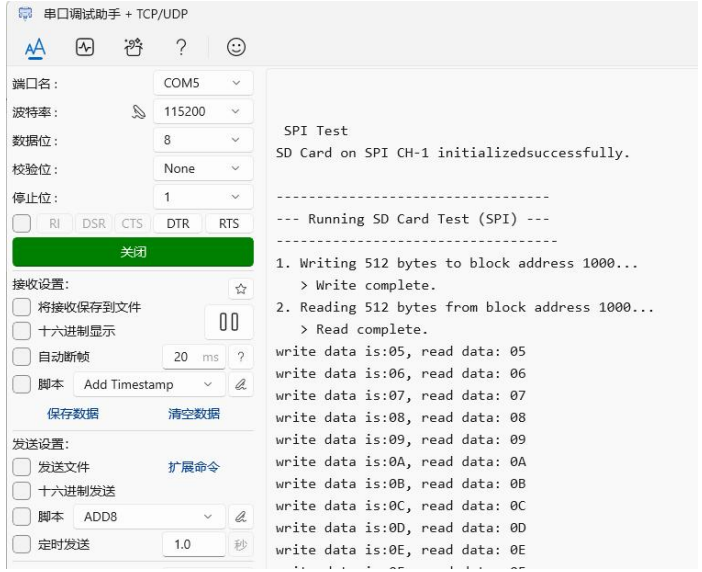
ROM 512x32
75.845*300.46



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

Output from SD card test



BOOT Loader Design

- Current RAM size: 16KB (14KB 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 (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);
}
```

Startup Assembly File

```
// 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 program

```

// ... (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);
}
```

Hardware Code Modifications for Bootloader

1、Fetch Module: Controlled by internal signals rst_n, flush_i, stall. No direct inputs from external debug 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.

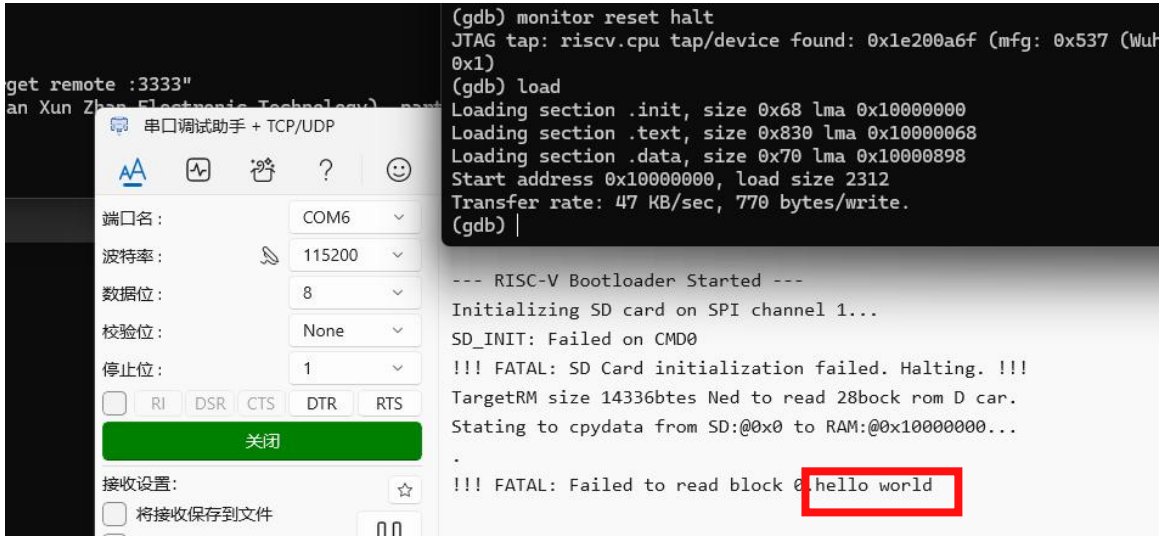
```
end else begin
    // when write dpc, we reset cpu here
    if (data[15:0] == DPC) begin
        //dm_reset_req <= 1'b1;
        jtag_pc_we <= 1'b1;
        jtag_pc_wdata <= data0;
        dm_halt_req <= 1'b0;
        dmstatus <= {dmstatus[31:12], 4'hc, dmstatus[7:0]};
    end else if (data[15:0] < 16'h1020) begin
        dm_reg_we <= 1'b1;
        dm_reg_wdata <= data0;
    end
```

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;
```

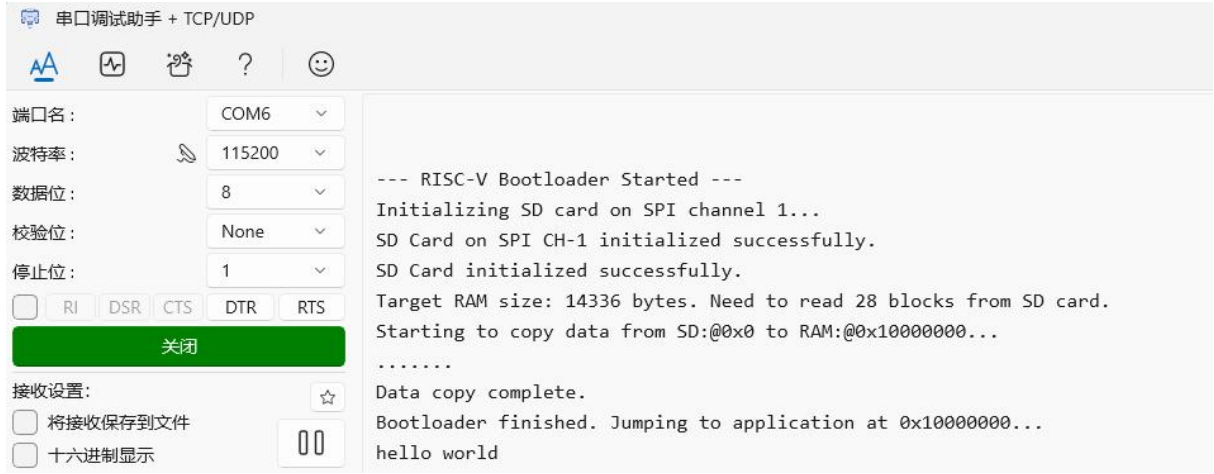
Post-modification: Normal startup via GDB is successful



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

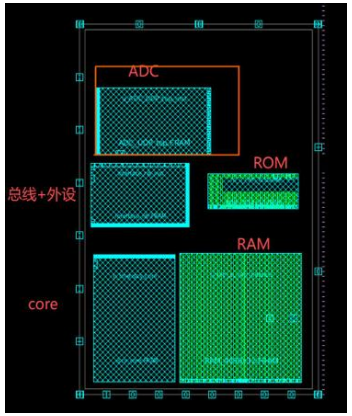
```
# file: debug.gdb
target remote localhost:3333
monitor reset halt
load
set $pc = 0x10000000
b main
c
```

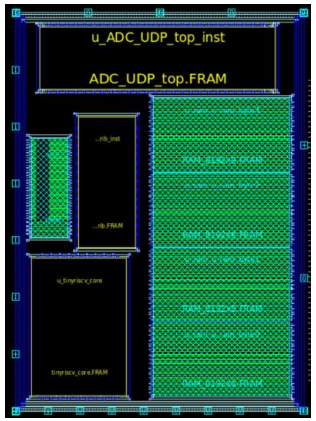
```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from .\gpio...
0x00000000 in ?? ()
JTAG tap: riscv.cpu tap/device found: 0x1e200a6f (mfg: 0x537 (Wuhan Xun Zhan Electronic Technology), part: 0xe200, ver:
0x1)
Loading section .init, size 0x6c lma 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.
```

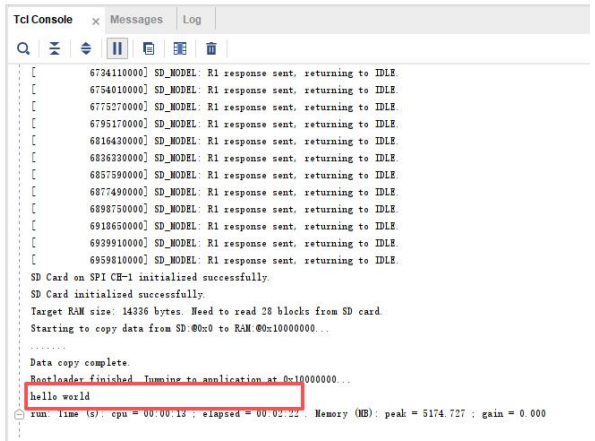
RAM Modification

Modified RAM architecture to implement byte-select functionality

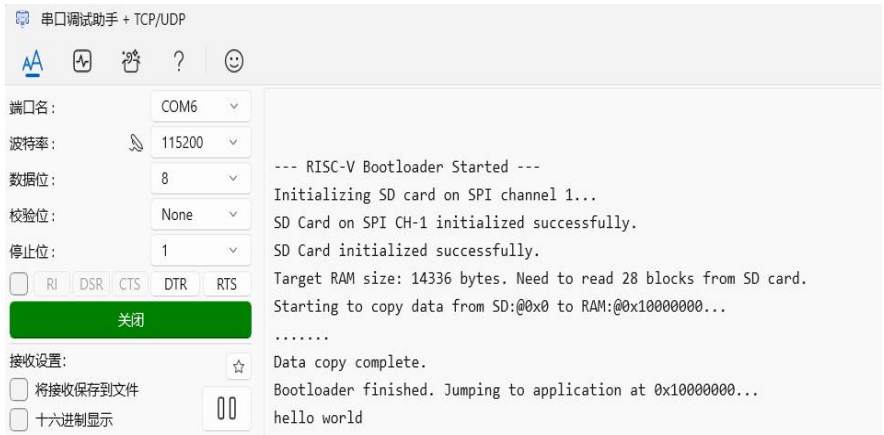
- Changed 32-bit RAM to 4 combined 8-bit RAMs.
- Added a Verilog model for SD card simulation for pre- and post-layout verification.



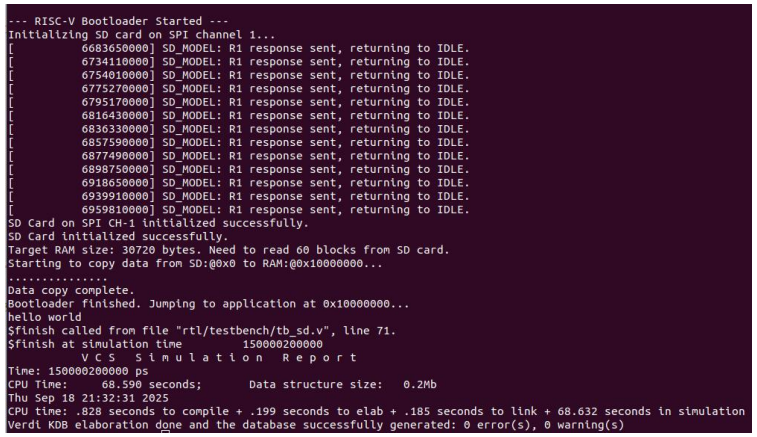




Pre-Simulation

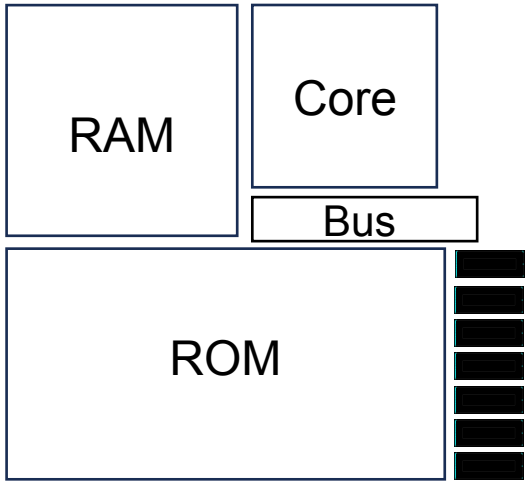


FPGA Validation

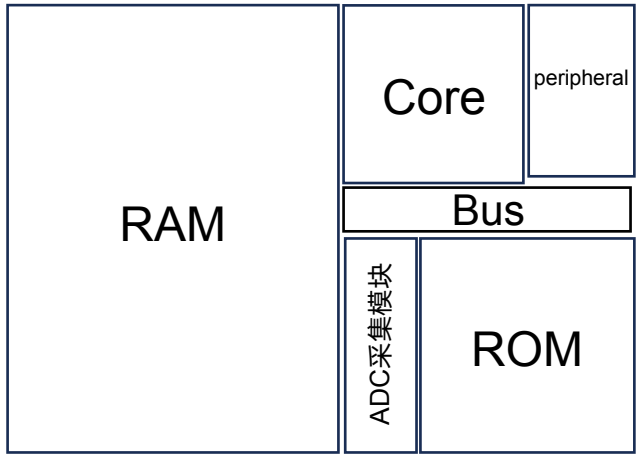


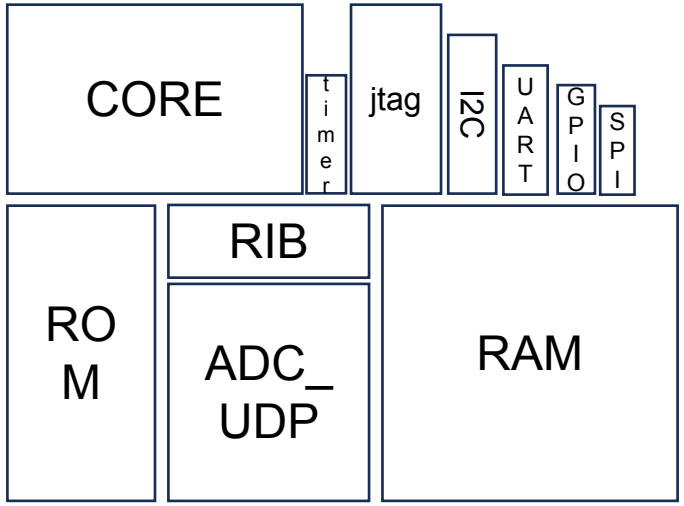
Post-Simulation

Digital IC Design Process

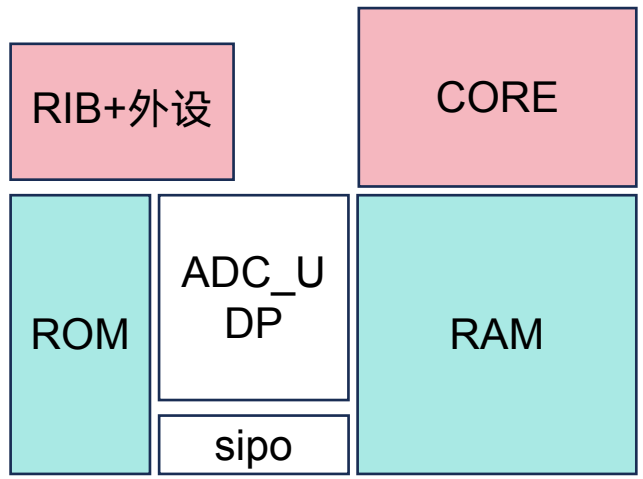


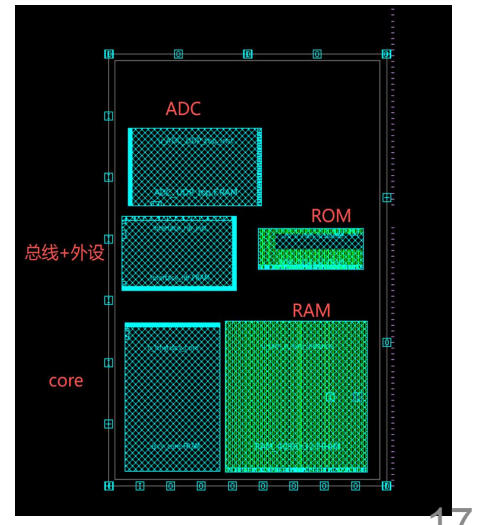
V1 Version 700*700 um

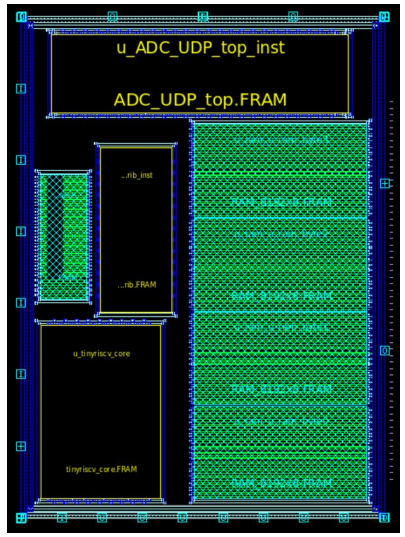


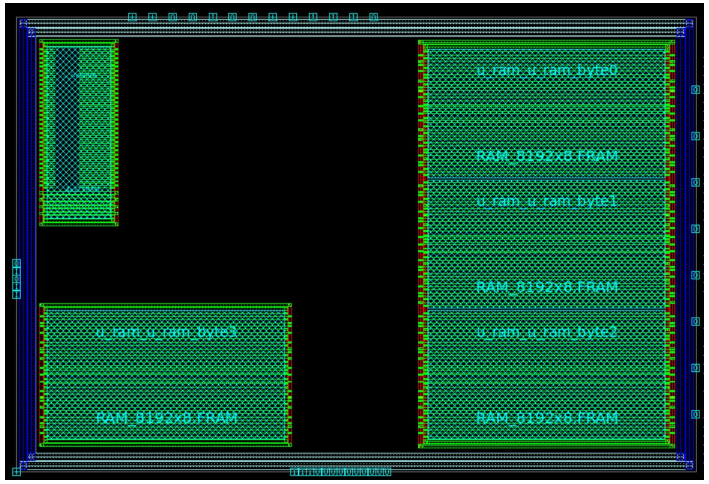


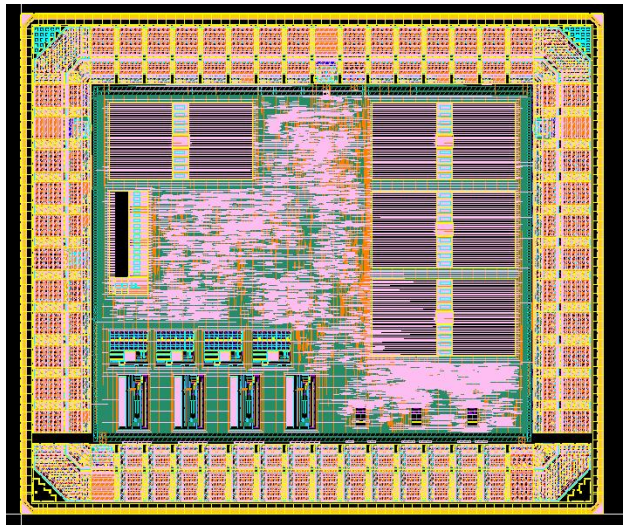
.....





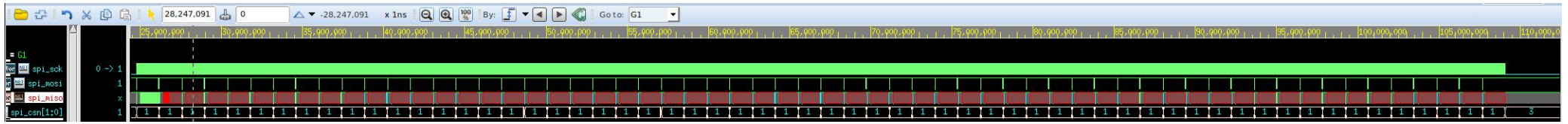




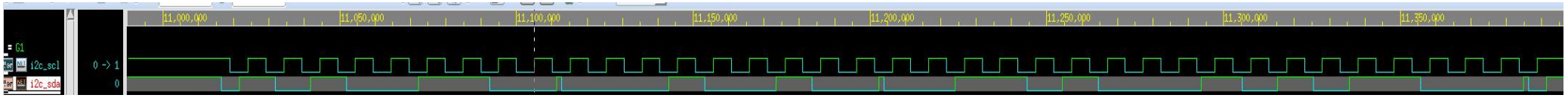


Peripheral Post-Simulation

- 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')



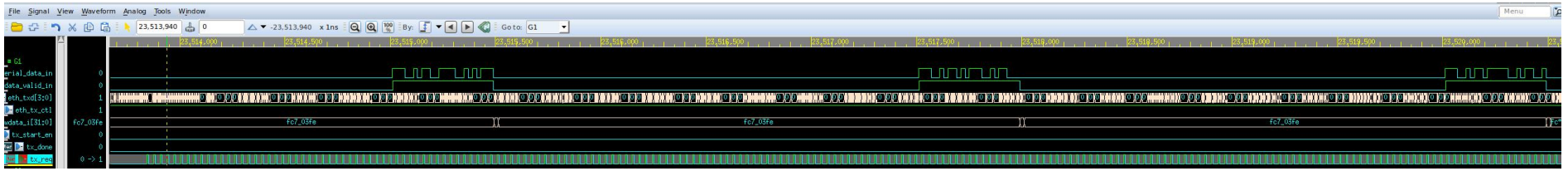
- Added the AT24C32 simulation model. Post-simulation output is normal



- Timer & ADC-Eithernet: 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.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)
```



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.

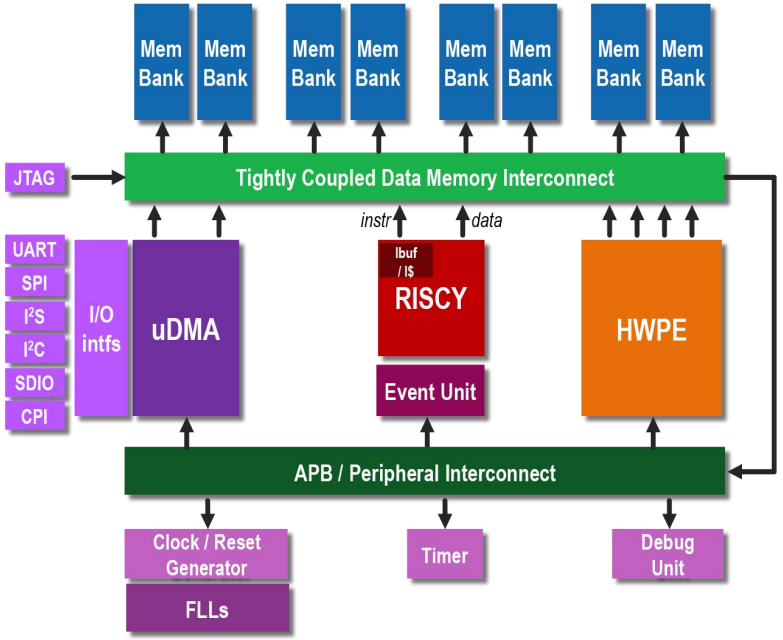
一、Background Introduction

二、R&D Process

三、Future Plan

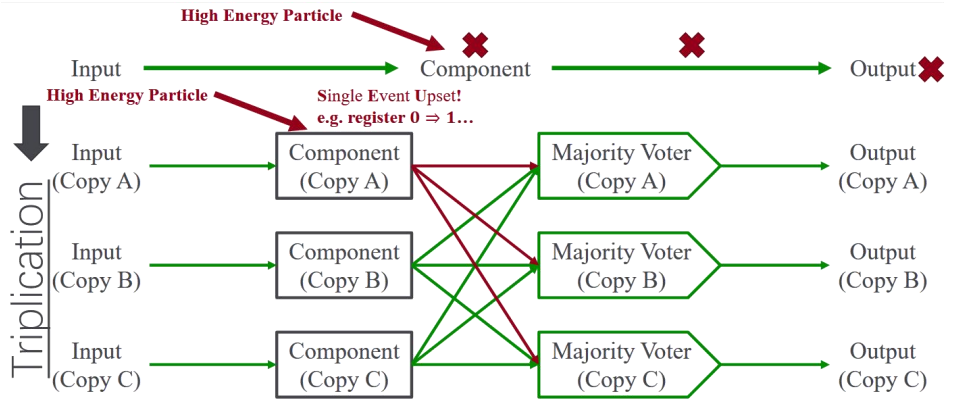
Agenda

- Migrate to PULP platform's Pulpissimo SoC (Already running on FPGA, passed partial 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

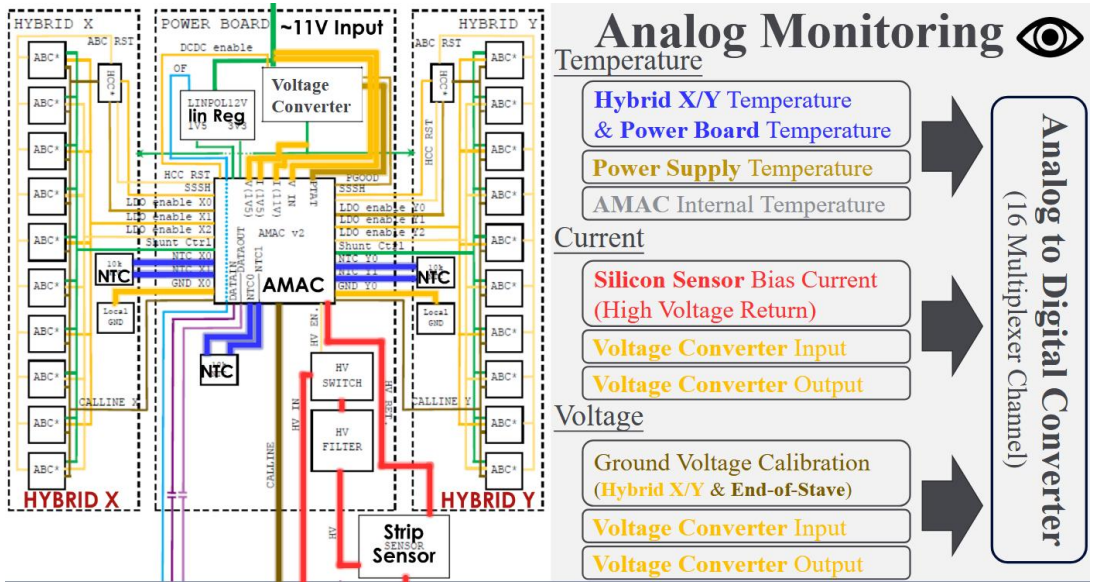
- Radiation Hardening Design
  - ECC(Error Correction Code)
  - LockStep + ECC
  - Full TMR (Triple Modular Redundancy)



Cite by CERN AMAC ASIC

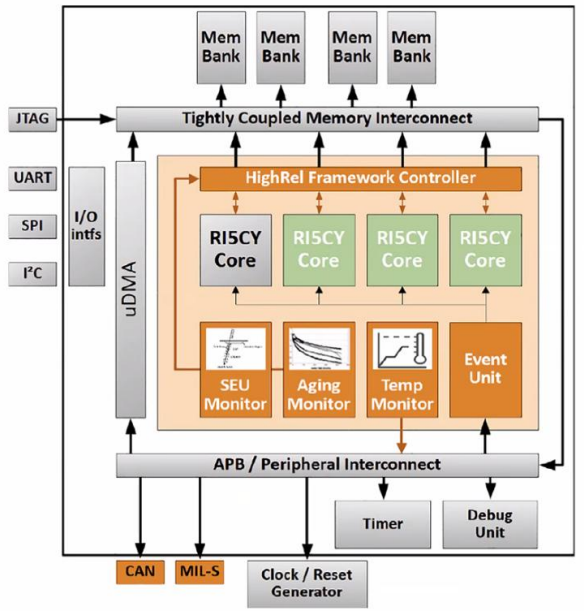
后续规划

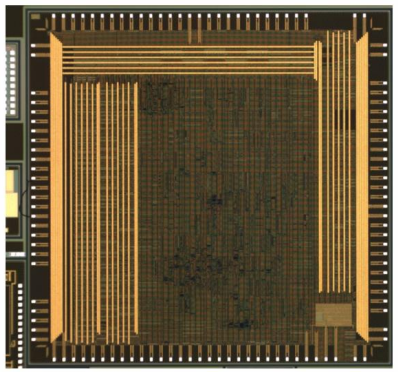
- Near-term Goal：Design test board, await chip return. Implement monitoring functions (e.g., temperature, voltage) similar to AMAC/TETRISC. Integrate with LATRIC for dynamic configuration (e.g., DAC)



AMAC by ATLAS

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





Chip area	43,56 mm^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

- 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 ASIC designed for HEP with an "Intelligent Brain"