Open Source VSRV1 and VSRV2 RISC-V Cores

Document History

Revision	Date	Author/Org.	Description
V1.0	2026-03-30	TS+TK	Original release

TRISTAN has received funding from the Key Digital Technologies Joint Undertaking (KDT JU) under grant agreement nr. 101095947. The KDT JU receives support from the European Union's Horizon Europe's research and innovation programme and Austria, Belgium, Bulgaria, Croatia, Cyprus, Czechia, Germany, Denmark, Estonia, Greece, Spain, Finland, France, Hungary, Ireland, Israel, Iceland, Italy, Lithuania, Luxembourg, Latvia, Malta, Netherlands, Norway, Poland, Portugal, Romania, Sweden, Slovenia, Slovakia, Turkey

Table of Contents

Figures

Tables

1 Introduction

1.1 Background

VLSI Solution is a company with more than 30 years of history, primarily focused on the development and sale of audio integrated circuits. The company's products extensively utilize an in-house DSP processor architecture known as VSDSP.

The original motivation for this project was to integrate Ethernet connectivity into an audio chip product family. During the initial feasibility analysis, it became clear that networking functionality is predominantly software-driven. Supporting protocols such as:

- TELNET
- HTTP
- DHCP
- Linux networking stack
- Ethernet drivers

requires a substantial and maintainable software ecosystem. However, these features do not demand high-performance application-class hardware.

It was therefore concluded that the existing proprietary DSP architecture was not suitable for running a modern networking software stack. This led to an investigation of alternative processor architectures, with particular interest in RISC-V and Linux due to their openness, ecosystem maturity, security features, and long-term sustainability.

1.2 General Information

The objective of the VSRV project was to design and implement two generations of 32-bit RISC-V CPU cores: **VSRV1** and **VSRV2**.

The primary deliverables of the project are the CPU core designs and their associated hardware descriptions. Integrated circuit (IC) implementations were developed as demonstrator chips to validate functionality, performance, and system integration. These chips serve as proof-of-concept platforms for the cores.

The VSRV1 and VSRV2 cores are based on the 32-bit RISC-V instruction set architecture (ISA) and are capable of running a standard Linux operating system kernel when integrated into a complete system that includes external memory and the required peripherals.

VSRV1 establishes the baseline RV32IM core capable of running a full Linux operating system at low clock frequencies. VSRV2 builds upon this foundation by introducing increased memory bandwidth, DMA support, and multiple performance-enhancing architectural extensions.

The project originated from an evaluation of the scalar version of the UltraEmbedded RISC-V core [1] (https://github.com/ultraembedded/). At the time of project initiation, this implementation was among the simplest 32-bit RISC-V cores capable of booting Linux without major architectural modifications. During the project, architectural and structural limitations were identified and addressed, resulting in the enhanced VSRV1 and VSRV2 cores.

For validation and integration purposes, the cores were assembled into complete processor systems including:

- RISC-V CPU core
- Memory Management Unit (MMU)
- AXI interface to an external LPDDR memory controller
- VSBUS peripheral interconnect
- Essential peripherals (UART, GPIO, SPI, timer, interrupt controller)
- Memories

These processor systems can be used directly in FPGA-based implementations, as most FPGA platforms provide the required macro blocks, such as LPDDR interfaces and on-chip SRAM memories.

For IC (ASIC) implementations, technology-specific components must be provided by the integrator or licensed from VLSI Solution, including:

- LPDDR controller interface
- SRAM memory macros
- Standard cell libraries
- Peripherals in addition to essentials which are included

All hardware descriptions of VSRV1 and VSRV2 are written exclusively in VHDL. This ensures language uniformity and avoids the additional licensing overhead associated with mixed-language (VHDL/Verilog) simulation environments.

VSRV1 and VSRV2 processor systems are available as **Open Source under permissive Solderpad license**.

1.3 Purpose and Scope

This document describes the architecture, design, and implementation of the VSRV1 and VSRV2 RISC-V CPU cores. It also provides a high-level overview of the demonstrator processor systems used for validation and integration testing.

All information in this document is public and may be freely distributed. **It is suggested that any copying, reproduction, or use of the material include a reference to this document**.

1.4 Acronyms and Definitions

ACRONYM	DESCRIPTION
ASIC	Application Specific Integrated Circuit
ATE	Automated Test Equipment
AXI	Advanced eXtensible Interface
CSR	Control and Status Registers
DDR	Double data rate high-speed dynamic random access memory
DMA	Direct Memory Access
DSP	Digital Signal Processing
DRAM	Dynamic Random Access Memory
DUT	Device Under Test
FPGA	Field Programmable Gate Array
IoT	Internet of Things
IP	Intellectual Property
ISA	Instruction Set Architecture
LEC	Conformal Logic Equivalence Checker
LPDDR	Low-Power Double Data Rate memory

ACRONYM	DESCRIPTION
LSB	Least Significant Bit
MMU	Memory Management Unit
MSB	Most Significant Bit
OS	Operating System
PC	Program Counter
PPA	Power, Perfirmance, and Area
RAM	Random Access Memory
SoC	System on a Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TBD	To be done
TLB	Translation Lookaside Buffer
UART	Universal Asynchronous Receiver Transmitter
VSBUS	Custom peripheral bus of VLSI Solution
VSRV	RISC-V core(s) of VLSI Solution
VSRV1/2	VSRV1 or VSRV2

2 Description of the Architecture

2.1 System Architecture

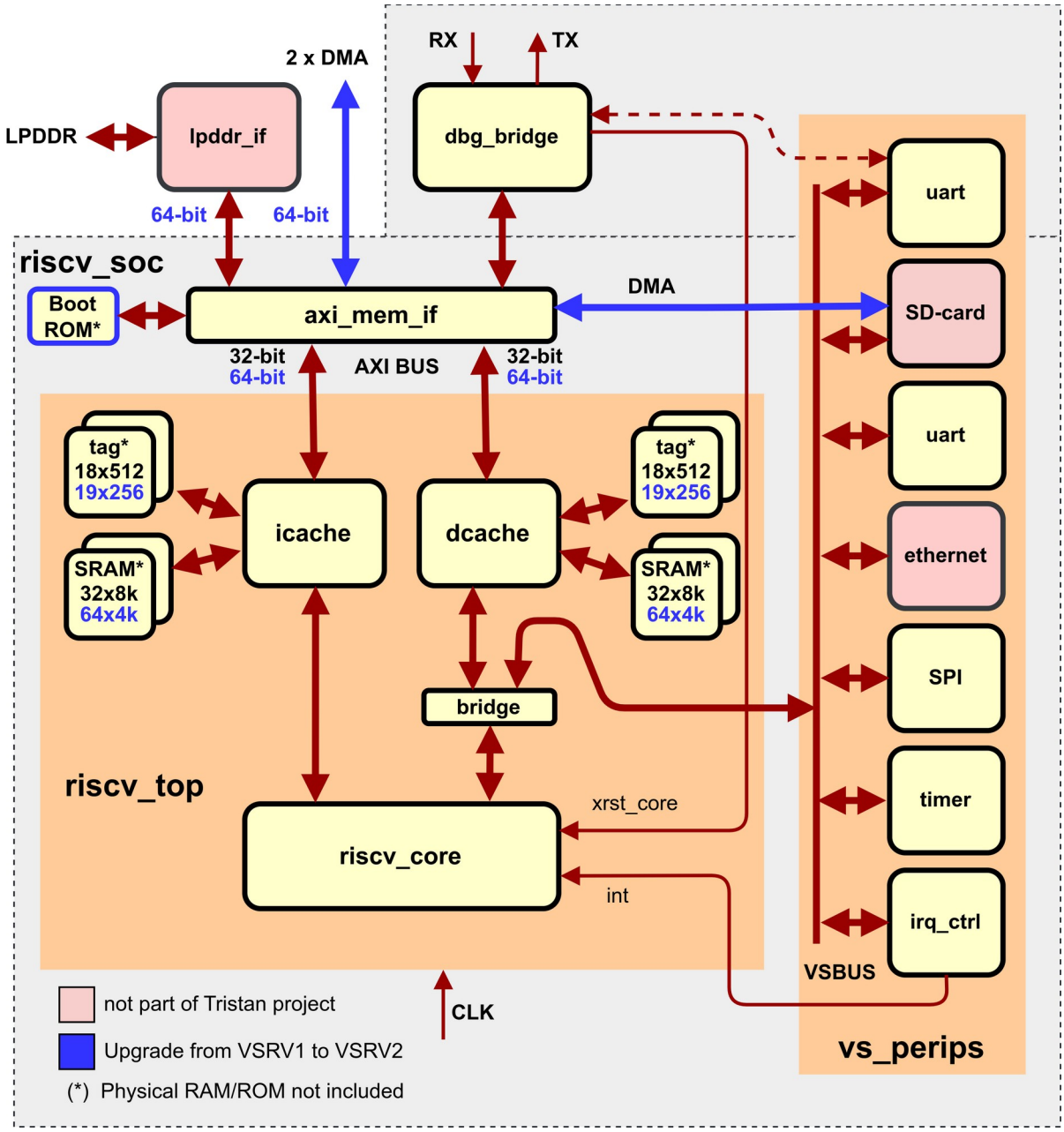

Figure 1: Architecture of VSRV1 and VSRV2 processors

The starting point of the development was the UltraEmbedded RISC-V core [1]. This implementation was capable of booting Linux 4.20 but was unable to reliably support newer Linux kernel versions. Furthermore, due to the absence of a data cache, the system boot time was significantly long (see Table 14).

2.1.1 VSRV1 Architecture

The VSRV1 processor was designed to establish a modernized and simplified 32-bit RISC-V core capable of running recent Linux kernels.

The deliverable includes a complete processor subsystem consisting of:

- RISC-V CPU core (VSRV1 core)
- Memory Management Unit (MMU)
- AXI interface to an external LPDDR2 memory controller
- VSBUS peripheral bus with three most important peripherals

The original Verilog implementation was completely rewritten and cleaned into a uniform VHDL environment. This eliminated the need for mixed-language simulation, thereby avoiding the additional licensing cost associated with maintaining both Verilog and VHDL simulation tools.

Several architectural improvements were introduced (see red lines of Figure 1):

New peripheral interconnect:

- The peripheral interface was moved away from the AXI bus to reduce latency and simplify peripheral access logic.
- A new peripheral interconnect, **VSBUS**, was adopted. This is the same bus architecture used in the VSDSP processor, enabling reuse and easy adaptation of existing peripherals.
- Four essential I/O peripherals were validated:
  - UART
  - Timer
  - Interrupt controller
  - SPI

AXI bus stripping:

- The AXI bus implementation was simplified (stripped) to reduce latency. This was possible because the AXI master interface connects exclusively to the LPDDR memory controller, eliminating the need for full multi-slave arbitration and complex interconnect features.

MMU parametrization:

- The MMU was extended to support larger memory configurations.

Cache modifications:

- The instruction cache was redesigned to use conventional single-port SRAM instead of dual-port SRAM to reduce silicon area.
- The cache architecture was made configurable to facilitate future architectural optimizations.
- A data cache was added, including support for a dirty flag to enable proper write-back operation.

RISC-V extensions:

- Zifencei was added to support correct cache flushing for SD-card operations. This ensures proper ordering and visibility of memory operations when interacting with external storage.

These changes established a clean, configurable, and Linux-capable baseline architecture.

2.1.2 VSRV2 Architectural Enhancements

The VSRV2 processor architecture includes four major enhancements compared to the VSRV1 architecture (see blue lines of Figure 1).

Wider AXI Bus:

- The AXI data bus width was increased from 32 bits to 64 bits.
- Corresponding changes were made in the instruction and data caches and related control logic.
- This modification increases memory bandwidth and improves overall system performance.
- Change of the bus width required update of the LPDDR2 controller

DMA Support:

Three identical Direct Memory Access (DMA) channels were added to the AXI bus to support:

- SD-card operations
- Data transfers to/from the DSP processor
- Memory-to-memory copying

The DMA channels reduce processor overhead and improve data movement efficiency.

Boot ROM:

In VSRV1, the boot image is loaded into LPDDR memory either via:

- Debug bridge UART interface, or
- A companion DSP processor

VSRV2 introduces an integrated boot ROM, providing the first step toward independent, fast, and potentially secure system boot.

RISC-V ISA Extensions:

VSRV2 adds multiple extensions to improve functionality and performance:

- Atomic operations (A)
- Bit manipulation and address generation (B / Zba, Zbb, Zs)
- Bit manipulation and cryptography (Zbkb)
- Conditional arithmetic operations (Zicond)

These extensions provide enhanced support for peripheral control, cryptography, and complex arithmetic, enabling faster execution of software routines compared to VSRV1 as well as significantly faster boot.

2.2 RISC-V Core Architecture

The **VSRV1** and **VSRV2** cores are designed to be simple, compact, and efficient, targeting low-power and low-latency applications such as Internet of Things (IoT) devices. Both cores implement a **single-issue, in-order 5-stage pipeline** supporting the **RV32IM RISC-V ISA**, along with the SU, Zicsr, and Zifencei extensions required for Linux.

Key features of the cores include:

- **32-bit RISC-V architecture** with 5-stage in-order pipeline
- **Linear pipeline** and in-order with controlled handshaking between stages to manage stalls due to memory, hazards, and exceptions
- **Flexible memory integration:** cores can operate with separate instruction and data caches or a single tightly-coupled dual-port RAM
- **Optimized instruction timing:** most instructions complete in a single clock cycle; multi-cycle instructions such as multiplication (2 cycles) and division (17 cycles) are supported
- **Result forwarding:** available to minimize pipeline stalls caused by data hazards

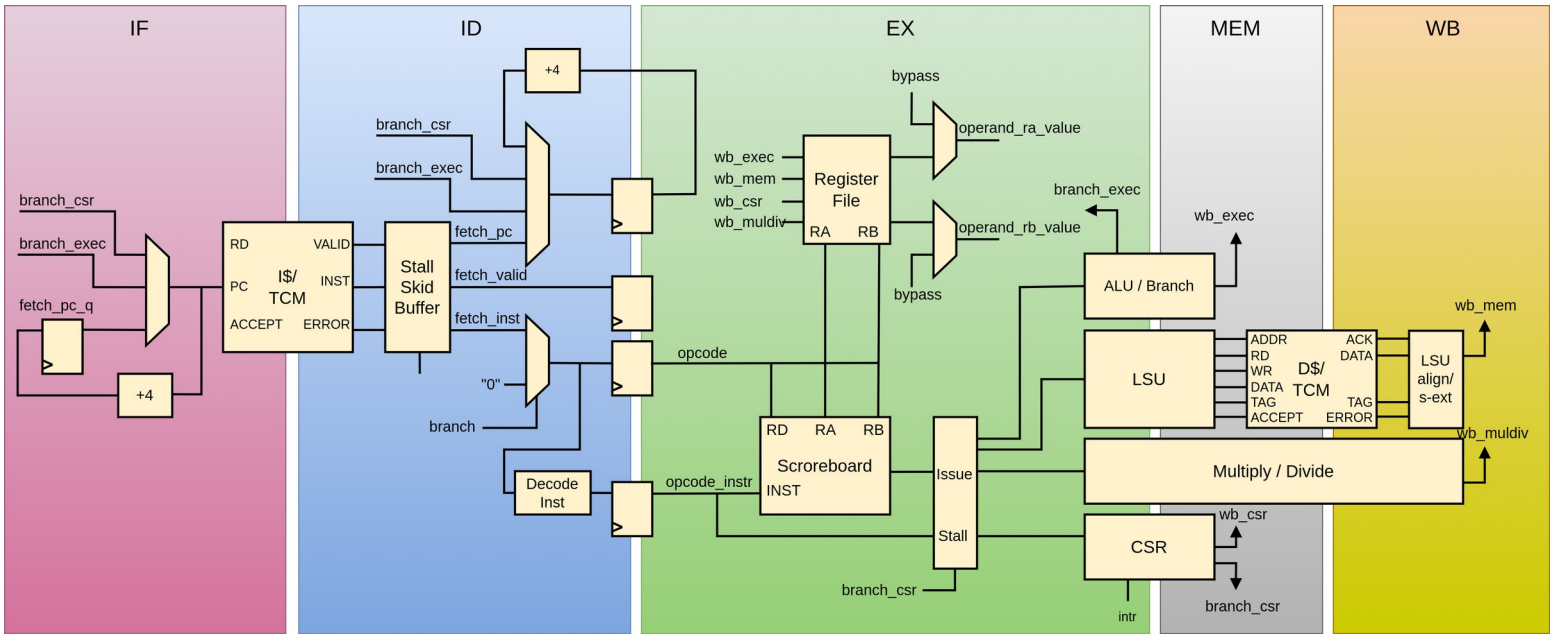

Figure 2: Architecture of VSRV1 and VSRV2 cores

2.2.1 Description of the 5-stage Pipeline

The VSRV1/2 pipeline consists of the following stages:

2.2.1.1 Instruction Fetch (IF)

- The **IF stage** sends the program counter (PC) to memory and fetches the current instruction.
- It updates the PC to point to the next instruction in sequence.

2.2.1.2 Instruction Decode (ID)

- The **ID stage** decodes the fetched instruction and reads the required registers from the register file.
- For branch instructions, it calculates the effective address for unconditional jumps.
- Instruction decoding and register reads are performed in parallel where possible.

2.2.1.3 Execute (EX)

- The **EX stage** executes instructions using the ALU, multiplier, and divider.
- Branch conditions are evaluated, and conditional branches are taken in this stage.
- Multi-cycle instructions stall the EX stage until completion.
- The EX stage also includes address generation for the load-store unit (LSU).

- Results from the ALU, multiplier, and divider are forwarded to the register file for write-back.

2.2.1.4 Memory (MEM)

- The **MEM stage** handles load and store instructions.
- For load instructions, data is read from memory at the effective address.
- For store instructions, data is written to the memory.

2.2.1.5 Writeback (WB)

- The **WB stage** writes the results of load instructions and other data from the EX stage back to the register file.

2.2.2 Architecture Differences Between VSRV1 and VSRV2 Cores

The differences between starting point [1], VSRV1 and VSRV2 are in extensions and data width of the AXI bus. Table 1 summarizes the differences.

Feature	Extension	[1]	VSRV1 core	VSRV2 core
Supervisor and User mode	-SU	yes	yes	yes
Integer	-I	yes	yes	yes
Integer multiplication and division	-M	yes	yes	yes
Instruction-fetch fence	-Zifencei	no	yes	yes
Control and Status Register Access / Privileged Architecture	-Zicsr	yes	yes	yes
Atomic instructions	-A (Zalrsc, Zaamo)	no	no	yes
Bit manipulation and address generation	- B (Zba, Zbb, Zbs)	no	no	yes
Bit manipulation for cryptography	-Zbkb	no	no	yes
Conditional arithmetic operations	-Zicond	no	no	yes
Width of AXI bus (affects on the MMU)		32 bits	32 bits	64 bits

Table 1: Differences between VSRV1 and VSRV2 RISC-V cores

Details of the implemented extensions are given in Table 2.

Feature	Extension	Detailed operations
Supervisor and User mode https://docs.riscv.org/reference/isa/priv/priv-intro.html	-SU	U, S, M levels
Integer https://docs.riscv.org/reference/isa/unpriv/m-st-ext.html	-I	integer multiply and divide
Integer multiplication and division	-M	

Feature	Extension	Detailed operations
Instruction-fetch fence https://docs.riscv.org/reference/isa/unpriv/zifencei.html	-Zifencei	• Fence.i
Control and status register access / privileged architecture https://www.five-embeddev.com/riscv-user-isa-manual/latest-adoc/zicsr.html	-Zicsr	• csrrw: Read/Write CSR • csrrs: Read/Set CSR bits • csrrc: Read/Clear CSR bits • Immediate versions of csrrwi, csrrsi, csrrcsi
Atomic instructions	-A	Zalrsc, Zaamo
Load-reserved/store-conditional https://github.com/riscv/riscv-zaamo-zalrsc/blob/main/zaamo-zalrsc.adoc	-Zalrsc	• LR • SC
Atomic memory operations https://github.com/riscv/riscv-zaamo-zalrsc/blob/main/zaamo-zalrsc.adoc	-Zaamo	• AMOSWAP • AMOADD • AMOXOR • AMOAND • AMOOR • AMOMIN • AMOMAX • AMOMINU • AMOMAXU
Bit manipulation and address generation	- B	• Zba, Zbb, Zbs
Address generation https://github.com/riscv/riscv-bitmanip/blob/main/bitmanip/zba.adoc	-Zba	• sh1add • sh2add • sh3add
Basic bit manipulation https://github.com/riscv/riscv-bitmanip/blob/main/bitmanip/zbb.adoc	-Zbb	• Logical with negate (andn, orn, xnor) • Count leading/trailing zero bits (clz, ctz) • Count population (cpop) • Integer minimum/maximum (max, maxu, min, minu) • Sign- and zero-extension (sext.b, sext.h, zext.h) • Bitwise rotation (rol, ror, rori) OR Combine (orc.b) • Byte-reverse (rev8)
Single-bit manipulation https://github.com/riscv/riscv-bitmanip/blob/main/bitmanip/zbs.adoc	-Zbs	• bclr • bclri • bext • bexti • binv • binvi

Feature	Extension	Detailed operations
		• bset • bseti
Bit manipulation for cryptography https://github.com/riscv/riscv-bitmanip/blob/main/bitmanip/zbkb.adoc	-Zbkb	• rol • ror • rori • andn • orn • xnor • pack • packh • rev.b • rev8 • zip • unzip
Conditional arithmetic operations https://github.com/riscvarchive/riscv-zicond/blob/main/zicondops.adoc	-Zicond	• czero.eqz • czero.nez

Table 2: Details of the implemented extensions

3 Design and Implementation

3.1 Design Methodology

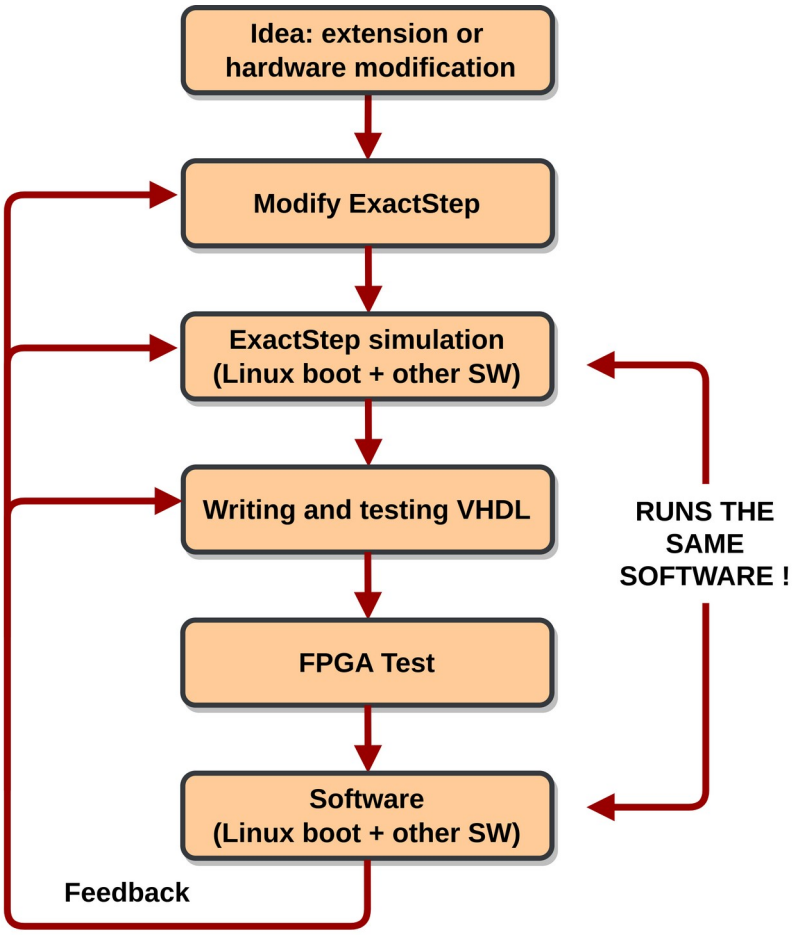

Figure 3: Design flow

The VSRV1/2 processors were designed as a **peripheral protocol processor**, intended to handle tasks such as network packet communication under a standard Linux operating system.

The primary design objective was to maintain **off-the-shelf Linux OS compatibility** while optimizing the trade-off between **moderate performance** and **logic gate utilization**.

3.1.1 Performance Evaluation and Benchmarking

Three benchmark tests were used to evaluate performance:

NBench [3] – measures overall computational performance

CoreMark [4] – evaluates CPU core efficiency

Linux boot time – provides a system-level measure of startup performance

For consistent and repeatable results, the Linux boot sequence was slightly modified by removing certain randomized self-tests.

3.1.2 Architecture Simulation and Optimization

The architecture of VSRV1 and VSRV2 was initially simulated and optimized using the ExactStep simulator [2] (https://github.com/ultraembedded/exactstep).

- The ExactStep **source code was modified** to reflect architectural changes.
- LPDDR memory, cache, and loop-control modules were added to the simulator to achieve close **cycle-accurate results**, which are difficult to obtain with simulators such as QEMU or Spike.
- ExactStep's simplicity allows **rapid experimentation**, enabling architectural modifications and execution of standard benchmarks like CoreMark.
- The simulator also evaluates **performance impact in the target application**.

Architectural improvements were selected for implementation based on:

- The improvement was **mandatory** for system performance or functionality.
- The improvement was **realistic** to implement within one year.

The detailed selection process is explained in Chapter 3.5.2 .

3.1.3 Implementation of Verification and Testing

The VSRV processor design flow included:

3.1.3.1 Block-Level Simulation

- Each processor block was partitioned and simulated individually.
- Verified blocks were integrated into the complete SoC system in VHDL.

3.1.3.2 System-Level Simulation

- RTL (Register-Transfer Level) VHDL simulation verified overall functionality.
- **System boot** was the primary verification scenario.

3.1.3.3 FPGA Prototyping

- After RTL verification, designs were transferred to FPGA platforms.
- FPGA testing focused on **hardware-software co-operation** and Linux OS compatibility, offering faster validation of software workloads than simulation.

3.1.3.4 Gate-Level Verification

- When synthesized to standard cells, the existing RTL testbench was reused for **gate-level simulation**.

3.1.3.5 Backend Layout

- Layout and physical design were completed using commercial EDA tools.

3.2 Pin-list

3.2.1 Pin List of riscv_top Module

PIN NAME	WIDTH	TYPE	NOTES	VSRV1	VSRV2
clk_i	1	in	Master clock	X	X
xrst_i	1	in	Synchronous reset	X	X
xrst_cpu_i	1	in	CPU xreset only	X	X
xrst_core_i	1	in	CPU/Cache xreset		X
TEST					
scanena_mem_i	1	in		X	X
scanin_mem_i	1	in		X	X
scanout_mem_o	1	out		X	X
RAM CONFIGURATION					
ram_delay_i	2	in	Select the rd/wr delay	X	X
ram_xpd_i	1	in	Power down the memory	X	X
MEM BIST INTERFACE (NOTE: mem count is 2x WAYS as there are tag/data RAMs) rd/wr select is 1-hot style in order like ...t1,d1,t0,d0					
mbist_ena_i	1	in		X	X
imbist_rd_i	4	in		X	X
imbist_wr_i	4	in		X	X
dmbist_rd_i	4	in		X	X
dmbist_wr_i	4	in		X	X

mbist_addr_i	13	in		X	X
mbist_data_i	W	in		W=32	W=64
mbist_data_o	W	out		W=32	W=64
AXI INTERFACE					
axi_i_awready_i	1	in		X	X
axi_i_wready_i	1	in		X	X
axi_i_bvalid_i	1	in		X	X
axi_i_bresp_i	2	in		X	X
axi_i_bid_i	4	in		X	X
axi_i_arready_i	1	in		X	X
axi_i_rvalid_i	1	in		X	X
axi_i_rdata_i	W	in		W=32	W=64
axi_i_rresp_i	2	in		X	X
axi_i_rid_i	4	in		X	X
axi_i_rlast_i	1	in		X	X
axi_d_awready_i	1	in		X	X
axi_d_wready_i	1	in		X	X
axi_d_bvalid_i	1	in		X	X
axi_d_bresp_i	2	in		X	X
axi_d_bid_i	4	in		X	X
axi_d_arready_i	1	in		X	X
axi_d_rvalid_i	1	in		X	X
axi_d_rdata_i	32	in		W=32	W=64
axi_d_rresp_i	2	in		X	X
axi_d_rid_i	4	in		X	X
axi_d_rlast_i	1	in		X	X
axi_i_awvalid_o	1	out		X	X
axi_i_awaddr_o	32	out		X	X
axi_i_awid_o	4	out		X	X
axi_i_awlen_o	8	out		X	X
axi_i_awburst_o	2			X	X
axi_i_wvalid_o	1	out		X	X
axi_i_wdata_o	W	out		W=32	W=64
axi_i_wstrb_o	Y	out		Y=4	Y=8
axi_i_wlast_o	1	out		X	X
axi_i_bready_o	1	out		X	X
axi_i_arvalid_o	1	out		X	X
axi_i_araddr_o	32	out		X	X
axi_i_arid_o	4	out		X	X
axi_i_arlen_o	8	out		X	X
axi_i_arburst_	2	out		X	X

o					
axi_i_rready_o	1	out		X	X
axi_d_awvalid_o	1	out		X	X
axi_d_awaddr_o	32	out		X	X
axi_d_awid_o	4	out		X	X
axi_d_awlen_o	8	out		X	X
axi_d_awburst_o	2	out		X	X
axi_d_wvalid_o	1	out		X	X
axi_d_wdata_o	W	out		W=32	W=64
axi_d_wstrb_o	Y	out		Y=4	Y=8
axi_d_wlast_o	1	out		X	X
axi_d_bready_o	1	out		X	X
axi_d_arvalid_o	1	out		X	X
axi_d_araddr_o	32	out		X	X
axi_d_arid_o	4	out		X	X
axi_d_arlen_o	8	out		X	X
axi_d_arburst_o	2	out		X	X
axi_d_rready_o	1	out		X	X
VSBUS					
pp_intr_i	1	in		X	X
pp_addr_o	32	out		X	X
pp_rd_o	1	out		X	X
pp_rd_data_i	32	in		X	X
pp_wr_o	1	out		X	X
pp_wr_data_o	32	out		X	X
DMA for memory-to-memory copying					
dma_id_i	1	in			X
dma_wr_i	1	in			X
dma_addr_i	30	in			X
dma_wr_data_i	32	in			X
dma_rd_data_o	32	out			X
dma_intr_o	1	out	dma interrupt		X
DMA for coprocessor (VSDSP6)					
dma3_id_i	1	in			X
dma3_wr_i	1	in			X
dma3_len_i	8	in			X
dma3_addr_i	30	in			X
dma3_wr_data_i	64	in			X

dma3_rd_data_o	64	out			X
dma_accept_o	1	out			X
dma_ack_o	1	out			X
dma_error_o	1	out			X
DMA for SDCARD					
sd_dmard_i	1	in			X
sd_dmawr_i	1	in			X
sd_len_i	8	in			X
sd_dmaaddr_i	32	in			X
sd_dmain_o	64	out			X
sd_dmaout_i	64	in			X
sd_accept_o	1	out			X
sd_ack_o	1	out			X
sd_error_o	1	out			X

Table 3: Pin list of riscv_top module

3.3 Register Map

3.3.1 Configuration Definitions of the VSRV1/2 Cores

The VSRV1/2 cores (riscv_core module of Figure 1) have multiple user-specified parameters (VHDL generics) that are set prior to synthesis/simulation. These parameters, their type, and default values are shown in Table 4.

Name	Bits	Default value	Description	VSRV1	VSRV2
BOOT_ADDR	32	0x80000000	Boot address	0x80000000	0x80000000
CPU_ID	32	0x00000000	Id of the CPU	0x00000000	0x00000000
CACHE_ADDR_L	32	0x80000000	cacheable address low limit	0x80000000	0x80000000
CACHE_ADDR_H	32	0xFFFFFFFF	cacheable address high limit	0xFFFFFFFF	0xFFFFFFFF
RISCV_M	1	1	mul / div extension	1	1
RISCV_A	1	1	Zalsrc + Zaamo extensions	0	1
RISCV_zalrsc	1	0	Obsolete, this feature is controlled by RISCV_A	0	1
RISCV_E	1	0	rf embedded extension, halves size of register files	0	1
RISCV_S_U	1	1	supervisor/user privilege levels	1	1
RISCV_Zba	1	1	zba extension	0	1
RISCV_ZB	1	1	Zbb + zbs + zbkb + Zicond extensions	0	1
RISCV_I	1	1	Always enable when RISCV_E is not set	0	1
RISCV_Zicsr	1	1	Always enabled	1	1
RISCV_Zifencei	1	1	Always enabled	1	1
SUPPORT_MMU	1	1	Always enabled for Linux	1	1
BYPASS_LOAD	1	1	Data load bypass	1	1
BYPASS_MULT	1	1	Multiplier bypass	1	1
EXTRA_DECODE_STAGE	1	1	Always enabled	1	1

Table 4: Configuration definitions of the VSRV1/2 cores

Based on the development history, VSRV1 and VSRV2 cores have their own repositories. However, both cores can be obtained from the VSRV2 repository by setting the configuration parameters as shown in Table 4.

3.3.2 Registers of the VSRV1/2 Cores

CSR REGISTERS		
Reset Value = 0x0000 0000		
Machine-Level CSRs (32-bit registers)		
Register	Address	Description (typical use)
MSTATUS	X"300"	Machine status register
MISA	X"301"	Machine ISA register, indicating supported ISA extensions
MIE	X"304"	Machine interrupt enable
MTVEC	X"305"	Machine trap vector
MSCRATCH	X"340"	Machine scratch register
MEPC	X"341"	Machine exception program counter
MCAUSE	X"342"	Machine exception cause
MTVAL	X"343"	Machine trap value (additional information about certain exceptions)
MIP	X"344"	Machine interrupt pending
MCYCLE	X"C00"	Machine performance counters
MTIME	X"C01"	Machine performance counters
MTIMEH	X"C81"	Upper 32 bits of performance counters
MHARTID	X"F14"	Machine hardware thread ID
MTIMECMP	X"7C0"	Timer compare register
MEDELEG	X"302"	Machine exception delegation register
MIDELEG	X"303"	Machine exception delegation register
Supervisor-Level CSRs (32-bit) (Some CSRs are shared with the Machine-Level)		
SSTATUS	X"100"	Supervisor status register
SIE	X"104"	Supervisor interrupt enable
STVEC	X"105"	Supervisor trap vector
SSCRATCH	X"140"	Supervisor scratch register
SEPC	X"141"	Supervisor exception program counter
SCAUSE	X"142"	Supervisor exception cause
STVAL	X"143"	Supervisor trap value
SIP	X"144"	Supervisor interrupt pending
SATP	X"180"	Supervisor address translation and protection register [31] MODE, MODE=1 uses sv32 address translation [30:22] ASID, address space identifier [21:0] PPN , physical page number of the root page table
DCACHE control CSRs		
DFLUSH	X"3A0"	Flush dcache
DWRITEBACK	X"3A1"	Writeback dcache line
DINVALIDATE	X"3A2"	Invalidate dcache line

Table 5: Registers of VSRV1/2 cores

3.3.3 Peripheral Registers of the VSRV1/2 Processors

The base address of peripheral registers is 0x10000000. The tables below have the base address included, i.e. they represent the actual address.

3.3.3.1 IRQ Registers

Address	Register	Bits	Field Name	Type	Description
0x1000_0000	IRQ_ISR	INTS_G-1:0	STATUS	RW	Pending interrupt (unmasked) bitmap
0x1000_0004	IRQ_IPR	INTS_G-1:0	PENDING	R	Pending interrupts (masked) bitmap
0x1000_0008	IRQ_IER	INTS_G-1:0	ENABLE	RW	Interrupt enable mask
0x1000_000C	IRQ_IAR	INTS_G-1:0	ACK	W	Bitmap of interrupts to acknowledge
0x1000_0010	IRQ_SIE	INTS_G-1:0	SET	W	Bitmap of interrupts to enable
0x1000_0014	IRQ_CIE	INTS_G-1:0	CLR	W	Bitmap of interrupts to disable
0x1000_0018	IRQ_IVR	31:0	VECTOR	RW	Highest priority active interrupt number
0x1000_001C	IRQ_MER	1	HIE	RW	Hardware interrupt enable
	IRQ_MER	0	ME	RW	Master enable

Table 6: IRQ registers

3.3.3.2 Timer Registers

Address	Timer Register	Bits	Field Name	Type	Description
0x1100_0008	TIMER_CTRL0	1	INTERRUPT	RW	Interrupt enable
	TIMER_CTRL0	2	ENABLE	RW	Timer enable
0x1100_000C	TIMER_CMP0	31:0	VALUE	RW	Match value
0x1100_0010	TIMER_VAL0	31:0	CURRENT	RW	Current timer value
0x1100_0014	TIMER_CTRL1	1	INTERRUPT	RW	Interrupt enable
	TIMER_CTRL1	2	ENABLE	RW	Timer enable
0x1100_0018	TIMER_CMP1	31:0	VALUE	RW	Match value
0x1100_001C	TIMER_VAL1	31:0	CURRENT	RW	Current timer value

Table 7: Timer registers

3.3.3.3 UART Registers

Address	Register	Bits	Field Name	Type	Description
0x1200_0000	ULITE_RX	7:0	DATA	R	Data byte
0x1200_0004	ULITE_TX	7:0	DATA	W	Data byte
0x1200_0008	ULITE_STATUS	4	IE	R	Interrupt enabled
	ULITE_STATUS	3	TXFULL	R	Transmit buffer full
	ULITE_STATUS	2	TXEMPTY	R	Transmit buffer empty
	ULITE_STATUS	1	RXFULL	R	Receive buffer full
	ULITE_STATUS	0	RXVALID	R	Receive buffer not empty
0x1200_000C	ULITE_CONTROL	4	IE	RW	Interrupt enable
	ULITE_CONTROL	1	RST_RX	RW	Flush RX Buffer
	ULITE_CONTROL	0	RST_TX	RW	Flush TX Buffer

Table 8: UART registers

3.3.3.4 SPI Registers

Address	Register	Bits	Name	Type	Description
0x1300_001C	SPI_DGIER	31	GIE	RW	Global interrupt enable
0x1300_0020	SPI_IPISR	2	STATUS	RW	TX FIFO empty interrupt status
0x1300_0028	SPI_IPIER	2	ENABLE	RW	TX FIFO interrupt enable
0x1300_0040,	SPI_SRR	31:0	RESET	RW	Software FIFO reset
	SPI_CR	0	LOOP	RW	Loopback enable (MOSI to MISO)
0x1300_0060	SPI_CR	1	SPE	RW	SPI Enable
	SPI_CR	2	MASTER	RW	Master mode (slave not supported)
	SPI_CR	3	CPOL	RW	Clock polarity
	SPI_CR	4	CPHA	RW	Clock phase
	SPI_CR	5	TXFIFO_RST	RW	TX FIFO reset
	SPI_CR	6	RXFIFO_RST	RW	RX FIFO reset
	SPI_CR	7	MANUAL_SS	RW	Manual chip select mode
	SPI_CR	8	TRANS_INHIBIT	RW	Transfer inhibit
	SPI_CR	9	LSB_FIRST	RW	Data LSB first (1) or MSB first (0)
	SPI_CR	0	RX_EMPTY	RW	RX FIFO empty
0x1300_0064	SPI_SR	1	RX_FULL	R	RX FIFO full
	SPI_SR	2	TX_EMPTY	R	TX FIFO empty
	SPI_SR	3	TX_FULL	R	TX FIFO full
0x1300_0068	SPI_DTR	7:0	TX_DATA	RW	Transmit data byte
0x1300_006C	SPI_DRR	7:0	RX_DATA	R	Receive data byte
0x1300_0070	SPI_SSR	0	VALUE	RW	Chip select value

Table 9: SPI registers

3.3.3.5 GPIO Registers

Address	Register	Bits	Name	Type	Description
0x1400_0000	GPIO_DIRECTION	31:0	OUTPUT	RW	0 = Input; 1 = Output
0x1400_0004	GPIO_INPUT	31:0	VALUE	R	Raw input status
0x1400_0008	GPIO_OUTPUT	31:0	DATA	RW	GPIO output value
0x1400_000C	GPIO_OUTPUT_SET	31:0	DATA	W	GPIO output mask - set for high
0x1400_0010	GPIO_OUTPUT_CLR	31:0	DATA	W	GPIO output mask - set for low
0x1400_0014	GPIO_INT_MASK	31:0	ENABLE	RW	GPIO Interrupt Enable Mask
0x1400_0018	GPIO_INT_SET	31:0	SW_IRQ	W	Write 1 to assert an interrupt
0x1400_001C	GPIO_INT_CLR	31:0	ACK	W	Write 1 to clear an interrupt enable
0x1400_0020	GPIO_INT_STATUS	31:0	RAW	R	Set if interrupt active (regardless of INT_MASK)
0x1400_0024	GPIO_INT_LEVEL	31:0	ACTIVE_HIGH	RW	GPIO Interrupt Level : 1 = active high / rising edge; 0 = active low / falling edge
0x1400_0028	GPIO_INT_MODE	31:0	EDGE	RW	GPIO Interrupt Mode : 1 = edge; 0 = level

Table 10: GPIO registers

3.4 Functional Description

3.4.1 Top Hierarchy of the VSRV1 Processor

The top-level hierarchy of the VSRV1 processor is shown below:

```
rv_top
   |--dbg_bridge                    : u_dbg
   |  |--dbg_bridge_uart            : u_dbg/u_uart
   |  |--dbg_bridge_fifo            : u_dbg/u_fifo_tx
   |  |--dbg_bridge_fifo            : u_dbg/u_fifo_rx
   |--riscv_soc (VSRV1 processor)   : u_soc
      |--riscv_top                  : u_soc/u_core
      |  |--riscv_core (VSRV1 core): u_soc/u_core/u_core
      |  |--icache                  : u_soc/u_core/u_icache
      |  |--dcache                  : u_soc/u_core/u_dcache
      |--vs_perips                  : u_soc/u_vs_perips
      |  |--vsirq_ctrl              : u_soc/u_vs_perips/u_vsirq_ctrl
      |  |--vsuart_lite             : u_soc/u_vs_perips/u_vsuart_lite
      |  |--vstimer                 : u_soc/u_vs_perips/u_vstimer
      |--axi_mem_if                 : u_soc/u_axi_mem_if
         |--axi4_lite_tap           : u_soc/u_axi_mem_if/u_axi_tap
         |--axi4_arb                : u_soc/u_axi_mem_if/u_arb
         |  |--axi4_arb_onehot4     : u_soc/...../u_arb/u_rd_arb
         |  |--axi4_arb_onehot4     : u_soc/...../u_arb/u_wr_arb
         |--axi4_retime             : u_soc/.../u_retime
            |--axi4retime_fifo2x46  : u_soc/...../u_write_cmd_req
            |--axi4retime_fifo2x37  : u_soc/...../u_write_data_req
            |--axi4retime_fifo2x6   : u_soc/...../u_write_resp
            |--axi4retime_fifo2x46  : u_soc/...../u_read_req
            |--axi4retime_fifo2x39  : u_soc/...../u_read_resp
```

Figure 4: Top Hierarchy of the VSRV1 processor

3.4.2 Top Hierarchy of the VSRV2 Processor

The top-level hierarchy of the VSRV2 processor is shown in below:

```
riscv_soc (VSRV2 processor)
    |--dbg_bridge                : u_dbg_bridge
    |  |--dbg_bridge_uart        : u_dbg_bridge/u_uart
    |  |--dbg_bridge_fifo        : u_dbg_bridge/u_fifo_tx
    |  |--dbg_bridge_fifo        : u_dbg_bridge/u_fifo_rx
    |
    |--riscv_top                 : u_core
    |  |--riscv_core (VSRV2 core): u_core/u_cpu
    |  |--icachew                : u_core/u_icache
    |  |  |--tag_ram             : u_core/u_icache/u_tag0    (18x512W)
    |  |  |--tag_ram             : u_core/u_icache/u_tag1    (18x512W)
    |  |  |--data_ram            : u_core/u_icache/u_data0   (32x8KW)
    |  |  |--data_ram            : u_core/u_icache/u_data1   (32x8KW)
    |  |--dcachew                : u_core/u_dcache
    |  |  |--tag_ram             : u_core/u_dcache/u_tag0    (19x512W)
    |  |  |--tag_ram             : u_core/u_dcache/u_tag1    (10x512W)
    |  |  |--data_ram            : u_core/u_dcache/u_data0   (8KW)
    |  |  |--data_ram            : u_core/u_dcache/u_data1   (8KW)
    |  |--dma                    : u_core/dma_1
    |  |--axi_wrapper            : u_core/u_axi
    |     |--rom32x16k           : u_core/u_axi/u_rvirom16k
    |
    |--vs_perips                 : u_vsperips
    |  |--vsirq_ctrl             : u_vs_perips/u_vsirq_ctrl
    |  |--vsuart_lite            : u_vs_perips/u_vsuart_lite
    |  |--vstimer                : u_vs_perips/u_vstimer
    |
    |--axi_mem_if                : u_axi_mem_if
       |--axi4_arb               : u_axi_mem_if/u_arb
       |  |--axi4_arb_onehot4    : u_axi_mem_if/u_arb/u_rd_arb
       |  |--axi4_arb_onehot4    : u_axi_mem_if/u_arb/u_wr_arb
       |--axi4_retime            : u_axi_mem_if//u_retime
```

Figure 5: Top Hierarchy of the VSRV2 processor

3.4.3 Functional Observations

3.4.3.1 Hierarchy Differences

- In VSRV1, the debug UART is one level higher.
- In VSRV2, the debug UART (dbg_bridge) is on the same hierarchy level as the CPU core. This change simplifies the debug functionality.

3.4.3.2 Debug UART (dbg_bridge)

- Special UART used to load the boot image and release the processor from reset.
- Provides debug/test access before the Linux kernel boots.

3.4.3.3 Submodules of riscv_soc

(1) External Memory Interface (axi_mem_if)

- Bridges cache RAMs and debug UART to external memory.
- Compatible with any memory type supporting an AXI4 interface.
- DMA traffic (VSRV2) also passes through this module.

(2) Peripherals (vs_perips)

- Minimum configuration for Linux OS requires interrupt controller (vsirq_ctrl) and UART (vsuart_lite).
- Additional peripherals (SPI, Ethernet, etc.) can be added.
- Simple bridge connects peripherals to the data bus.

(3) Memory Management Units (icache/icachew and dcache/dcachew)

- Instruction and data memory management is handled here.
- MMU is mandatory for running Linux.

3.4.4 Hierarchy of the VSRV1 Core

```
riscv_pkg                  (configuration/parameters file)
riscv_core (VSRV1 CORE)    :  /u_core
  |--riscv_decode          :  /u_core/u_decode
  |   |--riscv_decoder     :  /u_core/u_decode/u_dec
  |--riscv_exec            :  /u_core/u_exec
  |   |--riscv_alu         :  /u_core/u_exec/u_alu
  |--riscv_multiplier      :  /u_core/u_mul
  |--riscv_divider         :  /u_core/u_div
  |--riscv_mmu             :  /u_core/u_mmu
  |--riscv_lsu             :  /u_core/u_lsu
  |-- |-riscv_lsu_fifo     :  /u_core/u_lsu/u_lsu_request
  |--riscv_csr             :  /u_core/u_csr
  |   |--riscv_csr_regfile :  /u_core/u_csr/u_csrfile
  |--riscv_issue           :  /u_core/u_issue
  |   |--riscv_pipe_ctrl   :  /u_core/u_issue/u_pipe_ctrl
  |   |--riscv_regfile     :  /u_core/u_issue/u_regfile
  |--riscv_fetch           :  /u_core/u_fetch
```

Figure 6: Hierarchy of the VSRV2 core

3.4.5 Hierarchy of the VSRV2 Core

```
rv_pkg                    (configuration/parameters file)
riscv_core (VSRV2 CORE)   : u_cpu
  |--rv_fetch             : u_cpu/u_fetch
  |--rv_decode            : u_cpu/u_decode
  |  |--rv_decoder        : u_cpu/u_decode/u_dec
  |--rv_exec              : u_cpu/u_exec
  |  |--rv_alu            : u_cpu/u_exec/u_alu
  |  |--rv_multiplier     : u_cpu/u_exec/u_mul
  |  |--rv_divider        : u_cpu/u_exec/u_div
  |--rv_mmu               : u_cpu/u_mmu
  |  |--rv_lrsc           : u_cpu/u_mmu/u_rv_lrsc
  |  |--rv_amo            : u_cpu/u_mmu/u_rv_amo
  |--rv_lsu               : u_cpu/u_lsu
  |--rv_csr               : u_cpu/u_csr
  |  |--rv_csr_rf         : u_cpu/u_csr/u_csrfile
  |--rv_issue             : u_cpu/u_issue
     |--rv_pipe_ctrl      : u_cpu/u_issue/u_pipe_ctrl
     |--rv_regfile        : u_cpu/u_issue/u_regfile
```

Figure 7: Hierarchy of the VSRV2 core

3.4.6 Hierarchy of the Peripheral Devices of the VSRV2 Processor

```
vs_perips                    : u_vsperips
    |--vsirq_ctrl            : u_vs_perips/u_vsirq_ctrl
    |--vstimer               : u_vs_perips/u_vstimer
    |--vsuart_lite           : u_vs_perips/u_vsuart_lite
    |--vsspi_lite            : u_vs_perips/u_vsspi_lite
    |  |--spi_lite_fifo      : u_vs_perips/u_vsspi_lite/u_tx_fifo
    |  |--spi_lite_fifo      : u_vs_perips/u_vsspi_lite/u_rx_fifo
    |--vsgpio                : u_vs_perips/u_vsgpio
    |--uart16550             : u_vs_perips/u0_uart16550
    |  |--uart_pkg
    |  |--uart_transmitter   : u_vs_perips/u0_uart16550/transmitter
    |  |  |--uart_tfifo      : u_vs_perips/u0_uart16550/transmitter/fifo_tx
    |  |--uart_receiver      : u_vs_perips/u0_uart16550/receiver
    |     |--uart_rfifo      : u_vs_perips/u0_uart16550/receiver/fifo_rx
    |--uart16550             : u_vs_perips/u1_uart16550
    |  |--. . .
    |--uart16550             : u_vs_perips/u2_uart16550
    |  |--. . .
    |--rv_if                 : u_vs_perips/u_rv_if
    |--sdcard                : u_vs_perips/sdcard_i0
    |  |--sd_pkg
    |  |--ram_sd             : u_vs_perips/sdcard_i0/sd_ram_i0_32
    |  |--xperipsd_rv        : u_vs_perips/sdcard_i0/xperipsd_rv_1
    |     |--arbitrator      :
u_vs_perips/sdcard_i0/xperipsd_rv_1/arbitrator_i0
    |--eth_pp_top                : u_vs_perips/u_eth
       |--PHY_IF                 : u_vs_perips/u_eth/PHY_IF_i0
       |  |--PHY_IF_RX1          : u_vs_perips/u_eth/PHY_IF_i0/PHY_IF_RX1_i0
       |  |  |--RX_CRC_CHK       :
u_vs_perips/u_eth/PHY_IF_i0/PHY_IF_RX1_i0/RX_CRC_CHK_1
       |  |--PHY_IF_TX1          : u_vs_perips/u_eth/PHY_IF_i0/PHY_IF_TX_i0
       |     |--TX_CRC_GEN1      :
u_vs_perips/u_eth/PHY_IF_i0/PHY_IF_TX_i0/TX_CRC_GEN_i0
       |- MAC_RV                 : u_vs_perips/u_eth/MAC_RV_i0
       |  |--ETH_PP_RV           : u_vs_perips/u_eth/MAC_RV_i0/ETH_PP_RV_i0
       |  |--ETH_CPUMEM_ARB      :
u_vs_perips/u_eth/MAC_RV_i0/ETH_CPUMEM_ARB_i1
       |  |--rx_filter32_lite    : u_vs_perips/u_eth/MAC_RV_i0/filterlite_i0
       |  |--filtertrunc         : u_vs_perips/u_eth/MAC_RV_i0/truncfilter_i0
       |  |--RX_CTRL             : u_vs_perips/u_eth/MAC_RV_i0/RX_CTRL_i0
       |  |--TX_CTRL_RV          : u_vs_perips/u_eth/MAC_RV_i0/TX_CTRL_RV_i0
       |  |--MDIO                : u_vs_perips/u_eth/MAC_RV_i0/MDIO_i0
       |--eth_ram                : u_vs_perips/u_eth/u_eth_ram
```

Figure 8: Hierarchy of the peripheral devices of the VSRV2 processor

Note that SD-card (sdcard) and Ethernet (et_pp_top) modules (fast interfaces) are connected to the AXI bus instead of VSBUS and are not included in the deliveries.

3.4.7 System Boot

3.4.7.1 Debug UART

- A debug UART module is included in the design to upload software into the main memory (LPDDR).

3.4.7.2 Boot ROM

- VSRV1 does not include a boot ROM.
- VSRV2 includes a boot ROM, but it is not configured to perform automatic boot.

3.4.7.3 Reset Behavior

- While the system is in reset, the debug UART is enabled.
- The RISC-V core (riscv_core) and caches remain in reset.

3.4.7.4 Boot Procedure

- A compiled ELF file is uploaded to LPDDR via the AXI4 interface using the debug UART (which is connected to the laptop).
- Once the upload completes, a register controlling the reset of the riscv_core is cleared, releasing the core.
- The core's peripheral UART is switched to use the same pins previously occupied by the debug UART.
- At boot, the instruction and data caches initialize (flush) their tag memories.
- The core requests the first instruction from memory. Execution begins as soon as the first instruction is fetched.

3.4.7.5 Alternative Boot Procedure of the Demonstrator IC (Fast)

The demonstrator IC supports an alternative, faster boot sequence using the VSDSP6 DSP processor:

- The VSDSP6 DSP processor is released from the hardware reset.
- The DSP processor boots from its own ROM, which detects the **SPI Flash**.
- The SPI Flash contains boot code for the DSP processor
- The DSP processor boots
- The DSP processor's boot code reads the ELF file from the **SD-card** and writes it to the LPDDR memory.
- Once the ELF upload completes, the register controlling the reset of the RISC-V core (riscv_core) is cleared, releasing the core.
- The RISC-V core requests its first instruction from memory. Execution begins as soon as the first instruction is fetched.

3.4.8 VSRV1/2 Cores (riscv_core)

The VSRV1 and VSRV2 cores fetch instructions and data from memory and execute them according to the 32-bit RISC-V ISA. The hardware implementation details were described in Chapter 2.2 .

Referring to Figure 9, the functional pipeline stages can be summarized as:

FETCH (icache) -> DECODE -> ISSUE -> EXEC -> LSU -> MMU -> WRITEBACK (dcache)

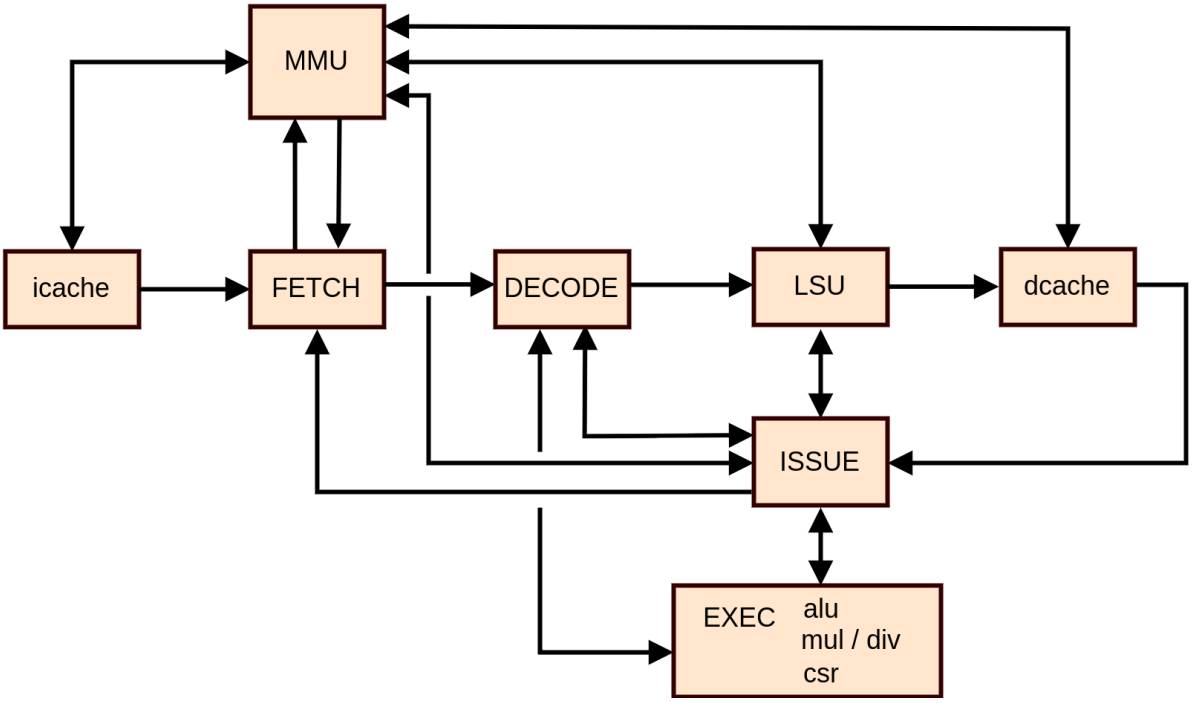

Figure 9: Functional block diagram of the VSRV1/2 core

3.4.8.1 Fetch Stage

- Fetches instructions from instruction memory using the current PC.
- When the MMU is enabled, address translation occurs before the fetch.
- The PC normally increments by 4, but redirects on branches, jumps (jal, jalr), or exceptions.
- Stalls occur on:
  - Memory wait states
  - MMU page misses
  - Pipeline backpressure

3.4.8.2 Decode Stage

- Decodes instruction fields and generates control signals.
- Determines instruction type (ALU, branch, load/store, CSR, etc.).
- Extracts source registers for operand values.
- Stalls occur if:
  - Source registers depend on results not yet written back
  - Instruction is long-latency

3.4.8.3 Issue Stage

- Acts as pipeline control and arbitration stage.
- Issues instructions to the correct execution unit (ALU, branch unit, LSU, CSR, etc.).
- Handles control flow for branches/jumps
- Flushes pipeline if branch/jump is taken
- Restarts fetch at the new PC
- Stalls occur if execution units or LSU/MMU are busy.

3.4.8.4 Execute Stage

- Performs the actual computation:
- ALU instructions: add, subtract, shifts, logic, comparisons, multiplication, division
- Branch instructions: condition evaluation, target address calculation
- Jump instructions: PC target generation
- CSR operations: read, modify, write control and status registers

3.4.8.5 Load/Store Unit (LSU)

- Handles data memory accesses.
- Determines:
  - Load or store operation
  - Access width (byte, halfword, word)
  - Sign extension for loads
- Stalls occur on MMU or memory response delays.

3.4.8.6 MMU Unit

- Performs virtual-to-physical address translation and access checking for both instructions and data.
- Operations include:
  - Address translation (virtual → physical)

    - Permission checks (read/write/execute, privilege level)
- Detects faults:
    - Page faults
    - Access faults
    - Misaligned accesses
- On a detected fault:
    - An exception is raised
    - Pipeline is flushed

3.4.9 Icache (icache / icachew)

The instruction cache submodule controls access between the processor core and SRAM cache memories. It is a 32-bit set-associative cache with parameters configurable via VHDL generics.

Parameter	Type	Default value in VSRV1	Default value in VSRV2
AXI_ID	std_logic_vector	0x8	0x8
WAYS	Natural range 1 to 8	2	2
LINE_ADDR_W (k)	Natural range 6 to 12	10	9
LINE_SIZE_W (n)	Natural range 2 to 8	5	6
BITS (b)	32 or 64	32 (m=2)	64 (m=3)

Table 11: Icache and dcache parameters

These default values implement a **2-way cache**:

- VSRV1: 28 lines with $2^{5-2} = 8$ words per line
- VSRV2: 29 lines with $2^{6-2} = 16$ words per line

Both cores have 64 kB RAM per cache.

3.4.9.1 Implementation Notes

The **VHDL code** allows:

- Conventional **single-port RAMs** for IC implementations.
- **Embedded memory** for FPGA implementations.

Each 2-way cache contains:

- **Two tag RAMs**
- **Two data RAMs**

RAMs must be **semi-synchronous**: address, data inputs, and RD/WR enable signals are registered on the clock edge.

Memory sizes depend on the cache parameters:

- **Tag RAM:**
  - VSRV1: 20-bit width × 256 entries
  - VSRV2: 18-bit width × 512 entries
  - Equation: (32-k–n+1) x (2^{k-1})
- **Data RAM:**
  - VSRV1: 32-bit × 8192 entries
  - VSRV2: 64-bit × 4096 entries
  - Equation for entries: 2^{k+n-m} , where $m=\log_2(b/8)$
- Since the instruction cache is read-only, the AXI4 write channel is omitted.

The detailed two-way cache implementation is illustrated in Figure 10.

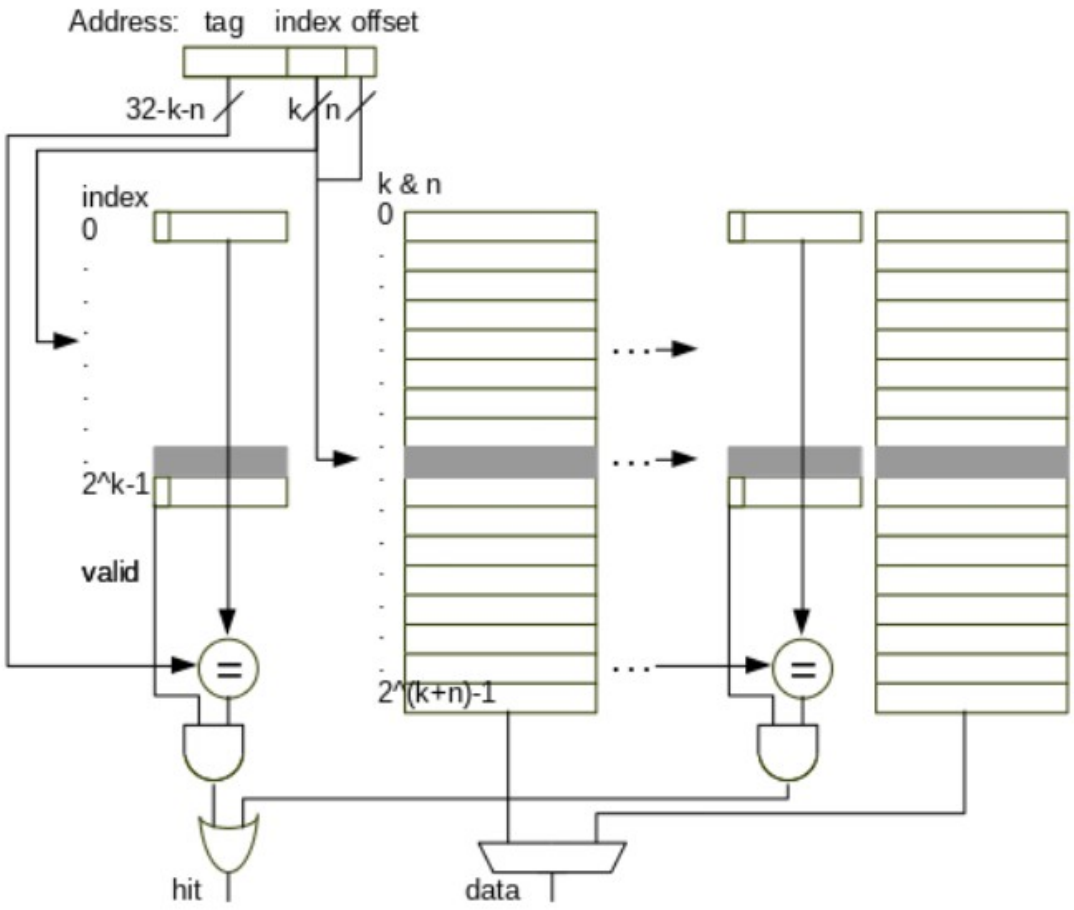

Figure 10: Two-way cache implementation

3.4.9.2 Cache Initialization and Control

After reset, the **tag RAMs must be initialized** before the cache becomes operational. The initialization process requires:

2^n clock cycles.

Once initialization is complete, the signal req_accept_o is asserted, indicating that the cache is ready to service core read requests.

Line Invalidation

Individual cache lines can be invalidated as follows:

- req_invalidate_i is asserted high for one clock cycle
- req_accept_o must be high at the same time

This operation invalidates the selected cache line without affecting other entries.

Cache Flush

The entire tag RAM can be flushed by:

- Asserting req_flush_i high for **one clock cycle**
- Ensuring the cache is ready to accept requests (req_accept_o high)

Flushing clears all valid bits in the tag RAM, effectively invalidating the entire cache.

The cache control state machine is illustrated in Figure 11.

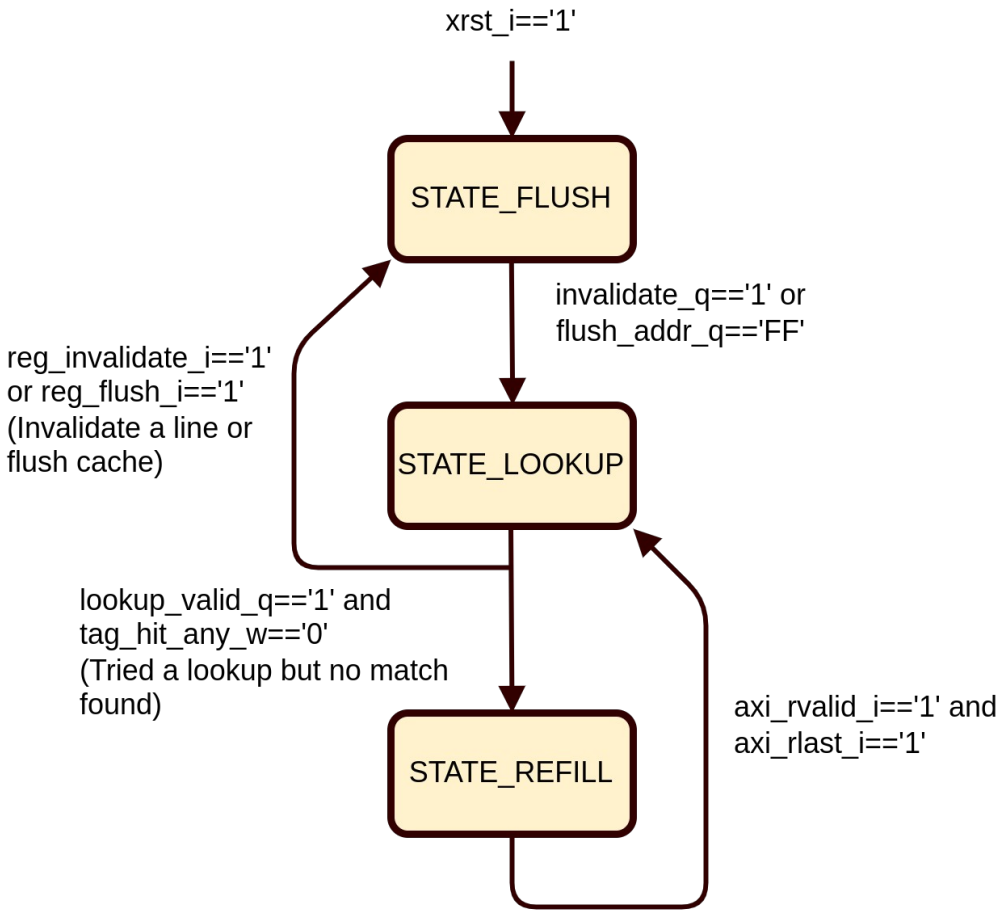

Figure 11: State diagram of icache

3.4.10 Dcache (dcache / dcachew)

The data cache is a set-associative cache with the same parameterization and memory configuration as the instruction cache described in Section 3.4.9 .

The configurable VHDL generics (AXI_ID, WAYS, LINE_ADDR_W, LINE_SIZE_W, BITS) and the overall memory sizing principles are identical to those of the instruction cache.

3.4.10.1 Architectural Differences Compared to Icache

Unlike the instruction cache, the data cache supports both read and write operations. Therefore, an additional control bit is required in the tag RAM:

- LINE_DIRTY flag

This flag indicates whether a cache line has been modified (written) by the processor and therefore differs from the corresponding data in main memory.

Tag RAM Contents

Each tag entry contains:

- Tag address bits
- Valid bit
- Dirty bit (LINE_DIRTY)

The dirty bit is used to determine whether a cache line must be written back to memory before being replaced.

3.4.10.2 Write Policy and Operation

Because the data cache supports writes:

- Cache lines may become dirty after store operations.
- On eviction of a dirty line, a write-back transaction is issued via AXI4.
- Clean lines can be replaced without write-back.

This introduces additional states in the cache control state machine compared to the instruction cache.

The data cache state machine is illustrated in Figure 12.

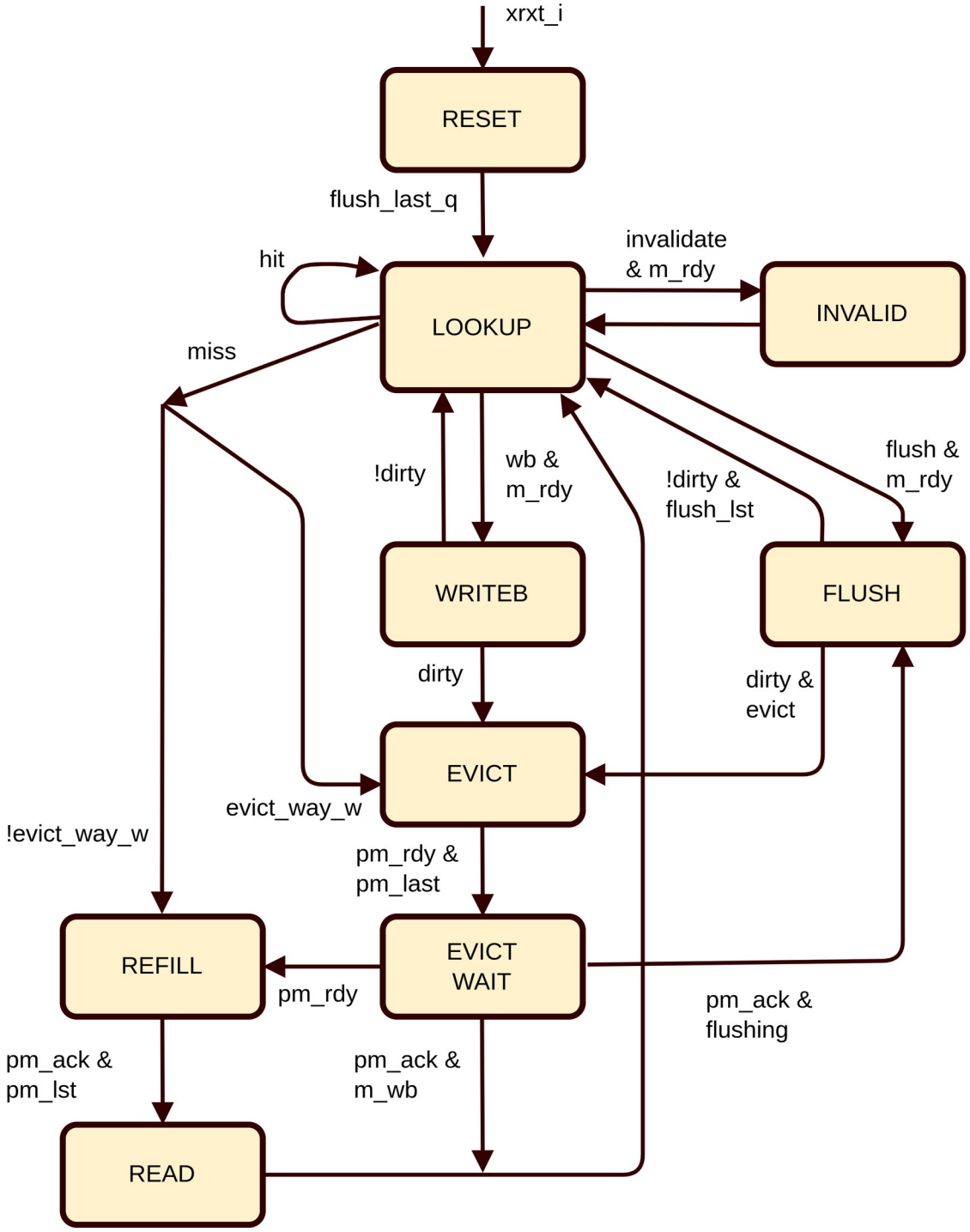

Figure 12: State diagram of dcache

3.4.11 AXI Memory Interface (Axi_mem_if)

The AXI memory interface (axi_mem_if) connects both the instruction cache and the data cache to the external memory subsystem using a standard AXI4 bus protocol.

3.4.11.1 Instruction Cache Interface

The instruction cache is a read-only cache. Therefore, it utilizes only the AXI4 read channel signals:

- Read address channel (AR)
- Read data channel (R)

The AXI4 write channels are not implemented for the instruction cache.

3.4.11.2 Data Cache Interface

The data cache supports both read and write operations and therefore uses the full AXI4 interface:

- Read address channel (AR)
- Read data channel (R)
- Write address channel (AW)
- Write data channel (W)
- Write response channel (B)

Write-back operations are issued through the AXI4 write channels when dirty cache lines are evicted.

3.4.12 Peripheral Interface (Vs_perips)

The vs_perips submodule provides access to the processor's peripherals through a simple bridge interface.

3.4.12.1 VSBUS Protocol

- The interface is compliant with the bus used in VSDSP6, a proprietary DSP processor developed by VLSI Solution.
- It supports single-word, single-cycle transactions only.
- Peripherals cannot halt the processor. All operations are non-blocking from the core perspective.

3.4.12.2 Connection to Processor Core

- The riscv_core and data cache (dcache) are connected to the peripherals through a small multiplexer-style bridge.
- This bridge alternately selects traffic between the core and data cache to access the vs_perips submodule.

3.4.12.3 Other features

- This interface is designed for simplicity and low-latency peripheral access.
- Minimal arbitration is needed since only single-cycle transactions are supported.

Timing of the VSBUS is shown in Figure 13.

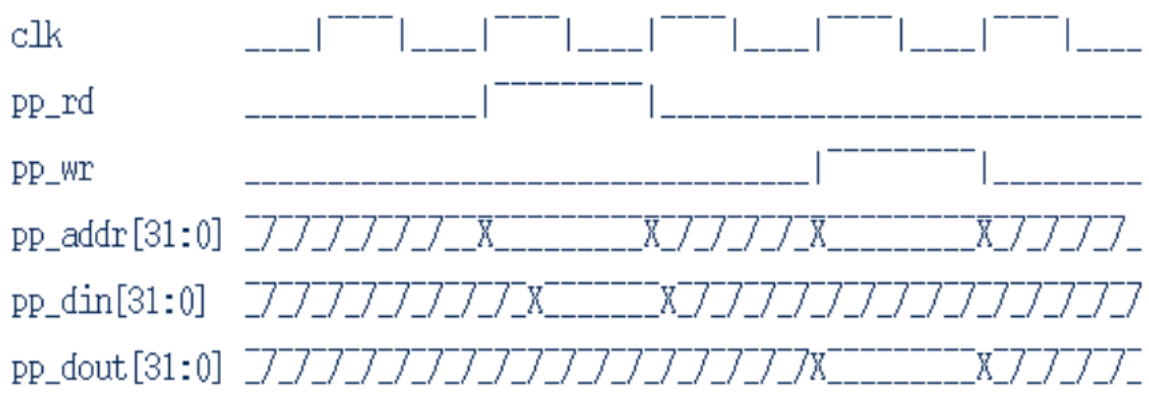

Figure 13: Timing diagram of vs_periph interface

3.5 PPA Analysis

3.5.1 Gate Count

The synthesis results of the riscv_core block are compared below:

- The starting point **UltraEmbedded scalar RISC-V core** [1]
- The riscv_core block of **VSRV1**
- The riscv_core block of **VSRV2**
- The **Core-v-Wally** core (with similar architecture and synthesis configuration)

Type	Icache Size (KiB)	Number of Registers	Equivalent NAND Gates
Starting point (UltraEmbedded [1])	8	3,400	13,993
riscv_core (VSRV1 core + MMU)	128	3,208	13,373
riscv_core (VSRV2 core + MMU)	128	3,939	16,043
Core-v-Wally (Note 1)	128	5,657	21,323

Table 12: Silicon area comparison of the RISC-V cores

Note 1: Modifications of the default Core-v-wally parameters are given in Table 13. Note that cache could not be made as big as it is in VSRV1/2 cores.

Parameter	Original default value	Updated value for synthesis
ZICBOM_SUPPORTED	0	1
ZICBOZ_SUPPORTED	0	1
ZICBOP_SUPPORTED	0	1
SVINVAL_SUPPORTED	0	1
SVADU_SUPPORTED	0	1
DCACHE_SUPPORTED	0	1
ICACHE_SUPPORTED	0	1
VIRTMEM_SUPPORTED	0	1
ITLB_ENTRIES	32'd1	32'd2
DTLB_ENTRIES	32'd1	32'd2
DCACHE_NUMWAYS	32'd2	32'd2
ICACHE_NUMWAYS	32'd2	32'd2
DTIM_SUPPORTED	1	0
IROM_SUPPORTED	1	0

Table 13: Modified parameters of Core-v-wally core for comparison

3.5.2 Architecture Development Steps Based on ExactStep Simulator

Linux boot time [s]	Nbench memory	Nbench integer	Nbench Floating-point	Coremark /MHz	Clock MHz	NOTE
332.55	0.016	0.024	0.019	0.1	50	Starting point [1]: First Linux running version: 8KiB ICache, No DCache, 50 MHz CPU, 17 MHz LPDDR1
67.883	0.095	0.160	0.129	1.06	50	4 KiB DCache
55.060	0.124	0.190	0.145	1.16	50	8 KiB DCache, MMU
40.327	0.154	0.221	0.183	1.30	50	25 MHz LPDDR1
34.494	0.181	0.247	0.219	1.42	50	16+16 KiB I+D Cache
28.294	0.208	0.277	0.255	1.52	50	32+32 KiB I+D Cache
23.270	0.225	0.287	0.273	1.58	50	50 MHz LPDDR2
9.846	0.239	0.295	0.289	1.72	50	64+64 KiB I+D Cache, non-compressed Linux (=FPGA proto)
4.587	0.497	0.618	0.601	1.68	100	100 MHz CPU + LPDDR2 (=VSRVES01 chip proto specification)
2.317	0.718	0.640	0.734	1.72	110	VSRV1 core in VSRVES01 chip, measured
2.317	0.940	0.641	0.731	1.83	110	+ compiler parameter optimization
1.842	0.940	0.642	0.730	1.83	110	+ A-extension (RV32IMA)
1.795	1.008	0.708	0.731	1.95	110	+ B-extension + zbkb
1.667	1.008	0.708	0.731	1.95	110	Cache line from 32 to 64 bytes
1.400	1.024	0.710	0.733	1.96	110	Double-speed DDR (=VSRVES02 chip proto specification)
TBD	TBD	TBD	TBD	TBD	TBD	VSRV2 core in VSRVES02 chip, measured

Table 14: Performance development improvement steps of the cores

Notes:

1. MEM, INT, FP: NBench Memory, Integer, and Floating-Point performance compared to ideal performance (1 operation per clock cycle) running at 100 MHz.
2. ExactStep simulator original code was modified to include realistic model of the DDR memory, cache and pipeline control.

4 Verification and Validation

4.1 Verification Methodology

The market applications of the VSRV1/VSRV2 cores are expected to be in the consumer and industrial sectors. These segments do not require the same level of certification-oriented verification as safety-critical domains such as some automotive or space applications.

It should also be noted that Tristan is a research project, and the available resources do not allow for complete commercialization-level verification processes. Therefore, the verification effort focuses primarily on:

- Functional correctness
- Performance validation
- Hardware–software co-operation

4.1.1 Verification Flow

The verification strategy follows a two-stage approach:

(1) RTL Functional Verification

- Individual RTL blocks are first verified using dedicated functional test benches.
- Emphasis is placed on correct logical behavior and corner-case handling.

(2) FPGA Prototyping

- After RTL verification, the design is synthesized to FPGA.
- Verification focus shifts to system-level behavior using real software.
- Performance measurements are carried out in realistic execution environments.

This flow enables:

- Parallel development of hardware and demo software
- Early performance evaluation
- Early identification of system-level issues
- Close collaboration between hardware and software developers

4.1.2 Silicon Prototyping

The VSRV1 and VSRV2 IPs were prototyped in two SoC chips (VSRVES01 and VSRVES02)

The engineering samples of this work item provide:

- Silicon-proven IP
- Silicon-validated performance metrics

The related demo PCBs allow low threshold to try and become familiar with the developed IP.

4.2 Testbench Architecture

The design flow was introduced in Chapter 3.1 . Verification is an integral part of this flow. Figure 14 shows a redrawn version of Figure 3 with verification-specific details added.

4.2.1 Conversion and Logic Equivalence

When converting and optimizing the starting point from Verilog to VHDL, the following steps were applied:

(1) Logic Equivalence Checking (LEC)

- The starting point of the project is a scalar version of the Ultraembedded core [1]. It is a pre-verified core. It has been verified against Google's RISCV-V-DV random instruction sequence by using co-simulation against C++ ISA model of ExactStep simulator [2].
- Commercial LEC tools ensured that the VHDL implementation is logically identical to the original Verilog core.

(2) Iterative Architectural Simulation

- The ExactStep simulator [2] was used to test architectural modifications.
- Multiple experiments were performed to select optimal architectural candidates.
- Changes were carefully validated to avoid violations of the ratified RISC-V ISA specification.

(3) Hardware Block Development

- Approved modifications were converted into block-level hardware specifications.

- RTL for new entities was first simulated independently, then integrated into the processor system.
- The complete processor was synthesized to FPGA for system-level verification.

4.2.2 Three-Step Verification Approach

Verification was performed in three main stages:

(1) Functional Verification at Block Level

- VHDL code was verified against designer-created, block-specific test benches.
- Each new instruction was tested with a limited set of carefully chosen operands to verify correct functionality.

(2) FPGA-Based Extension Verification

- FPGA prototypes allowed large-scale testing with many operand combinations.
- Verification methodology:
  - Generate a reference trace file by running a test program on a Linux image where the specific extension is emulated in software.
  - Run the same program on the FPGA with hardware support for the extension enabled.
  - Compare results against the reference trace.
- Assembly language programs were often written manually to cover corner cases. Alternatively, computer-generated programs with random operands could be used.
- For cryptographic extensions, OpenSSL benchmarking [5] was also used to:
  - Measure achievable performance
  - Verify signature correctness across all instructions and operands
- The expected output signatures remain identical whether the extension is software-emulated or hardware-accelerated; only the execution time differs.

(3) System-Level Verification

- Larger software, such as a Linux boot, was executed on FPGA.
- Results were compared with ExactStep simulations to verify hardware-software co-operation.
- This step is critical for:
  - Peripheral devices
  - Identifying hardware improvements suggested by software developers
- Matching hardware to existing software drivers is generally easier than rewriting drivers for Linux.
- FPGA prototyping also allowed fast performance validation using the same software used in simulations.

4.2.3 Long-Term Validation and Stress Tests

Once the system demonstrated stable HW-SW co-operation, extended validation was performed:

- Ethernet stress tests: Millions of packets transmitted under intranet traffic to measure packet loss.
- Long-term Linux operation: Evaluation boards ran Linux continuously for over 240 days without any /var/log/ issues reported.

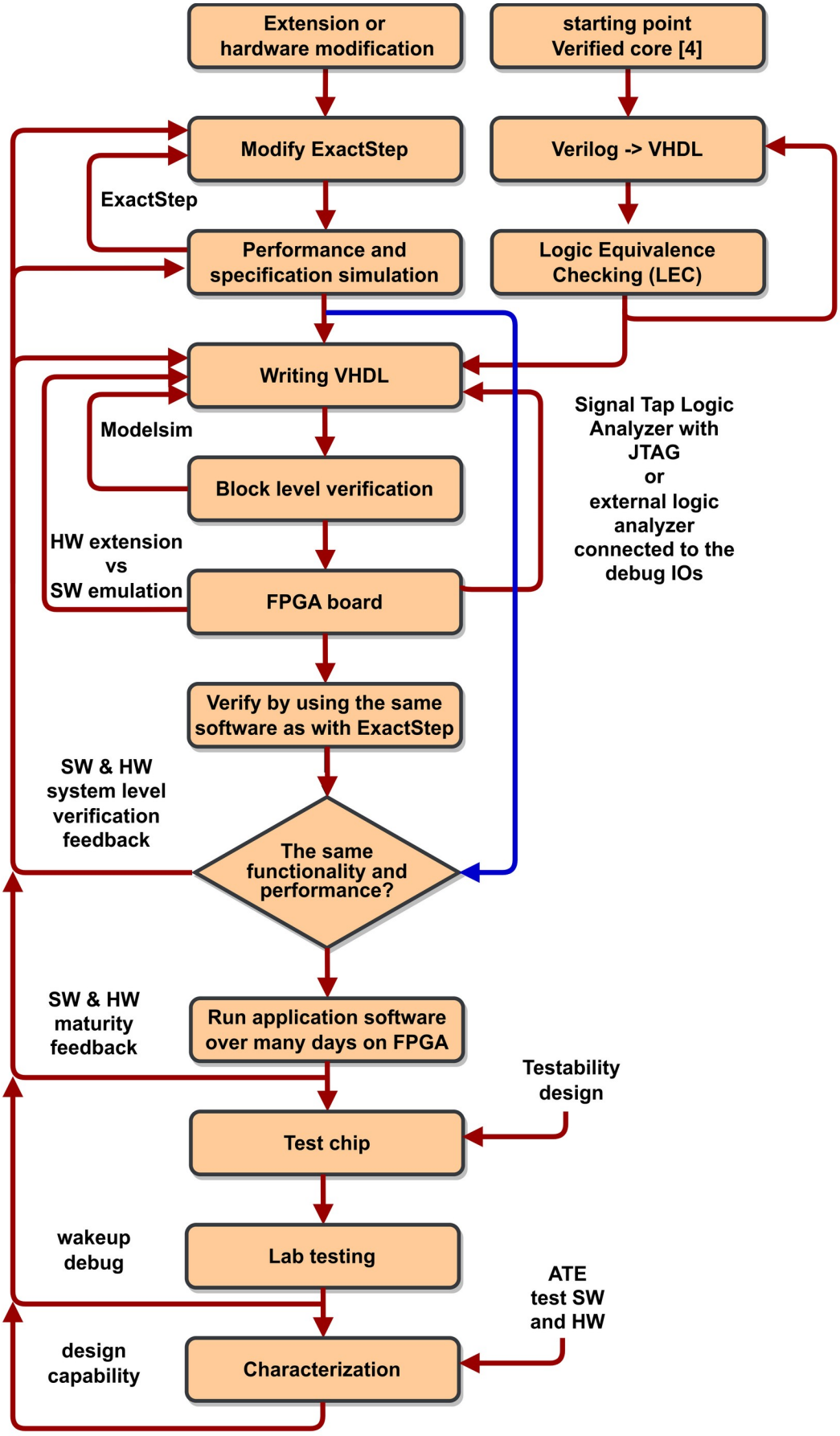

Figure 14: Verification flow

4.3 Prototyping Architecture

Prototyping of the VSRV1/2 cores is primarily based on FPGA implementation.

An in-house FPGA board (shown in Figure 15) was used for system-level validation, debugging, and performance evaluation.

4.3.1 FPGA-Based Verification Environment

The FPGA prototyping platform supports multiple debug and verification methods:

(1) On-Chip Debugging

- The Signal Tap Logic Analyzer integrated into the FPGA was used to capture internal signals.
- Debug data is accessed via JTAG.
- Captured traces were compared against:
  - RTL simulation results
  - ExactStep simulation results

(2) External Logic Analysis

- An external Logic Analyzer was connected through a wide parallel bus routed to the FPGA I/O pins.
- This enabled timing-accurate trace comparison between:
  - FPGA execution
  - RTL simulation
  - ExactStep architectural simulation

4.3.2 FPGA Board Features

The FPGA board includes the following functional elements:

- **FPGA configuration loading**
  - Configuration images are stored on a SD card (bottom edge of the board).
  - A VS1010 chip on the board loads the FPGA image.
  - Image selection is done using push-buttons and a small display connected to the VS1010.
- **Linux boot image**
  - The Linux image is loaded from a separate SD card located on the top edge of the board.
  - Boot time is only a few seconds.

- **Ethernet testing**
  - A separate green daughter board provides the physical Ethernet interface.
  - This enabled testing of real-time Ethernet traffic and stress conditions.
- **Debug user interface**
  - UART ports are used for console output and debugging.
  - A PC connects via USB-to-UART converters using text-based terminals.

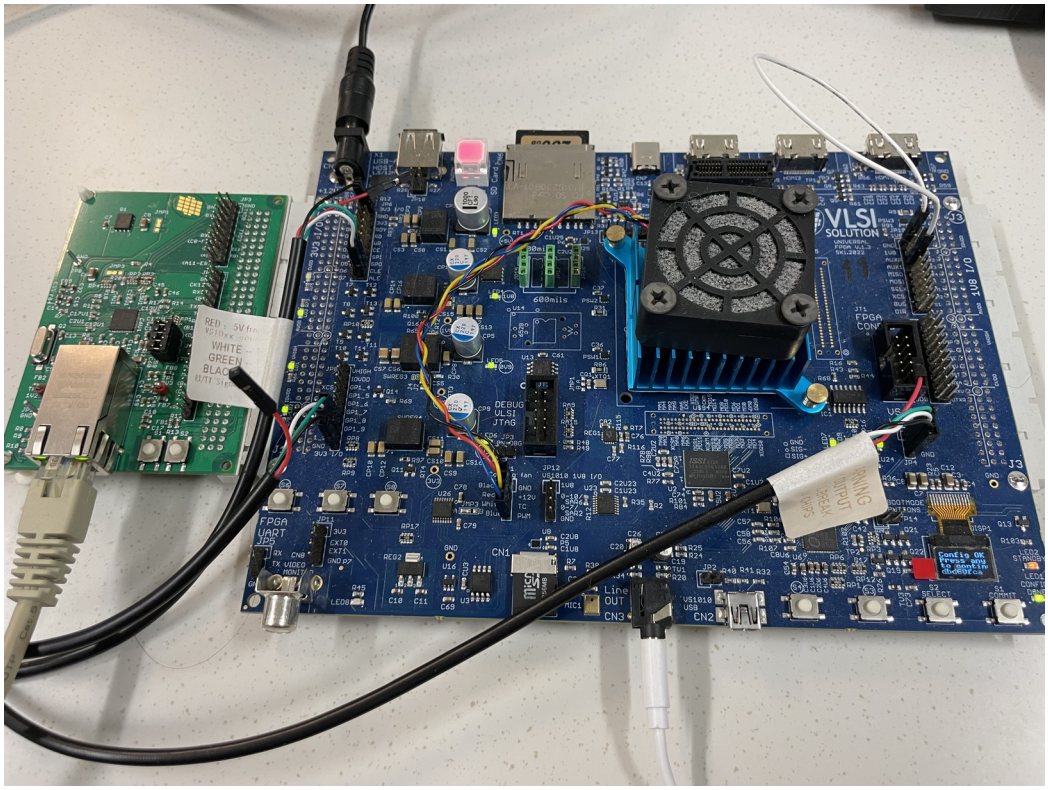

Figure 15: An in-house FPGA board used for system-level validation, debugging, and performance evaluation.

4.3.3 Prototype ICs

In addition to FPGA prototyping, two prototype ICs were developed in WI6.2.3 (Tristan WP6) to validate real silicon implementation:

- **VSRVES01**
  - Demonstrates the VSRV1 core
  - Already available for demonstration
- **VSRVES02**
  - Demonstrates the VSRV2 core
  - Expected to be available by the end of the project

Demonstration activities are reported in Tristan project WI6.2.3.

4.3.4 ATE Test Setup

A complete Automated Test Equipment (ATE) setup was developed for the first prototype IC (see Figure 16).

The setup consists of:

- DUT board with test socket and handler adaptation plate
- General-purpose interfacing load board
- High-performance coaxial cable connection
- Dedicated ATE test program

The setup was used to:

- Test IC samples for assembly onto demonstration boards (WP6)
- Provide silicon feedback to designers for the second demonstrator IC

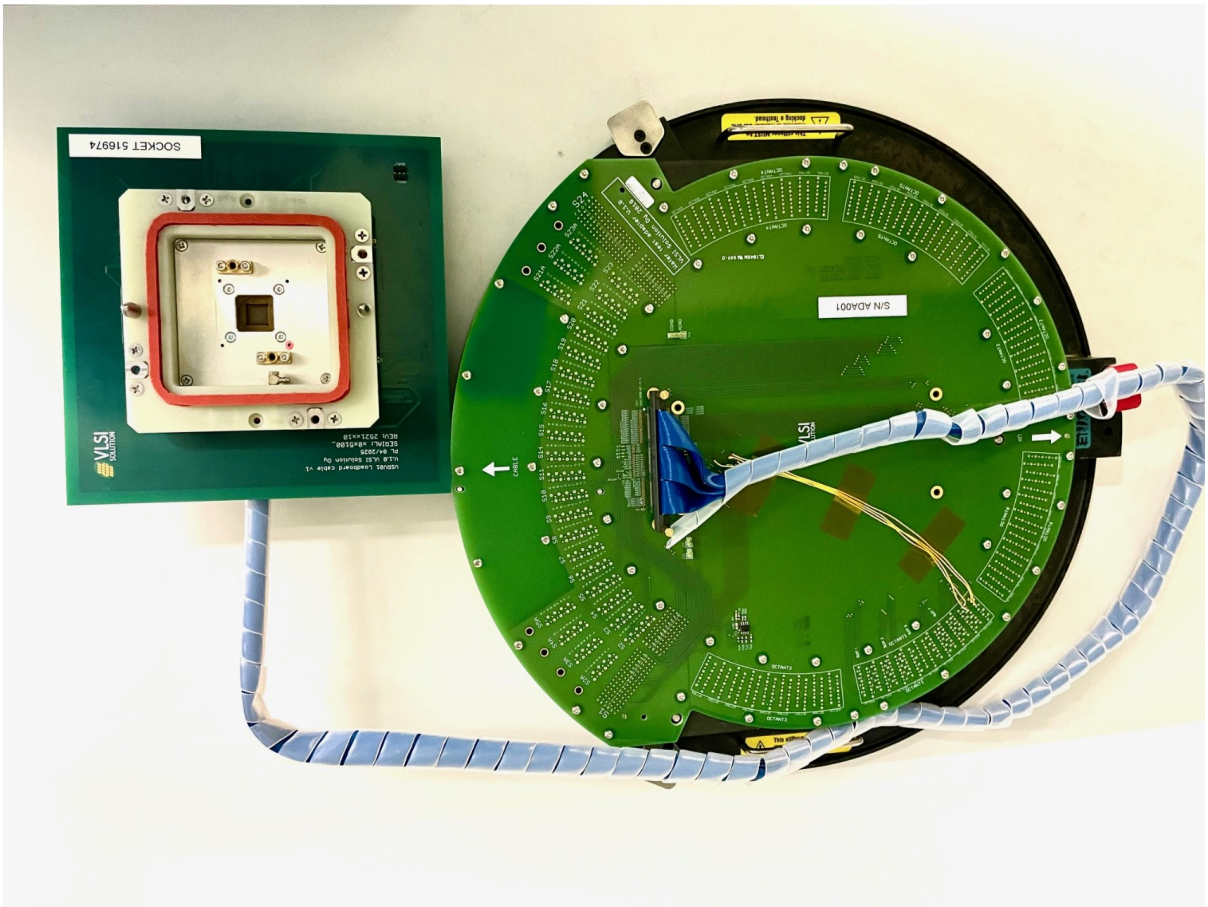

Figure 16: ATE setup of VSRVES01 prototype IC

4.3.5 Electrical and Temperature Characterization

During the R&D phase, the most important validation activity with the ATE setup is electrical temperature characterization.

Test conditions:

- Temperatures (by using forced-air thermostream equipment):
  - –30°C
  - +25°C
  - +85°C

Supply voltages (per supply):

- Minimum
- Nominal
- Maximum

This results in approximately 10 voltage combinations per temperature point.

The characterization program effectively exercises nearly every transistor in the IC.

In electrical temperature characterization:

- Test time is not critical.
- Extensive shmoo sweeps are performed to determine:
  - Functional limits
  - Performance boundaries

Detailed results are outside the scope of this project, but selected findings are summarized in Table 15 of Chapter 4.4 .

4.4 Test Results

Test	Description	Result
Block level functional test by using VHDL test bench	Block level functional verification by using RTL simulation	All pass
Trace of extension specific test, implemented HW vs SW emulation	Verify that result of RISC-V HW extension matches the result of SW emulation by using FPGA	All pass
Linux boot trace	Verify that Linux boot sequence of FPGA matches ExactStep simulation	Pass
Netlist vs. RTL	Verify that simulation result of the netlist of the IC layout matches the VHDL RTL simulation	All pass
LVS	Verify that the final layout of proto IC matches the netlist of it	All pass
Performance test by using Linux boot, Nbench, and CoreMark	Verify that FPGA and chip performance are as expected by ExactStep simulation	VSRV1 all pass (FPGA, VSRVES01 IC), VSRV2 FPGA pass, VSRVES02 IC not yet available, see Table 14.
Ethernet lost packet test of FPGA	Load ethernet maximally over millions of ethernet packets to estimate the packet loss	VSRVES02 FPGA: <3ppm lost packets in highly loaded ethernet
Long term reliability of the proto IC	Estimate maturity of HW+SW for Linux OS	VSRVES01: 240 days since boot and Linux system reports no issues
ATE characterization by using thermostream form -30°C...+85°C	Estimate the robustness of the design in process corners, operating voltages and temperatures	VSRVES01 prototype: • Core VDD can be from 1.5V down to 1.05V. • Max clock speed is about 110 MHz (by using 110nm technology) at 1.25V • Typical dynamic power consumption with maximum activity of VSRV1 core at 100MHz is about 2.8 mW • Scan chain ok. • LPDDR2 timing to external memory ok.

Table 15: Main test results of VSRV1/2 cores

5 Tools

Tool	Purpose	Type + name/suppler of the tool
ExactStep	System level simulation and optimization	Free and open source
VHDL/Verilog simulator	RTL simulation	Commercial, mostly Modelsim
Lint and compliance for synthesis, synthesis logs	Reduce synthesis errors and functional errors/weaknesses of the VHDL code	check_synthesis and -lint options of Modelsim
Synthesis	Synthesis of VHDL to FPGA or std cell library	Commercial, for IC: Cadence VDI, for FPGA: Altera Quartus
Logic Equivalence Checker	Verify that conversion from Verilog to VHDL is equivalent with the original functionality	Commercial, Cadence Conformal LEC
Emacs	editor	Free and open source
Libreoffice	Documentation (this document)	Free and open source
LaTeX	Documentation (data sheets)	Free and open source

Table 16: Main tools used for the design

6 References

[1]	“RISC-V Core” https://github.com/ultraembedded/riscv
[2]	“ExactStep - Instruction Accurate Instruction Set Simulator” https://github.com/ultraembedded/exactstep
[3]	https://github.com/petabridge/NBench
[4]	https://github.com/eembc/coremark
[5]	https://openbenchmarking.org/test/pts/openssl