VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

Logic Design Project Report

SRAM interface with RISC-V

Instructor: Mr Phạm Kiều Nhật Anh

Students: Lê Chương Quyền - 2353034
Nguyễn Tuấn Ngọc - 2352815
Phạm Lê Minh Khôi - 2352622

Ho Chi Minh City, 12 / 2025.

Group Members

Full Name	Student ID
Phạm Lê Minh Khôi	2352622
Lê Chương Quyền	2353034
Nguyễn Tuấn Ngọc	2352815

Source code

https://drive.google.com/drive/folders/1y-7fm55cTjYmG7Dko5NyFLDZGATGITfo?usp=sharing

Contents

1 Introduction

1.1 Project Title

This project corresponds to the topic "SRAM interface with RISC-V". The goal is to design an SRAM module in Verilog, connect it to a RISC-V core, run a program on that core, and demonstrate that the SRAM correctly exchanges data with the CPU on an FPGA board using LEDs or an LCD.

1.2 Topic Introduction

In many digital systems, a processor core communicates with external and on-chip memories through a well-defined bus interface. Designing such an interface is a fundamental step toward building a complete System-on-Chip (SoC). RISC-V is an open instruction set architecture that is well suited for educational SoC projects because a variety of small, synthesizable cores are available.
In this project, we implement a small SoC on an FPGA that integrates:

- a lightweight RISC-V core (PicoRV32, functionally similar to other low-resource cores such as iBex),
- an SRAM controller implemented in Verilog and mapped onto the FPGA's Block RAM,
- simple memory-mapped peripherals such as LEDs (and optionally an LCD).

The RISC-V program runs from instruction memory, reads and writes data via the SRAM interface, and uses memory-mapped registers to drive LEDs. Correct operation of the SRAM interface is validated by running a test program and observing the results on the FPGA board.

1.3 Tools Used

To design, implement and verify the system, the following tools are used:

- **Hardware description:** Verilog HDL for all digital modules (SRAM controller, UART, top-level SoC integration, etc.).
- **FPGA design suite:** An FPGA synthesis, placement and routing tool (e.g. Xilinx Vivado) to generate the bitstream for the target board.
- **Simulation tools:** A Verilog simulator (e.g. ModelSim, Vivado Simulator) to functionally verify the SRAM interface and RISC-V system before programming the FPGA.
- **RISC-V software toolchain:** GCC-based RISC-V cross-compiler to compile C code into an ELF and binary image for the PicoRV32 core.
- **Python utilities:** Helper scripts (such as `make_mem.py`) to convert compiled binaries into memory initialisation files (`.mem`) that can be loaded into FPGA Block RAM.

1.4 Equipment Used

The practical implementation and testing are carried out on the following hardware:

- **FPGA development board** with on-chip Block RAM and user LEDs, used to host the RISC-V core, SRAM controller and peripherals.
- **USB programming cable** for downloading the FPGA bitstream and establishing a serial connection with the PC.
- **Host PC or laptop** running the FPGA tools, RISC-V toolchain, and Python scripts for sending/receiving data.
- **Optional LCD module** (if used) to display status messages or data values beside the LEDs.

1.5 Product Functions

The final "product" can be viewed as a small RISC-V based SoC implemented on FPGA with the following main functions:

- **Instruction execution:** The RISC-V core (PicoRV32) fetches instructions from on-chip memory and executes a C program compiled for the RV32I instruction set.
- **SRAM interface:** A custom SRAM controller in Verilog connects the CPU memory bus to Block RAM. It correctly handles read and write operations, including any necessary pipelining and handshake signals.
- **Data storage and retrieval:** The running program writes data (for example, an image or arbitrary byte stream) into SRAM and later reads it back, verifying the integrity of the memory system.
- **Status indication via LED/LCD:** Memory-mapped registers allow the program to control LEDs (and optionally an LCD) to indicate the current stage of operation, such as "start", "receiving data", "processing", or "finished".
- **Demonstration on FPGA:** By observing the LED patterns and the behaviour of the software running on the RISC-V core, students can confirm that the SRAM interface operates correctly and that the SoC behaves as expected.

This chapter introduces the overall topic, tools, equipment and functional requirements of the project. Subsequent chapters will describe the theoretical background, detailed design, implementation, and experimental results of the "SRAM Interface with RISC-V" system.

2 Background Knowledge

This chapter presents the theoretical background required to understand the design and implementation of the "SRAM Interface with RISC-V" system. It introduces the main building blocks used in the project and explains how their features are applied in practice to the final product.

2.1 System-on-Chip and FPGA Platforms

A **System-on-Chip (SoC)** integrates a processor core, memory and peripherals into a single hardware system. In commercial chips, these blocks are fabricated on a single silicon die. In this project, a similar structure is created using an FPGA:

- The **processor core** is a soft RISC-V CPU (PicoRV32) implemented with logic resources.
- The **SRAM** is implemented using on-chip Block RAM primitives of the FPGA.
- **Peripherals** such as UART and LEDs are custom Verilog modules connected to the same memory bus as the SRAM.

FPGAs are particularly suitable for educational SoC designs because they allow rapid iteration: designers can modify Verilog code, synthesise, and re-program the board without changing any physical hardware.
In practice, the SoC on FPGA behaves like a small microcontroller system, but with a fully customisable CPU–memory–peripheral interface. This makes it ideal for experimenting with memory protocols, bus timing, and hardware/software co-design.

2.2 RISC-V Architecture and PicoRV32 Core

RISC-V is an open Instruction Set Architecture (ISA) based on a load–store, reduced-instruction philosophy. The base integer subset RV32I provides:

- 32-bit registers and 32-bit addresses,
- arithmetic/logic instructions (ADD, SUB, AND, OR, etc.),
- load and store instructions for memory access,
- control-flow instructions (branches and jumps).

The project uses **PicoRV32**, a compact RV32I-compatible core optimised for low resource usage on FPGAs. Instead of exposing a complex bus like AXI, PicoRV32 provides a simple memory interface with signals such as:

- `mem_valid`, `mem_instr`: indicate a valid memory access (data or instruction).
- `mem_addr`, `mem_wdata`, `mem_wstrb`: address and write data.
- `mem_rdata`: data read from memory or peripherals.
- `mem_ready`: handshake signal from memory/peripheral to acknowledge completion.

This interface is straightforward to connect to custom SRAM controllers and memory-mapped peripherals, which is exactly what the project requires. In practice, the firmware written in C does not see these signals directly; it simply performs loads and stores to addresses, and the hardware interface handles the timing and handshake.

2.3 SRAM and FPGA Block RAM

Static RAM (SRAM) is a type of memory that stores data as long as power is applied, without the need for refresh cycles (unlike DRAM). In discrete systems, SRAM chips connect to a CPU via address, data and control lines. On an FPGA, user SRAM is typically implemented using **Block RAM (BRAM)** primitives:

- BRAM provides synchronous read and write operations with fixed latency (usually one or more clock cycles).
- Addresses and data are registered to meet timing constraints at high clock frequencies.

In this project, the **SRAM controller**:

- maps a region of the RISC-V address space (e.g. starting at `0x0001_0000`) onto Block RAM,
- uses a small pipeline so that requests from PicoRV32 are correctly aligned with BRAM latency,
- generates the `mem_ready` signal to indicate to the CPU when a read or write has completed.

Practically, this allows the RISC-V program to treat the SRAM as a normal data memory region to store an image or byte stream received from the PC.

2.4 Memory-Mapped I/O and Handshake Mechanisms

Instead of using separate instruction sets for I/O, many processors use **Memory-Mapped I/O (MMIO)**: peripherals appear as addresses in the same address space as RAM. A write to a peripheral address becomes a command; a read returns a status or data word.
In the project, MMIO is used as follows:

- Writing to the LED address (e.g. `0x1000_0000`) updates an 8-bit register connected to the board LEDs.
- Writing to `0x1000_0004` sends a byte to the UART transmitter.
- Reading from `0x1000_0008` retrieves a byte from the UART receiver.

The `mem_valid`/`mem_ready` handshake plays a crucial role:

- If a peripheral is ready (e.g. UART TX is idle, UART RX has a byte available), it responds by asserting `mem_ready` quickly.
- If the peripheral is not ready (e.g. UART TX is still sending the previous byte), it keeps `mem_ready` low. PicoRV32 automatically stalls on that instruction until `mem_ready` goes high.

This mechanism implements **hardware-assisted flow control**. In practice, it means the C code can call simple functions like `uart_putc()` and `uart_getc()` without manually polling busy flags; the hardware ensures the CPU only progresses when the operation is safe.

2.5 UART Serial Communication

UART (Universal Asynchronous Receiver/Transmitter) is a widely used interface for serial communication between a microcontroller/FPGA and a PC. It sends and receives frames consisting of:

- one start bit (logic 0),
- a fixed number of data bits (typically 8),
- an optional parity bit,
- one or more stop bits (logic 1).

The project uses a typical configuration of 8 data bits, no parity, and 1 stop bit at 115200 baud. Two custom Verilog modules are used:

UART RX: synchronises the asynchronous `rx` line to the system clock, detects the start bit, samples each data bit at the middle of its period, and assembles the received byte. In the project, a double-buffer scheme is used so that a new byte can be received even while the CPU is still reading the previous one.

UART TX: generates the serial waveform from a byte written by the CPU. A `busy` flag indicates that a frame is in progress, and this flag feeds back into the MMIO logic to control `mem_ready`.

In practical use within the product:

- The PC runs a Python script that opens the serial port and streams a binary file (e.g. an image) to the FPGA.
- The FPGA's UART RX converts the serial bitstream into bytes that the RISC-V CPU stores into SRAM.
- Later, the RISC-V firmware reads bytes back from SRAM and sends them out via UART TX, and the PC script receives and saves the data to a file.

2.6 Practical Features Applied in the Product

Several theoretical concepts from above are directly reflected in the features of the final product:

- **SRAM buffering of large data:** The use of Block RAM as SRAM allows the system to store a relatively large data block (e.g. 96 kB) entirely on-chip. This demonstrates how external data (image, audio, etc.) can be buffered and processed locally by the CPU.
- **Flow control without software polling:** By using `mem_ready` to stall the CPU when UART is busy, the design shows a clean hardware solution to match CPU speed to slow peripherals, simplifying the firmware.
- **Reliable data streaming:** The combination of double buffering in UART RX, pipelined SRAM access, and MMIO handshaking enables reliable end-to-end streaming of data between PC and FPGA with no lost or corrupted bytes.

- **Visual feedback via LEDs (and optionally LCD):** The LEDs are driven by memory-mapped registers to indicate stages such as “start”, “receiving”, and “finished”. This illustrates a common embedded design pattern where software reports internal states through simple output devices.
- **Hardware/software co-design:** The project integrates custom hardware (SRAM controller, UART, MMIO) with software (RISC-V C program and PC-side Python script), providing a realistic example of how low-level digital design and high-level programming work together in an SoC.

This background sets the stage for the following chapters, where the concrete architecture, implementation details, and experimental results of the “SRAM Interface with RISC-V” system are presented.

3 Design

This chapter presents the hardware architecture of the "SRAM Interface with RISC-V" system. It includes the block diagram of the SoC, descriptions of each functional block, and the internal operation of the memory and I/O subsystems.

3.1 System Block Diagram

The complete SoC is implemented on an FPGA and consists of a RISC-V CPU, a custom SRAM controller, UART communication modules, and simple Memory-Mapped I/O (MMIO) peripherals such as LEDs. The high-level architecture is shown in Figure 3.1.

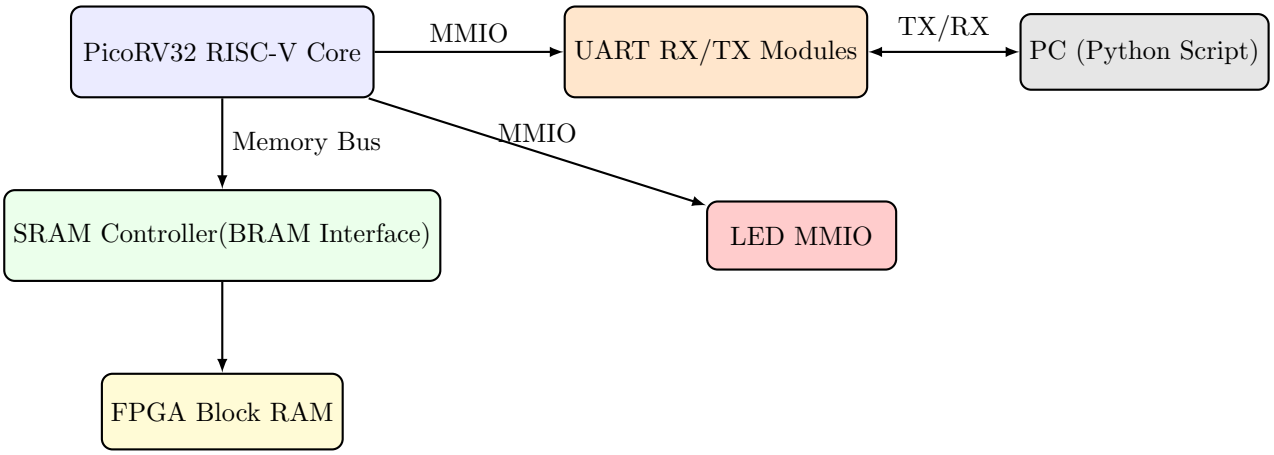

Figure 3.1: High-level block diagram of the RISC-V SoC.

3.2 Functional Blocks

The system is composed of the following major hardware modules:

- PicoRV32 RISC-V Core
- SRAM Controller (interfacing with FPGA Block RAM)
- UART Receiver and Transmitter
- LED Memory-Mapped Register
- Top-level Interconnect and Address Decoder

Each block and its internal functionality are described in detail below.

3.3 PicoRV32 RISC-V Processor

The PicoRV32 core serves as the central processing unit of the system. Its main features include:

- Implements the RV32I instruction set (32-bit integer operations).
- Compact, low-resource design ideal for FPGA-based SoC systems.

- Exposes a **simple memory interface** instead of a complex bus:
  - `mem_valid`, `mem_instr` — indicate instruction or data access.
  - `mem_addr` — 32-bit byte address.
  - `mem_wdata`, `mem_wstrb` — write data and byte enables.
  - `mem_rdata` — returned read data.
  - `mem_ready` — handshake signal from memory/peripherals.

This lightweight memory interface makes it easy to integrate the CPU with custom-designed memories and peripherals.

3.4 SRAM Controller

The SRAM controller bridges the RISC-V memory bus and the FPGA's Block RAM. Since Block RAM has a synchronous read latency, a **two-stage pipeline** is used:

- **Stage 1:** Latch incoming memory request (address, write data, write strobes).
- **Stage 2:** Perform BRAM read/write and return valid output data.

The controller asserts `mem_ready` in alignment with the BRAM response cycle. It supports:

- Byte, half-word, and word writes through byte-enable signals.
- Continuous read/write of large data blocks (e.g., 96 kB image file).
- Mapping of CPU address space, such as:
  - `0x0000_0000` — Instruction memory
  - `0x0001_0000` — SRAM data memory

Functionally, the controller acts exactly as static RAM from the CPU's perspective.

3.5 UART Receiver (uart_rx)

The UART RX module receives serial data from the PC. It includes:

- **Metastability protection** using a double-flop synchronizer.
- **Bit sampling logic** that detects start bit and samples incoming bits at the middle of the bit period.
- **Double-buffering mechanism** that allows:
  - One byte to be consumed by the CPU,
  - While the next byte is being received.
- A `rx_ready` flag that integrates with PicoRV32's `mem_ready` for automatic CPU stalling.

This ensures no data is lost even when the PC transmits bytes continuously.

3.6 UART Transmitter (uart_tx)

The UART TX module handles outgoing serial data. Its functions include:

- Create UART frame:

$$\text{Start bit } (0) + 8\,\text{data bits} + \text{stop bit } (1).$$

- Maintain a `busy` signal during transmission.
- Prevent CPU writes while busy by forcing `mem_ready = 0`, causing the CPU to stall.

This hardware-level flow control allows firmware to use a very simple `uart_putc()` function without checking any flags.

3.7 LED MMIO Peripheral

The LED peripheral is a simple 8-bit register mapped to:

`0x1000_0000`

Writing to this address updates the on-board LEDs. It is used to indicate:

- System startup (0x55)
- Data fully received (0xCC)
- Data fully transmitted (0xAA)

This serves as a visual verification mechanism during FPGA testing.

3.8 Address Decoder and Top-Level Integration

The top-level module `picorv32_top` connects all hardware blocks. Its tasks include:

- Decoding `mem_addr` to determine whether the CPU is accessing:
  - Instruction memory,
  - SRAM data memory,
  - UART TX/RX,
  - LED register.
- Merging ready/data signals from multiple peripherals into the CPU bus.
- Exposing UART TX/RX pins to the FPGA I/O headers.

This module defines the SoC memory map and ensures that only one peripheral responds to each CPU request.

3.9 Overall Functional Description

Putting all modules together, the SoC operates as follows:

1. The RISC-V core fetches and executes instructions from instruction memory.
2. When the program needs to store or load data, it accesses the SRAM controller, which handles timing and BRAM access.
3. When interacting with UART, the program writes/reads MMIO registers, and the hardware stalls the CPU until UART is ready.
4. LEDs provide debugging and status information.
5. The PC streams an image/file to the FPGA; the RISC-V core stores it in SRAM and later sends it back, verifying correctness of the memory and communication system.

This modular hardware design demonstrates how a small RISC-V processor can be integrated with custom-designed memory and peripherals to form a functional FPGA-based SoC.

4 Implementation

This chapter describes how the "SRAM Interface with RISC-V" system is implemented on the FPGA. While Chapter 3 focused on architectural concepts, the present chapter goes into the concrete Verilog modules, signal-level behaviour and the interaction between hardware and firmware. Clear block diagrams, flow charts and timing diagrams are used to illustrate the implementation.

4.1 Implementation Overview

The complete design follows a modular, top–down structure:

- The **PicoRV32** RISC-V processor core executes firmware compiled from C.
- An **SRAM controller** maps the CPU *memory* region to FPGA Block RAM using a pipelined interface.
- **UART RX/TX** modules provide serial communication with a PC.
- A simple **LED peripheral** exposes an 8-bit status register.
- A **top-level address decoder** steers the CPU bus to the correct target (SRAM or MMIO).

All digital blocks are written in Verilog, synthesised by the FPGA toolchain, and finally combined into a single bitstream for the development board.

4.2 Top-Level SoC Structure

The top-level module `picorv32_top_refactored.v` is implemented as a thin **Level-0 wrapper**. It keeps the PicoRV32 core unchanged and delegates all address decoding and MMIO glue logic to the `picorv32_mmio_bus` module (**Level-1**). Figure 4.1 shows an implementation-level block diagram with emphasis on the bus and address decoding.

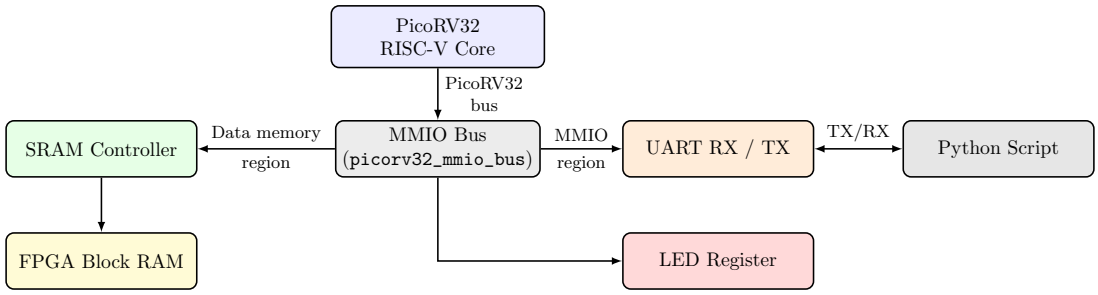

Figure 4.1: Implementation-level block diagram of the refactored PicoRV32-based SoC.

In the refactored structure, `picorv32_top_refactored` mainly instantiates: (i) the PicoRV32 core, and (ii) the `picorv32_mmio_bus` subsystem. All core memory interface signals (`mem_valid`, `mem_instr`, `mem_addr`, `mem_wdata`, `mem_wstrb`, `mem_rdata`, `mem_ready`) are wired directly from the PicoRV32 core into `picorv32_mmio_bus`. The bus module performs address decoding and ensures that only one target responds per transaction, i.e., only one block asserts `mem_ready` and drives `mem_rdata` at a time.

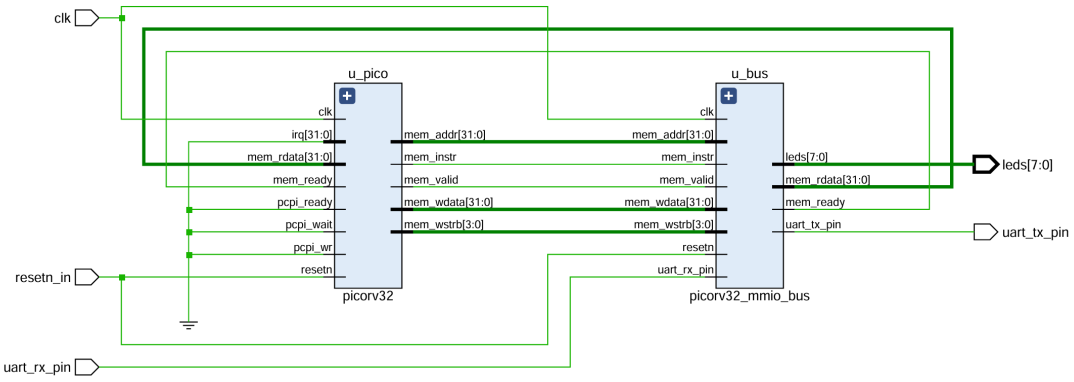

Figure 4.2: Post-synthesis RTL schematic of `picorv32_top_refactored` (Level-0 wrapper) instantiating PicoRV32 and `picorv32_mmio_bus` (Level-1).

Figure 4.2 confirms the structural integration after refactoring: the wrapper exposes only the external I/O (`clk`, `resetn_in`, `uart_rx_pin`, `uart_tx_pin`, and `leds`), while all memory and MMIO routing logic is encapsulated inside `picorv32_mmio_bus`.

4.3 MMIO Bus and Address Decoding

Address decoding and MMIO routing are implemented inside the `picorv32_mmio_bus.v` module. Based on the value of `mem_addr`, the bus selects one of the following targets: SRAM controller, UART RX, UART TX, or LED register.

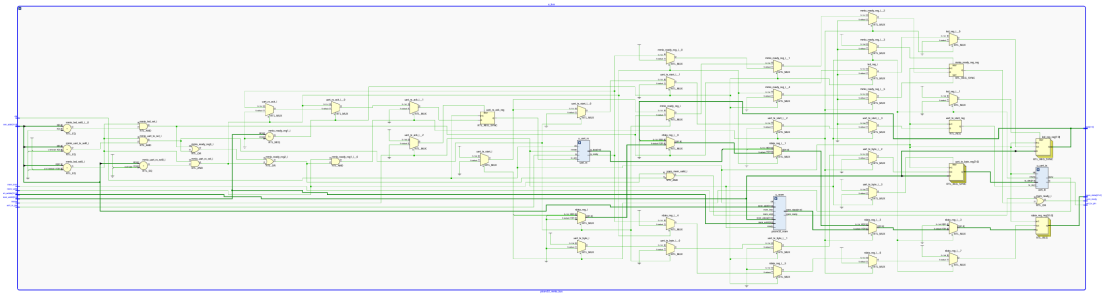

Figure 4.3: Internal RTL schematic of the `picorv32_mmio_bus` module, showing address decoding, data multiplexing, and `mem_ready` generation for SRAM and MMIO peripherals.

As illustrated in Figure 4.3, multiple multiplexers and logic gates are used to ensure that only one peripheral drives `mem_rdata` and asserts `mem_ready` at a time. This hardware-based arbitration guarantees correct bus behaviour and enables safe CPU stalling when peripherals are not ready.

4.4 SRAM Controller Implementation

The SRAM controller (`picorv32_sram.v`) converts PicoRV32 bus cycles into synchronous Block RAM operations. Because a Block RAM read cannot return data in the same cycle as the address is presented, the controller uses a **two-stage pipeline**:

1. **Stage 1 (S1): Request capture:** When `mem_valid` is asserted and the address falls inside the SRAM window, the controller latches the address, write-data and byte strobes. At this cycle, `mem_ready` remains low.
2. **Stage 2 (S2): BRAM access and response:** The latched address is applied to the Block RAM. For reads, the word at that address becomes available at the end of this cycle and is returned on `mem_rdata`. For writes, the data is committed to memory. At this point the controller asserts `mem_ready` for one cycle so that PicoRV32 can retire the instruction.

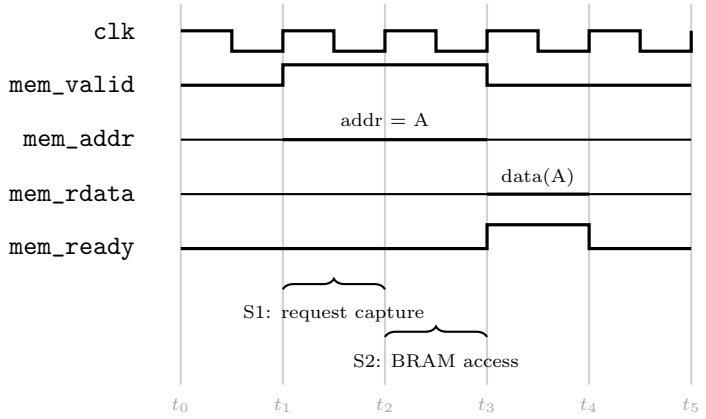

Figure 4.4: Timing diagram of a PicoRV32 read transaction through the two-stage pipelined SRAM controller.

From the CPU point of view, the SRAM behaves as a normal memory with a two-cycle latency: the instruction accessing the data simply stalls until `mem_ready` becomes high.

4.4.1 Read vs. Write Timing Comparison

Although both load (read) and store (write) operations go through the same two-stage SRAM controller pipeline, their observable timing differs mainly in the presence of return data:

4.4.1.1 Read cycle (LOAD). For a read, the CPU asserts `mem_valid` with `mem_wstrb=0`. During **S1**, the controller latches the address and keeps `mem_ready=0`, so the CPU stalls. During **S2**, the BRAM outputs the requested word at the end of the cycle; the controller drives `mem_rdata` and asserts `mem_ready` for one cycle. The CPU retires the load instruction when `mem_ready` is high.

4.4.1.2 Write cycle (STORE). For a write, the CPU asserts `mem_valid` with non-zero `mem_wstrb` to indicate which bytes are valid. In **S1**, the controller latches `mem_addr`, `mem_wdata`, and `mem_wstrb`. In **S2**, the data is written into BRAM using the byte enables. The controller

then asserts `mem_ready` for one cycle to complete the store. In contrast to reads, `mem_rdata` is not used and may remain don't-care.

4.4.1.3 Key difference. A read requires BRAM output data to become valid (one-cycle latency), therefore `mem_rdata` is meaningful only in S2. A write commits data in S2 using `mem_wstrb` byte enables, and does not require returning data to the CPU.

4.5 UART Receiver Implementation

The UART receiver (`uart_rx.v`) converts the asynchronous `rx` line from the PC into bytes readable by the CPU.

4.5.0.1 Key implementation aspects.

- The `rx` input is first passed through a two-flip-flop synchroniser to reduce metastability risk.
- A baud-rate counter divides the system clock (e.g. 125 MHz) down to the UART bit period at 115200 baud.
- When a falling edge (start bit) is detected, the module waits 1.5 bit periods for the first sample, then samples each next bit in the middle of its interval.
- Two internal buffers (`buf0`, `buf1`) store received bytes so that a new byte can be captured while the CPU is still reading the previous one.
- A flag `rx_ready` indicates that at least one byte is waiting; when the CPU reads from the UART RX MMIO address, one buffered byte is returned and the internal FIFO is shifted.

If the firmware attempts to read from UART RX while no byte is ready, the top-level logic keeps `mem_ready` low, effectively stalling the CPU until a complete frame has been received.

4.6 UART Transmitter Implementation

The UART transmitter (`uart_tx.v`) sends bytes from the CPU to the PC. It uses a 10-bit shift register containing:

$$\underbrace{0}_{\text{start}} \quad d_0, \ldots, d_7 \quad \underbrace{1}_{\text{stop}}$$

Once the CPU writes a byte to the UART TX MMIO address, the transmitter:

1. Loads the shift register with start bit, data bits and stop bit.
2. Asserts a `busy` flag for the whole frame duration.
3. On each baud tick, outputs the least significant bit and shifts.
4. De-asserts `busy` after ten bit periods.

4.6.1 UART Timing (10-bit Frame)

UART uses an asynchronous serial frame format **8N1**, meaning: **1 start bit**, **8 data bits** (LSB first), **no parity**, and **1 stop bit**. In the idle state, the UART line stays at logic '1'. A transmission begins when the line falls to '0' to indicate the start bit.
Figure 4.5 shows a complete **10-bit UART frame**. The receiver samples near the **middle of each bit interval** to maximise noise margin.

- **Start bit:** line goes low ('0') for one bit period T_{bit}.
- **Data bits:** 8 bits are transmitted **LSB first**, each held for T_{bit}.
- **Stop bit:** line returns high ('1') for at least one bit period.

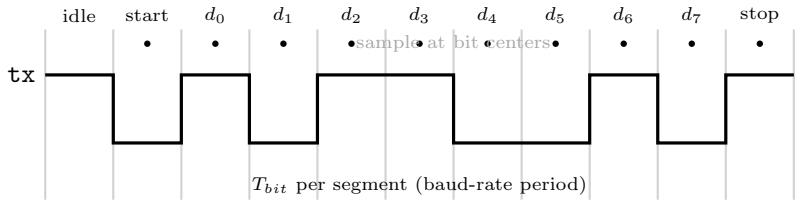

Figure 4.5: UART timing for a full 10-bit frame (8N1): start bit, 8 data bits (LSB first), and stop bit.

At the top level, a CPU write to the UART TX MMIO address is only acknowledged (`mem_ready=1`) when the transmitter is not busy. While `busy` is high, the top-level logic keeps `mem_ready=0`, stalling the CPU on the store instruction. Similarly, UART RX reads stall when no received byte is available.

4.7 LED MMIO Peripheral

The LED peripheral is implemented as a simple 8-bit register mapped at address `0x10000000`. It has one write-only port from the CPU bus and eight output bits connected directly to the FPGA user LEDs.
The firmware uses three distinctive values:

- `0x55` – system start.
- `0xCC` – data reception completed.
- `0xAA` – data transmission completed.

These patterns allow quick visual verification of system progress.

4.8 Firmware Implementation and Processing Flow

The firmware written in C is compiled with the RISC-V GCC toolchain to an ELF and binary image. A Python script (`make_mem.py`) converts the binary into a `.mem` file suitable for initialising instruction memory in Verilog using `$readmemh`.

Firmware pseudocode

```
BEGIN
    WRITE_LED(0x55)              // Startup indicator

    // Receive N bytes from PC and store into SRAM
    FOR i = 0 TO IMAGE_SIZE - 1 DO
        b := UART_GETC()         // blocks until byte is received
        SRAM[IMAGE_BASE + i] := b
    END FOR

    WRITE_LED(0xCC)              // Reception complete

    // Read SRAM and transmit back to PC
    FOR i = 0 TO IMAGE_SIZE - 1 DO
        b := SRAM[IMAGE_BASE + i]
        UART_PUTC(b)             // blocks while UART TX is busy
    END FOR

    WRITE_LED(0xAA)              // Transmission complete
END
```

Figure 4.6: Pseudocode of the RISC-V firmware.

Processing flow chart

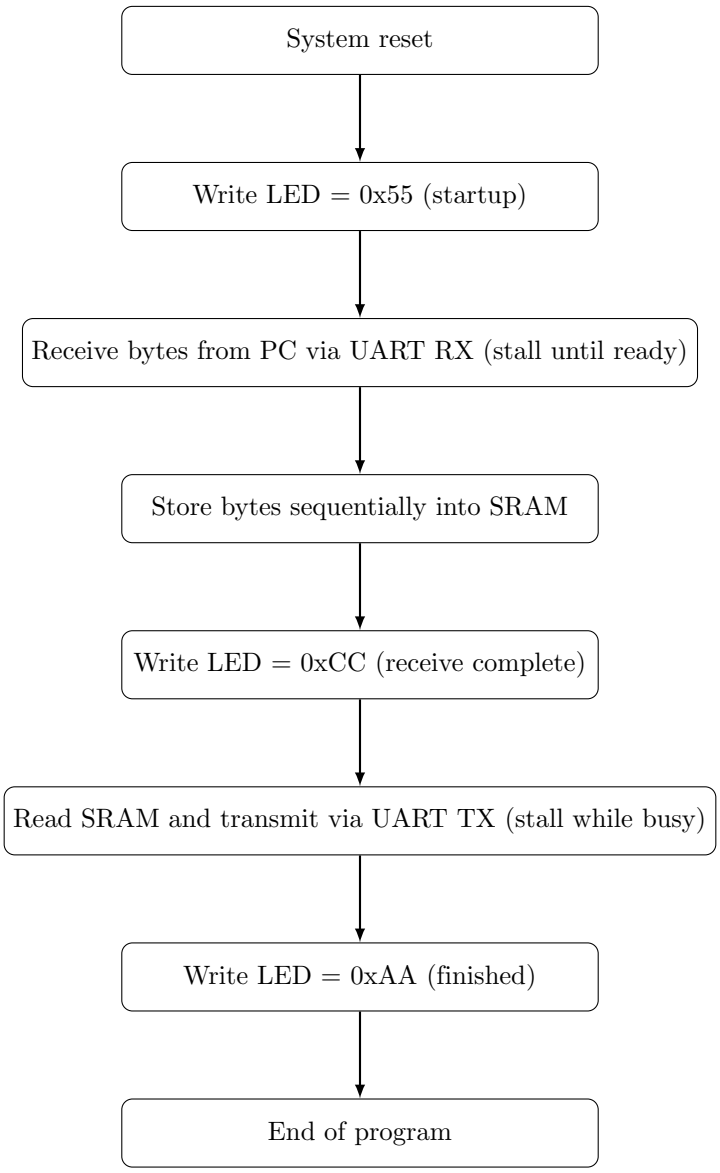

Figure 4.7: Processing flow of the firmware running on PicoRV32.

4.9 FPGA Resource Utilisation

After synthesis and implementation, the FPGA tool reports the resources consumed by each part of the design. Table 1 provides a template for reporting LUT, FF and BRAM usage. The address decoding logic and MMIO routing are not implemented as separate modules. Instead, they are integrated directly into the top-level module (`picorv32_top.v`), where the PicoRV32 bus is decoded and steered toward either the SRAM controller or the corresponding MMIO peripherals.

Table 1: FPGA resource utilisation after synthesis.

Module	LUTs	FFs	BRAM
PicoRV32 CPU	927	574	0
SRAM controller	42	285	0
UART RX	56	50	0
UART TX	23	25	0
Address decode + MMIO	included	included	0
Instruction/Data memory	0	0	64
Total	1048	985	64

4.10 Experimental Results

The design was validated on hardware through end-to-end UART transfer.

End-to-end file loopback

A file (e.g. 96 kB) is transmitted from the PC to the FPGA via UART. The firmware stores it in SRAM and then sends the same bytes back to the PC. A byte-wise comparison between the original file and the returned file should match exactly, confirming correct operation of: (i) UART RX/TX framing, (ii) SRAM controller pipeline, (iii) MMIO stalling logic using `mem_ready`.

LED status observation

The LED patterns provide visual confirmation of program progress: `0x55` at startup, `0xCC` after receiving the full file, and `0xAA` after transmitting the file back.

4.11 Summary

This chapter detailed the FPGA implementation of the PicoRV32-based SoC, including the pipelined SRAM controller, UART RX/TX peripherals, and the firmware workflow for UART-to-SRAM loopback. The timing diagrams demonstrate how `mem_ready` provides hardware-assisted flow control, allowing the CPU to stall safely without complex firmware polling or interrupt logic.

5 Conclusion and Future Work

5.1 Conclusion

This project has successfully implemented a complete **SRAM Interface with a RISC-V processor** on FPGA, demonstrating a practical System-on-Chip (SoC) design that integrates computation, memory, and communication peripherals.
A lightweight **PicoRV32 RISC-V core** was selected as the CPU, providing a simple yet effective bus interface suitable for educational and experimental purposes. A custom **SRAM controller** was designed to interface the CPU with FPGA Block RAM. Due to the synchronous nature of Block RAM, a **two-stage pipeline** mechanism was introduced, allowing correct handling of both read and write operations while maintaining compatibility with the PicoRV32 bus protocol.
UART communication was implemented using dedicated **UART RX and UART TX modules**. These modules handle asynchronous serial data transfers with correct timing, framing, and buffering. Importantly, the system employs a **hardware-assisted flow control mechanism** based on the `mem_ready` signal. This approach allows the CPU to stall automatically when peripherals are not ready, eliminating the need for polling loops or interrupt handling in firmware.
The firmware running on the RISC-V core demonstrates an end-to-end data path: receiving data from a PC via UART, storing it in SRAM, and transmitting it back without loss or corruption. The correct operation of the system is verified through:

- successful loopback of large data blocks (tens of kilobytes),
- timing-accurate UART communication,
- correct SRAM read/write behaviour,
- clear visual status indication using on-board LEDs.

Overall, the project validates the feasibility of building a compact, modular RISC-V SoC on FPGA and highlights how careful hardware design can significantly simplify firmware logic.

5.2 Key Achievements

The main achievements of this project can be summarised as follows:

- Design and integration of a PicoRV32-based RISC-V system on FPGA.
- Implementation of a pipelined SRAM controller compatible with synchronous Block RAM.
- Correct handling of memory-mapped I/O (MMIO) for UART and LED peripherals.
- Demonstration of hardware-based stalling using `mem_ready` for both memory and peripheral accesses.
- Successful end-to-end experimental validation on real FPGA hardware.

These results show that even a minimal RISC-V core can be effectively used to build a functional embedded system when combined with well- designed peripherals.

5.3 Limitations

Although the system meets the project requirements, several limitations remain:

- The design does not include caches; all memory accesses go directly to Block RAM, which limits performance.
- UART communication speed is constrained by the chosen baud rate and remains much slower than the system clock.
- The system relies on polling-style blocking via `mem_ready` rather than interrupts, which may not scale well for more complex multitasking systems.
- Error detection mechanisms such as UART parity or CRC are not implemented.

These limitations are acceptable for a learning-oriented design but would need to be addressed in a production-level system.

5.4 Future Work

Several extensions can be considered to further improve and expand the system:

- **Interrupt support:** Adding interrupt handling would allow more efficient CPU utilisation and support concurrent tasks.
- **Cache memory:** Introducing instruction and/or data caches could significantly improve performance.
- **DMA engine:** A direct memory access controller could offload large data transfers from the CPU.
- **Additional peripherals:** Timers, GPIO, SPI, or I^2C interfaces could be added to build a more complete SoC.
- **Higher-speed interfaces:** Replacing UART with faster interfaces (e.g. USB or Ethernet) would enable more demanding applications.

With these improvements, the presented architecture could serve as a foundation for more advanced RISC-V based embedded systems and research projects.

5.5 Final Remarks

In conclusion, this project provides a clear and practical example of how a RISC-V processor can be integrated with custom memory and I/O interfaces on FPGA. The combination of simple hardware design, cycle-accurate control, and clean firmware structure makes the system easy to understand, verify, and extend. This work demonstrates the strong potential of open-source RISC-V cores for both education and real-world embedded system development.