AGSTU **Examensarbete/Thesis**

Author Freddy Avaria		Assignement FPGA & TEIS Thesis	
Email address freddy_avaria@hotmail.com	Controlled by	Version 1.0	File Freddy_Avaria_FPGA_&_TEIS_Thesis_B. docx

AGSTU VOCATIONAL UNIVERSITY

PXE-Based Remote Reboot for Speed Enforcement System

FPGA & TEIS (Applied Electronics in Embedded Systems) Thesis - part B

Freddy Avaria

2024-11-23

ABSTRACT

This thesis recreates the basic steps of a PXE-style network boot inside an FPGA and adds a hardware accelerator that can observe and analyze the process in real time. A simple TFTP client and server were built in VHDL and connected internally so they can exchange boot messages without using any external network hardware. The accelerator watches this traffic as it flows through the system and identifies the key parts of each message, making it easier to understand how the protocol works and how it could be sped up in hardware. The design was tested both in simulation and on the FPGA board, resulting in a clear learning platform for network booting and for developing future FPGA-based acceleration techniques.

Content

1 REQUIREMENTS

Table 1: Requirement list

Requirement	Description	Done
Pre-study/initial report (also to be included in the final report)		
1	**Initial report** The deliverable must be in Word format and written in either Swedish or English. The report should include the following: 1. **Summary:** A brief overview of the project and its objectives. 2. **Project goals:** A clear description of what the project is intended to deliver. 3. **Specification:** A preliminary specification describing the system or IP component to be delivered (may be adjusted as needed). 4. **Component reuse:** Identify components that may be reused, including references. Also, review previous thesis projects as supporting material. 5. **Architecture:** Outline a proposed hardware (HW) and/or software (SW) architecture. 6. **Verification and Validation:** Describe how the project will be verified and validated. The methods can be revised during the project. 7. **Activity and Time Plan:** Create a timeline for the project in accordance with the course syllabus. Follow up on the plan and present the results in the final report, including a calculation of labor costs (800 SEK/hour). 8. **Role Distribution in Group Work:** If the project is conducted in a group, specify which part of the work you are responsible for. All group members must submit full copies of the project. 9. **Chapter Division and Structure:** Propose a chapter outline for the final report and a folder structure for the project delivery (e.g., source code, documentation, and test results). See Final Delivery, item 3. 10. **Scope:** Maximum six pages. **Note: If the project is more theoretical or focused on another type of embedded system, the initial report may differ.**	**OK**
2	The initial report must be a standard report in Word format. The deliverable must be submitted to Lennart (lennart.lindh@agstu.com).	**OK**

	The filename should be “firstname_lastname_thesis_A.zip”. Use "Examensarbete" as the subject line.	
End of pre-study (must be approved before proceeding to the next phase)		

Table 2: Requirement list

Requiremet	Description	Done
	Documentation requirements and folder structure	
1	Leverans med strukturerad standardrapport. 1. Report (**NOTE:** file name: firstname_lastname_thesis_B) a. Delivery in **Word format**, swedish or english. b. Maximum 30 pages	**OK**
2	**Presentation of results:** To present the thesis work, a video of maximum 10 minutes should be created and published on Youtube. The video should contain the following: 1. **Introduction:** Present yourself and your role on the project. 2. **Overview:** Give a summary of the goal and purpose of the project. 3. **Architecture:** Describe the system’s architecture and main components. 4. **Demo:** Show a practical demonstration of the system in function. 5. **Conclusion:** Feel free to share the project’s conclusions, including what worked well, any challenges encountered and possible areas of improvement. 6. **Tag:** Please add the tag **"AGSTU"** to the video. This will make it easier for others to find all videos from the program. To view previous projects, search for the tag **"TEIS"**. **Important note:** If you do not want the video to be public, clearly indicate this along with the link to the video.	**OK**
	Delivery requirements	
3	Final delivery may include the following folders: • **Documentation folder** • **Design materials folder** • **Demo examples folder** • **Testbenches folder** • **Miscellaneous folder**	**OK**

	If the project is more theoretical or focused on another type of embedded system, an alternative folder may be defined. The delivery should be made via the Itslearning platform. The file name should be ”firstname_lastname_thesis.zip” (single file).	
4	The student must demonstrate the ability to report the status of the thesis work each week during the status meeting.	**OK**
lG/G/VG	The grade is based on an overall assessment made by those who evaluate the different stages of the project.	

2 INTRODUCTION

PXE allows a device to start up using information sent over a network instead of from local storage. This thesis explores how the basic steps of that process can be recreated inside an FPGA. By building a simple TFTP client and server that communicates internally, the project makes it easy to observe how the boot messages are exchanged and how the protocol works in practice. Alongside, a hardware accelerator was designed to watch and analyze this traffic in real time. By identifying key parts of each message as they pass through the system, the accelerator helps show how the protocol works and how it might be improved or sped up in hardware. The result is a clear, controlled platform for learning about network booting and for future development of FPGA-based boot systems.

3 BACKGROUND

3.1 PXE – Origins and Evolution

3.1.1 History

PXE, or Preboot Execution Environment, was introduced by Intel in the late 1990s as part of its Wired for Management initiative. Its purpose was to let computers boot directly from a network rather than from a local drive, making it easier to deploy operating systems and manage diskless workstations. In a PXE boot, the network interface card's firmware uses DHCP to get an IP address and the location of a boot server, then downloads a bootloader via TFTP before the operating system starts. Through the 2000s, PXE became a standard feature in most network cards and BIOS firmware, widely used in enterprises for centralized OS installation, recovery, and thin client setups. With the shift to UEFI firmware, PXE evolved to support newer protocols like HTTP Boot for faster and more secure network starts, but its core role—network-based booting and deployment—remains the same today.

A basic PXE setup can be seen in the following figure.

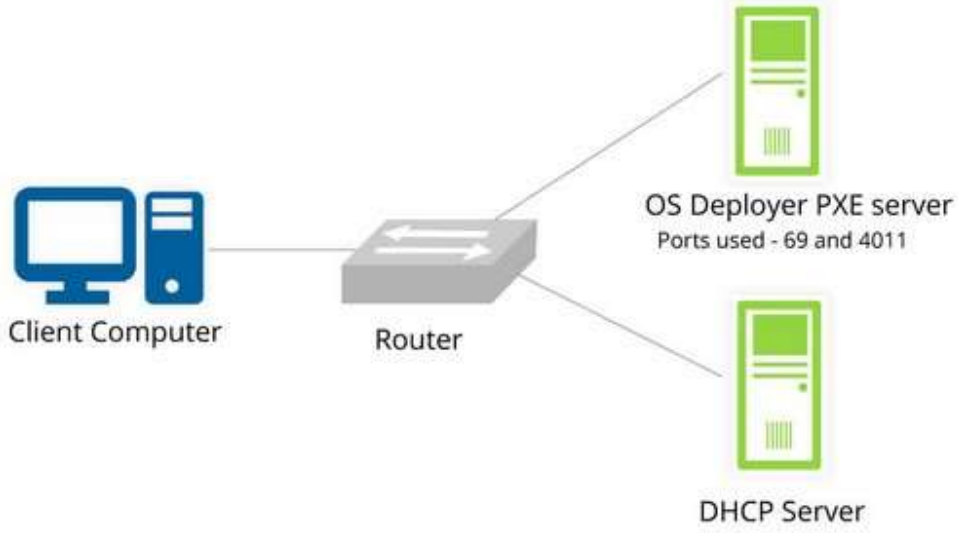


Figure 1: PXE server setup

3.2 Networking Terminology

This section introduces the fundamental networking layers and protocols relevant to the PXE-style boot architecture implemented in this project. The design emulates a minimal network stack—Ethernet → IPv4 → UDP → TFTP—entirely inside the FPGA fabric. Understanding the responsibilities of each protocol layer is essential for interpreting how packets are constructed, parsed, and transported through the hardware modules.

3.2.1 Ethernet

Ethernet is a Layer-2 (Data Link Layer) protocol defining how frames are formatted and physically transmitted across a wired network. Although the DE10-Lite board does not include a physical Ethernet PHY, the FPGA still constructs valid Ethernet frames internally so that each subsystem can operate as if connected to a real network.

The following figure shows one of the most common Ethernet type frames:

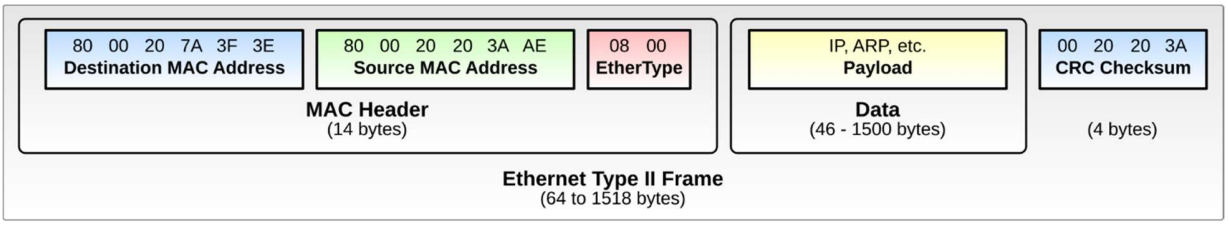


Figure 2: Ethernet type 2 frame

Key properties used in this project

- MAC addressing (48-bit source and destination).
- EtherType field identifying the encapsulated protocol (IPv4 = 0x0800).
- Fixed 14-byte header, followed by payload.
- No CRC generation in this design (unused due to lack of a physical link).

Role in the system

Ethernet framing ensures that higher-layer protocol emulation remains realistic. Both the TFTP Mini-Server and the Accelerator parse Ethernet headers to maintain correct offset alignment.

3.2.2 IP

IPv4 (Internet Protocol ver.4) is a Layer-3 protocol responsible for packet addressing and routing. In the FPGA design, IPv4 packets are constructed and parsed, including proper header fields and checksum generation.

The following figure shows how an IPv4 header is formed:

Figure 3: IPv4 header

Key IPv4 concepts used

- 32-bit source/destination IP addresses.
- Protocol field (UDP = 0x11).
- Total length field, indicating full packet size.
- Header checksum, represented using constant predefined values in this revision (implemented in VHDL).
- 20-byte header without options.

Role in the system

The TFTP Mini-Server constructs valid IPv4 headers for each DATA response, and the Accelerator RX Parser decodes IPv4 headers to classify packet types and extract metadata. In this revision, no validation of checksum integrity is performed.

3.2.3 UDP

UDP (User Datagram Protocol) is a Layer-4 transport protocol designed for low-overhead, connectionless communication. TFTP uses UDP because it is lightweight and does not require session establishment.

The following figure shows a typical UDP header:

Figure 4: UDP header

Key UDP characteristics used

- Source and destination ports (client ephemeral port ↔ server port 69).
- 8-byte UDP header.
- Length field covering header + payload.
- Checksum, which is legally allowed to be zero in IPv4 (used in this design).

Role in the system

Both transmit and receive logic encode/decode:

- UDP ports (to match client/server roles)
- Length of TFTP payload
- Protocol identification during parsing

3.2.4 TFTP

TFTP (Trivial File Transfer Protocol) is a simple file transfer protocol commonly used by PXE boot environments. It operates on top of UDP and defines a minimal set of message types, each with a well-defined binary format.

The following figure shows a typical TFTP Protocol Transfer:

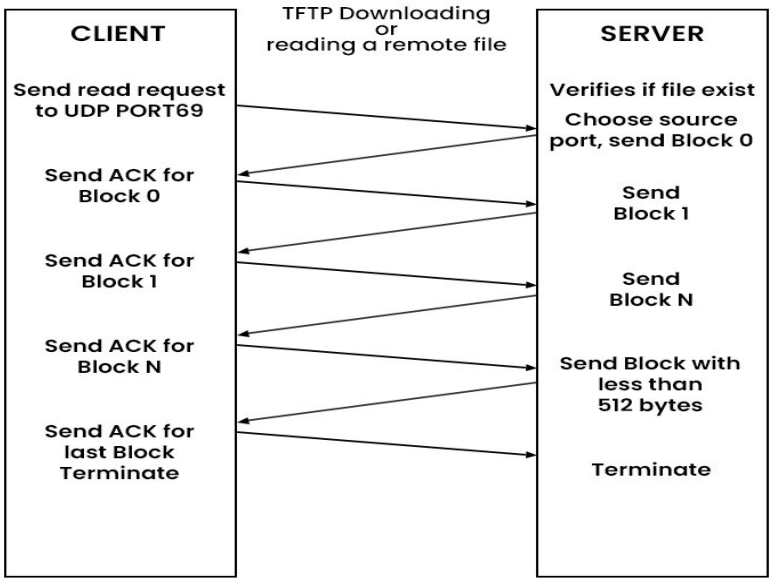


Figure 5: TFTP transferring

TFTP message types used

- RRQ (Read Request) – client asks for a file.

- DATA – server sends a block of the file.
- ACK – client acknowledges receipt of a block.

Key protocol properties

- Block-oriented: data is sent in fixed-size blocks (512 bytes here).
- Stop-and-wait flow control: only one block in flight at a time.
- Transfer ends when the final DATA packet has a payload < 512 bytes.
- Port negotiation: server responds from a dynamically chosen port.

Role in the system

The FPGA's Mini-Server implements:

- DATA block generation.
- ACK monitoring
- Session tracking (active, block number, bytes remaining)

The min-server transmits DATA packets but does not process RRQ in this revision.

4 PROJECT GOALS

This project aims to design and validate a lightweight, PXE-style TFTP exchange implemented entirely in FPGA hardware. Both the TFTP client and the simplified TFTP mini-server are written in VHDL and run on an Intel MAX 10 (DE10-Lite) FPGA, communicating through an internal loopback rather than a physical Ethernet interface. This allows the system to generate realistic Ethernet/IPv4/UDP/TFTP frames and evaluate their behavior entirely on-chip.

The primary goal is to create a self-contained validation platform that models the timing, control flow, and protocol-header structure of a PXE/TFTP transaction. The system implements the essential parts of the exchange—RRQ, DATA, and ACK—using correct protocol headers and deterministic FSM sequencing. Although the design does not yet transmit actual file payloads, it produces a repeatable and fully observable traffic pattern suitable for simulation, SignalTap debugging, and studies of hardware-accelerated protocol handling.

A secondary goal is to ensure architectural modularity and future expandability. The streaming interfaces, loopback routing, and ROM infrastructure are designed so that payload support, full session logic, or an external MAC/PHY can later be added with minimal changes. This prepares the system for eventual integration with real PXE servers such as Serva.

By implementing the client, mini-server, and accelerator entirely in hardware, the project demonstrates how network-protocol behavior can be executed without a CPU, using deterministic FSMs and cycle-accurate timing. The platform serves as a clean foundation for further work in FPGA-based network offloading, embedded boot mechanisms, and hardware-driven protocol modeling.

5 SCOPE OF THE PROJECT

This chapter defines the boundaries, objectives, and deliverables of the project. The work centers on designing and validating a PXE-style TFTP header-level exchange implemented entirely in hardware on an Intel MAX 10 (DE10-Lite) FPGA. Instead of relying on a full PXE stack or external servers, both the TFTP client and a simplified TFTP mini-server are implemented directly in VHDL and communicate through an internal loopback path. This enables realistic Ethernet/IPv4/UDP/TFTP frames to be generated and analyzed without requiring a physical network interface.

Although the design does not implement full file transfer or payload handling, it accurately reproduces the core PXE/TFTP sequence—RRQ, DATA, and ACK—using synthesizable finite state machines. This provides deterministic timing, cycle-accurate control flow, and full observability of all protocol layers. The platform serves as a practical starting point for later integration with external MAC/PHY hardware or PXE tools such as Serva.

5.1 System Design and Architecture

The system is built around a unified on-chip validation platform where the TFTP client FSM, Mini-Server TX FSM, and Network Accelerator operate together in a shared clock domain. A ROM module is included for future payload support but is not yet used in the active DATA path.

A configurable top-level supports two operating modes:

1) **Validation Mode** – Both client and server are active, exchanging RRQ, DATA, and ACK packets over the internal loopback. This enables complete observation of header generation, FSM transitions, and frame timing.

2) **Accelerator-Only Mode –** The client and server are disabled, and only the accelerator's RX parser operates. This is used to test parsing logic and collect protocol statistics independently.

This architecture creates a scalable and deterministic testbed for studying protocol structure, block sequencing, and hardware-based control flow.

5.2 Network Protocol Emulation and Evaluation

Instead of using live Ethernet traffic, the system emulates packet-level TFTP communication internally. The focus is on functional correctness of headers, predictable FSM behavior, and controlled timing—not on throughput or full file transfer.

Each frame (Ethernet, IPv4, UDP, TFTP) is assembled with valid structural fields and streamed byte-by-byte through a standard tvalid/tready interface. This allows precise observation of:

- block numbers,
- opcode sequences,
- header offsets,
- FSM transitions,
- tvalid / tready handshake timing.

This makes the simplified TFTP model suitable for studying protocol behavior and for evaluating potential hardware acceleration techniques.

5.3 FPGA-Based Hardware Acceleration

The design includes a Network Accelerator module that parses all outgoing frames in real time and maintains protocol-level statistics. It identifies Ethernet, IPv4, UDP, and TFTP headers; extracts opcodes and block numbers; and updates counters for DATA and ACK packets.

While the accelerator does not yet modify or generate packets, it establishes a solid foundation for future offload mechanisms, including:

- parallel header decoding,
- opcode/block number tracking,
- potential automated ACK generation.
- timing or flow-control assistance

The loopback-based architecture ensures that an external MAC/PHY or SPI-to-network bridge can be added later with minimal redesign.

5.4 Validation and Analysis

The system is validated through ModelSim simulation and SignalTap on-device capture. Key evaluation aspects include:

- FSM transition coverage and correctness,
- cycle-accurate header timing,
- latency across the streaming interface (tvalid/tready interaction),
- stability under configuration changes (block count, IP/MAC values).

These validation steps confirm that the hardware behaves predictably and follows the expected TFTP-style flow, making it suitable for future expansion into a full PXE-capable environment.

5.5 Documentation and Deliverables

Project deliverables include:

- Fully commented on VHDL source files for all modules (client, server, accelerator, ROM, and unified top).
- A unified top-level design (na_unified_top.vhd) supporting both validation and accelerator-only modes.
- Simulation waveforms, SignalTap captures, and FSM diagrams demonstrating traffic generation and parsing.
- A technical report describing the architecture, design rationale, and guidelines for future evolution into a full PXE/TFTP engine.

Overall, the project delivers a controlled FPGA platform for protocol exploration, FSM design, and early hardware-accelerated processing. It provides a solid foundation for future PXE-oriented, embedded-network, or remote-update systems built on FPGA technology.

6 SCOPE REDEFINITION AND JUSTIFICATION

The original plan for this project was to build a complete, network-based PXE boot environment using a Raspberry Pi 5 as the PXE client and a 64-bit industrial PC as the TFTP/PXE server. The goal was to perform diskless booting over Ethernet, measure performance, and explore FPGA-based acceleration in a real network.

During early development, it became clear that the Intel MAX 10 (DE10-Lite) board lacks the hardware needed for native Ethernet: no integrated MAC/PHY, no RMII/GMII interfaces, and insufficient I/O to attach a standard PHY without additional expansion hardware. Achieving full PXE capability would require a MAC IP core, external PHY, memory resources, and a complete network stack—well beyond the feasible scope of this project.

To keep the work focused and technically meaningful, the scope was redefined to a self-contained PXE/TFTP transaction model implemented entirely in FPGA logic. The TFTP client and a simplified TFTP mini-server now run on the same device and communicate through an internal streaming loopback. This approach reproduces the essential packet-level behavior of RRQ, DATA, and ACK exchanges without relying on external hardware.

This redefinition brings several advantages:

6.1 Hardware Focus

By removing external networking, the project concentrates on designing and optimizing FPGA-resident FSMs that implement PXE-style control flow, timing, and protocol behavior.

6.2 Deterministic Validation

With all traffic kept internal, every byte, handshake, and state transition can be observed in ModelSim and SignalTap with cycle-accurate precision—visibility that is impossible in a real Ethernet environment.

6.3 Extensibility

The modular streaming interfaces allow the loopback path to be replaced later by an external MAC/PHY or network bridge. The validated client, mini-server, and accelerator logic can eventually interact with real PXE servers such as Serva with minimal structural changes.

In summary, while full PXE deployment remains a long-term direction, the revised scope delivers a technically solid and achievable FPGA-based model of PXE/TFTP header-level behavior. It enables meaningful analysis of protocol control, timing, and hardware acceleration while remaining compatible with future expansion into real network environments.

7 PROJECT DESIGN

This chapter describes the overall design and internal structure of the PXE-style TFTP exchange system implemented on the Intel MAX 10 (DE10-Lite) FPGA. The implementation is written in VHDL-93 and consists of several synthesizable hardware subsystems that emulate the header-level behavior of a simplified PXE/TFTP client–server interaction while also including a hardware accelerator module for protocol-parsing experiments.

Although the DE10-Lite board lacks a physical Ethernet interface, the system reproduces the essential parts of a PXE/TFTP transaction using fully on-chip, stream-based connections. This allows the fundamental control flow—RRQ, DATA, and ACK sequences—to be modeled and observed deterministically without external network hardware. The design therefore serves both as a validation platform for FPGA-based protocol controllers and as a research environment for future network-acceleration architectures.

7.1 Hardware Specification

The implementation was developed and tested on the Intel/Altera DE10-Lite FPGA development board. The board is built around an Intel MAX 10 device, model 10M50DAF484C7G, which provides a compact but capable platform for embedded logic design, on-chip memory, and real-time validation tools such as SignalTap. The figure below shows the board itself:

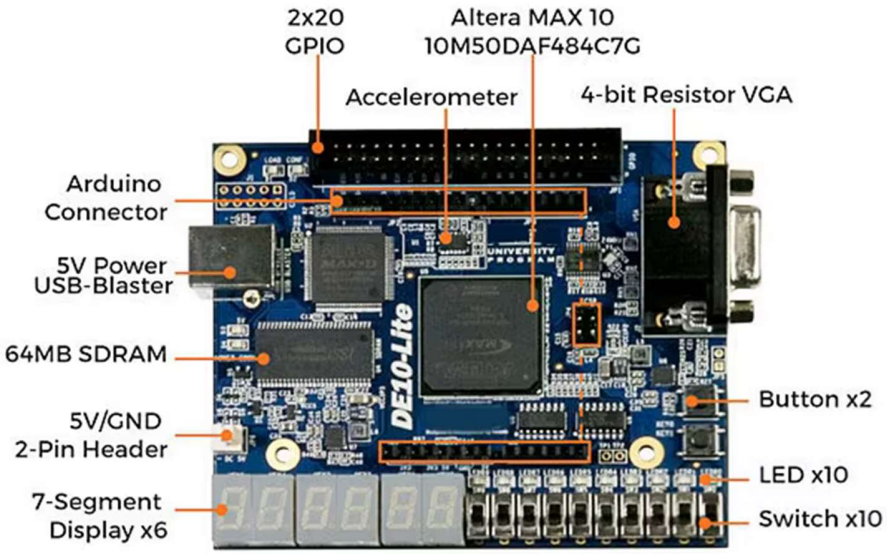


Figure 6: FPGA board DE10-Lite

FPGA Device:

- ***Family:*** Intel MAX 10
- ***Part number:*** 10M50DAF484C7G
- ***Logic elements***: ~50,000
- ***Embedded memory (M9K blocks):*** 1,638 Kbits
- ***Embedded multipliers:*** 144 × 18×18
- ***Package:*** 484-pin FBGA
- ***On-chip flash:*** Yes (dual-image configuration support)
- ***Core voltage:*** 1.2 V

Clock and Timing:

- ***Primary system clock:*** 50 MHz onboard oscillator
- ***PLL support:*** Yes, multiple PLLs for derived clocks
- ***Internal timing:*** Stable operation confirmed at >70 MHz (design target: 50 MHz)

Board Features Used:

- ***User LEDs:*** Used for PASS/FAIL indication.
- ***User Switches:*** SW0 (Mode selection) and SW1 (Force Fail).
- ***On-Board USB Blaster 2:*** For programming, Signal Tap capture and debugging.

Relevant Limitations:

- No built-in Ethernet MAC/PHY.
- Limited internal cache and RAM
- Student version, with limited functionality.

7.2 System Architecture

The implemented system is organized into a set of hardware modules, each controlled by its own finite-state machine (FSM) and interconnected through simple, synchronous byte-stream interfaces. Every module performs one well-defined task in the simplified PXE/TFTP chain, and all of them operate in parallel, reacting to incoming data on a cycle-by-cycle basis. This structure makes the design predictable, easy to observe, and suitable for running entirely inside the FPGA without requiring real Ethernet hardware.

The full system consists of three main FSM-driven blocks:

- **TFTP Client FSM** – Generates a single Read Request (RRQ) at startup and produces ACK frames for each incoming DATA block. It extracts the block number from the received DATA packet and uses it to construct the corresponding ACK packet, ensuring correct sequencing.
- **TFTP Mini-Server TX FSM** – Produces a fixed sequence of TFTP DATA frames. Each frame contains valid and correctly formatted Ethernet, IPv4, UDP, and TFTP headers, along with a monotonically increasing block number. Payload transmission is intentionally omitted in this revision to avoid synchronization issues between ROM access and the streaming data path. This keeps the system deterministic and avoids timing uncertainty.
- **Network Accelerator RX Parser FSM** – Monitors the outgoing data stream and classifies packets by protocol (IPv4, UDP, TFTP). It inspects header fields at specific byte offsets and updates counters for DATA packets, ACK packets, TFTP opcodes, and other protocol-level statistics. The accelerator monitors the unified internal stream, meaning it sees traffic from both the Client and the Mini-Server. Although it does not alter the data stream, it demonstrates how hardware offload can track network activity in real time.

All modules share a common 50 MHz system clock and are kept in the same reset domain. Communication between blocks uses a uniform byte-wide streaming interface (*tvalid, tready, tdata, tlast*), similar to AXI-Stream but simplified for educational and analysis purposes. This unified interface allows the design to be simulated with cycle-level accuracy in ModelSim and captured internally with SignalTap, without the need for a MAC/PHY, external cabling, or network equipment.

The system does not attempt to implement full PXE negotiation, DHCP discovery, or actual file transfer. Instead, it focuses on a minimal, but representative TFTP exchange that is sufficient to validate header construction, FSM timing, control-flow sequencing, and offload behavior:

i) The client transmits an RRQ frame through the loopback stream.

ii) The mini-server transmits a series of DATA frames, each containing valid headers and a monotonically increasing block number.

iii) The client generates ACK frames using the block number extracted by its receive path.

iv) The Network Accelerator observes these streams, parsing header fields and maintaining counters for DATA, ACK, UDP, and IPv4 traffic.

The hierarchical architecture is shown in the figure below.

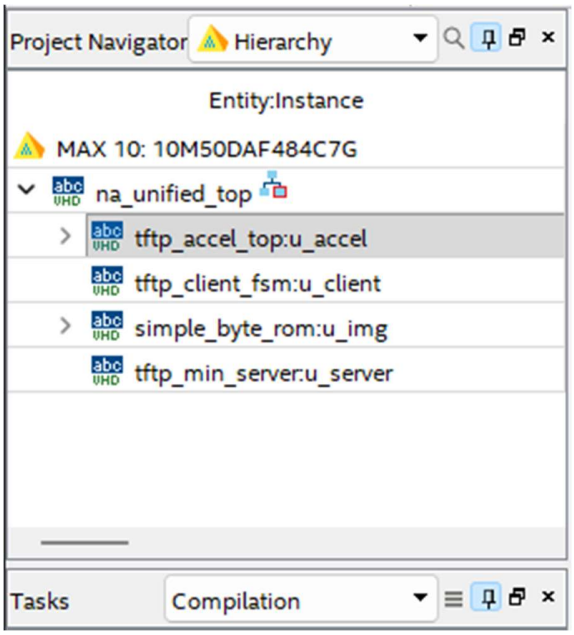


Figure 7: Hierarchical structure

7.3 System Behaviour Flow

The following subchapter summarizes the runtime behavior of the unified PXE/TFTP validation system. All blocks share a 50 MHz clock and are coordinated by the top-level entity *na_unified_top*, which synchronizes the external reset, interprets the mode switch SW0, and drives the PASS/FAIL LEDs.

I. **Reset and mode selection:** At power-up, the external reset is synchronized to *rst_n_sync*, and *SW0* is sampled into *mode_accel_only*. These signals generate local resets for the TFTP client and mini-server, and a global reset for the accelerator.
   - Validation Mode (SW0 = 0) - client and server enabled
   - Accelerator-Only Mode (SW0 = 1) - server disabled; client stays idle.

II. **Client RRQ generation:** When *rst_client_n* is released in Validation Mode, the Client FSM leaves TX_IDLE and sends one complete TFTP Read Request (Ethernet → IPv4 → UDP → TFTP RRQ). This frame is placed on the client TX stream and forwarded directly to the mini-server.

III. **Mini-server RRQ handling and DATA generation:** In this revision, the Mini-Server does not parse RRQ packets, but instead begins transmitting DATA frames when *server_enable* = *1*. Each DATA packet includes headers (Ethernet/IPv4/UDP) and a TFTP DATA header (opcode 3) with an incrementing block number.

IV. **Broadcast of DATA stream:** The server's TX stream feeds both the accelerator RX path and the client RX path. This means every DATA frame is simultaneously:
   - parsed by the accelerator, and
   - processed by the client receive logic.

V. **Accelerator RX parsing:** The accelerator walks through Ethernet, IPv4, UDP, and TFTP headers using an internal byte counter. It extracts:
   - opcode
   - block number
   - UDP/IP/Ethernet fields

   Payload bytes pass through unchanged; the accelerator acts only as a non-intrusive monitor and does not parse or count payload bytes in this revision, only header fields are decoded.

VI. **Client DATA reception and ACK scheduling:** The client RX path verifies the TFTP opcode, reads the block number, increments *blocks_seen*, and asserts *ack_pending*. These events raise *stat_data_seen* and keep *stat_active* high.

VII. **Client ACK generation:** With *ack_pending* = *1*, the Client TX FSM sends an ACK packet (Ethernet/IPv4/UDP + TFTP opcode 4 and the received block number).
After transmission, the FSM:

- clears *ack_pending*,
- increments *stat_ack_sent*,
- returns to TX_IDLE.

This closes the control loop and allows block-by-block transfer to continue until G_MAX_BLOCKS is reached.

VIII. **Session completion:** When all blocks have been transmitted and ACKed:

- the client drops *stat_active*,
- the mini-server returns to idle,
- the accelerator also returns to its idle state.

In Accelerator-Only Mode, this idle condition is reached immediately because no traffic is generated.

IX. **PASS/FAIL LED evaluation:** The top level derives the test result from client statistics and a watchdog:

- ***PASS:*** the first assertion of *stat_ack_c* latches *ack_seen* = *1* and sets *led_pass* = *1* permanently.
- ***FAIL:*** if no ACK is ever seen, a watchdog expiry asserts *led_fail*.

Under normal Validation Mode, ACKs always arrive and only *led_pass* lights.
A dedicated hardware test case can force the FAIL condition by disabling the server.

Together, these stages form a deterministic, fully on-chip model of a PXE-style TFTP transaction. This is shown in the following figure:

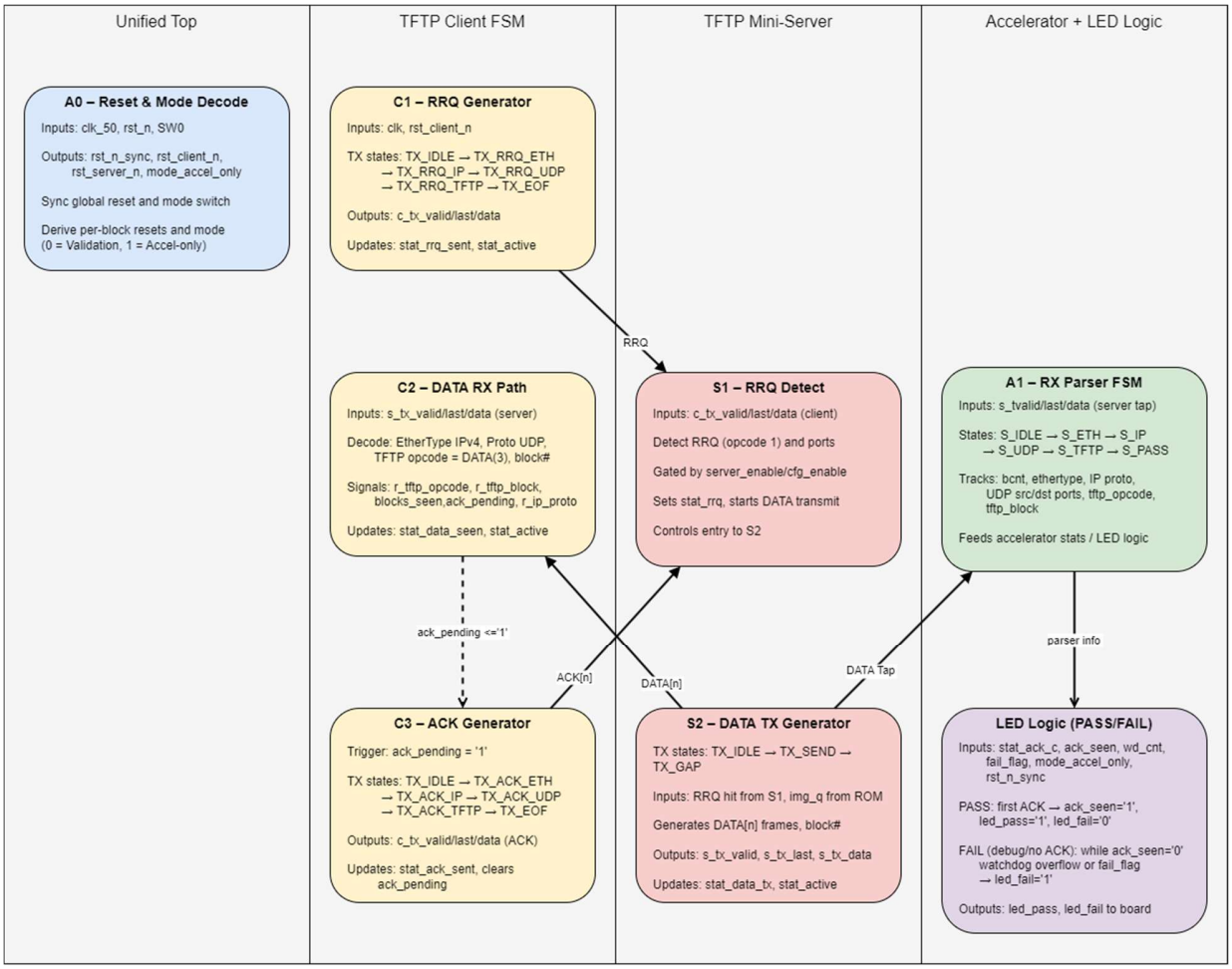


Figure 8: Flow diagram of system

7.4 Unified Control and Validation System

The unified control system serves as the top-level integration layer that coordinates all subsystems and manages test execution. It provides a common environment for the TFTP client, TFTP mini-server, and Network Accelerator to operate in a deterministic and fully observable manner.

Its responsibilities include:

- ***Reset and mode control:*** Distributes the global reset (rst_n) and interprets switch SW0 to select between Validation Mode (client + server active) and Accelerator-Only Mode (only the RX parser enabled).
- ***Status monitoring:*** Collects counters and status signals from the subsystems—RRQ count, DATA count, ACK count, active-transfer flag, and accelerator statistics—and drives the board LEDs to indicate basic PASS/FAIL behavior during hardware tests.
- ***Signal routing:*** Connects the byte-stream interfaces so that client TX and server TX traffic both pass through the accelerator's RX path. The internal loopback allows complete client–server interaction without any physical Ethernet hardware.
- ***Validation interface:*** Supports deterministic testing in both ModelSim and SignalTap by exposing all key signals, FSM states, and byte-level timing. This makes subsystem behavior easy to verify and compare across test runs.

The structure of this block is depicted in the table and figure below:

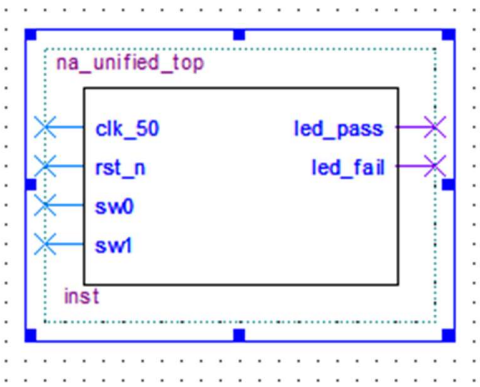


Figure 9: Unified control & validation top file

Table 3: Unified control & validation top file

Signal	Name	Direction	Type
System clock (50MHz)	clk_50	in	Std_logic
Reset (Active low)	rst_n	in	Std_logic
Switch 0	SW0	in	Std_logic
Switch 1	SW1	in	Std_logic
Pass signal	led_pass	out	Std_logic
Fail signal	led_fail	out	Std_logic

7.5 Subsystems

7.5.1 Ethernet and Transport Layer Subsystem

Although the system operates without a physical Ethernet interface, the Ethernet and transport layers are fully represented inside the TFTP Client FSM and the TFTP Mini-Server TX FSM. Both modules generate complete protocol headers directly in VHDL, producing well-formed frames suitable for validation and parsing.

The subsystem handles:

- ***Ethernet frame construction:*** destination/source MAC addresses and EtherType = 0x0800 (IPv4).
- ***IPv4 header construction:*** fixed Version/IHL fields, total-length values matching the generated frame, protocol ID 0x11 (UDP), and static source/destination IP addresses.
- ***UDP encapsulation:*** source/destination ports, header length fields, and a checksum set to zero (valid for IPv4 and sufficient for internal testing).
- ***TFTP header generation:*** opcode fields and block numbers for RRQ, DATA, and ACK messages.

This subsystem effectively models the complete protocol stack in a minimal hardware form, allowing end-to-end control of network-layer behavior without requiring a PHY or MAC. A more graphical representation of this is depicted on the figure below:

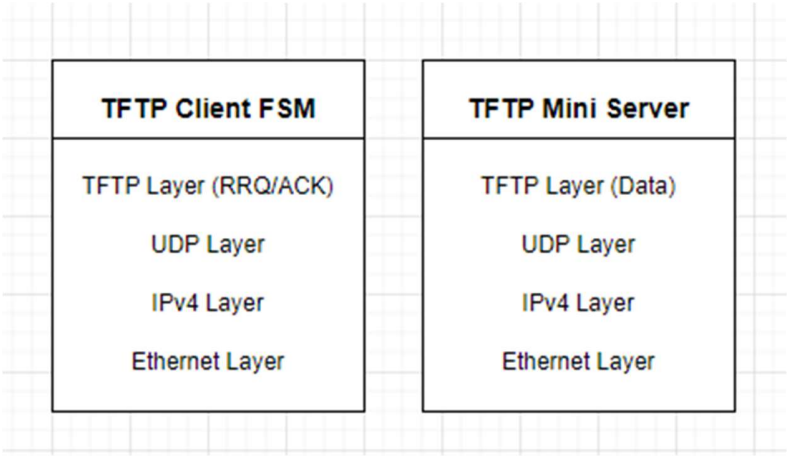


Figure 10: Ethernet and Transport layer

TFTP Client Subsystem

The TFTP Client FSM serves as the initiator of the exchange and as the acknowledgment generator for incoming DATA frames. It implements the following functions:

- ***Request Generation:*** Constructs a complete Ethernet → IPv4 → UDP → TFTP frame containing a Read Request in octet mode. This is the first packet sent in Validation Mode.
- ***Data Handling:*** Monitors incoming frames from the Mini-Server, detects TFTP DATA packets (opcode 3), and extracts their block numbers.
- ***ACK Transmission***: For every DATA block received, the client automatically builds and transmits a matching ACK (opcode 4) using the same layered header structure.
- ***State Control:*** Operates as a structured multi-stage FSM, progressing through Ethernet, IPv4, UDP, and TFTP output states for both RRQ and ACK sequences, with a shared TX_EOF termination state.
- ***Status and Counters:*** Tracks the number of RRQ, DATA, and ACK events, along with an active-session indicator, supporting both simulation-based and on-board validation.
- ***Debug Visibility:*** Exposes its current FSM state (dbg_tx_state) for direct observation in SignalTap.

In summary, the client subsystem acts as a compact, deterministic TFTP agent that drives the entire on-chip exchange, enabling predictable timing and full visibility of all protocol layers. The structure of this block is illustrated in the accompanying table and figure.

The structure of this block is depicted in the table and figure below:

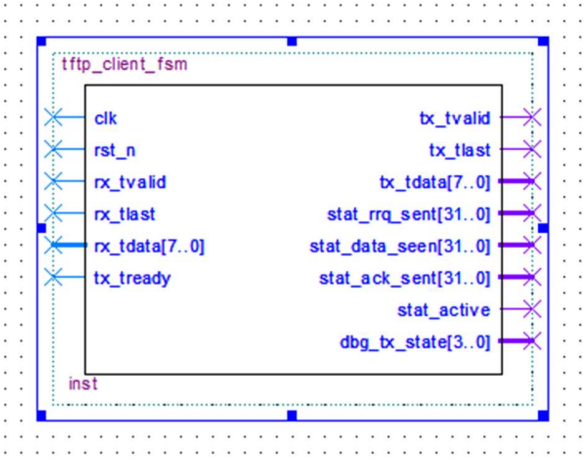


Figure 11: TFTP Client FSM

Table 4: TFTP Client FSM

Signal	Name	Direction	Type
System clock (50MHz)	clk	in	Std_logic
Reset (Active low)	rst_n	in	Std_logic
Rx data valid (handshake)	rx_tvalid	in	Std_logic
Rx last (end of frame)	rx_tlast	in	Std_logic
Rx data (payload)	rx_tdata	in	Std_logic_vector
Tx ready (handshake)	tx_tready	in	Std_logic
Tx valid (handshake)	tx_tvalid	out	Std_logic
Tx last (end of frame)	tx_tlast	out	Std_logic
Tx data (payload)	tx_tdata	out	Std_logic_vector
RRQ frames sent	stat_rrq_sent	out	Unsigned
TFTP Rx data sent	stat_data_sent	out	Unsigned
TFTP ACK frames sent	stat_ack_sent	out	Unsigned
Status signal	stat_active	out	Std_logic
Debug Tx state	dgb_tx_state	out	Std_logic_vector

TFTP Mini Server Subsystem

The TFTP Mini-Server FSM emulates a simplified TFTP server entirely in FPGA logic. Instead of implementing full request/response behavior, it functions as a deterministic DATA-frame generator that provides the client with a sequence of correctly formatted TFTP DATA packets. This enables closed-loop testing of the client FSM and the Network Accelerator without requiring a physical network or external server.

The Mini-Server performs the following functions:

- ***Data Frame Generation:*** The mini server is a pure transmit engine. Once *cfg_enable* is asserted and reset is released, it emits a fixed sequence of TFTP DATA frames (opcode 3) with monotonically increasing block numbers, without parsing any incoming packets.
- ***Header-Only Operation:*** Each transmitted frame is 46 bytes long: 14-byte Ethernet header, 20-byte IPv4 header, 8-byte UDP header, and a 4-byte TFTP header (opcode + block number). The payload length in the UDP/IP headers is set to 4 bytes, but no actual data bytes follow the TFTP header. The payload ROM ports (*img_addr, img_rd, img_q*) are present for future use but are tied off in this revision, so the ROM is never read.
- ***Simple State Control:*** A three-state FSM (TX_IDLE → TX_SEND → TX_GAP) produces one byte per clock when *tx_tready = '1'*. In TX_SEND the FSM steps through the header template, and at the last TFTP header byte (*tx_bcnt = 45*) it asserts *tx_tlast*, updates statistics, and moves to TX_GAP for a single idle cycle before returning to TX_IDLE.
- ***Statistics Tracking:*** The module maintains a counter of DATA frames sent. For compatibility with earlier versions, the RRQ counter is forced to the value 1 and the ACK counter remains 0, since no RX parsing is implemented in this revision.

In summary, the Mini-Server acts as a small, deterministic traffic generator that reproduces the TFTP DATA path in hardware. Its predictable timing and fully observable state transitions make it ideal for validating the client FSM and the accelerator logic.

The structure of this block is depicted in the table and figure below:

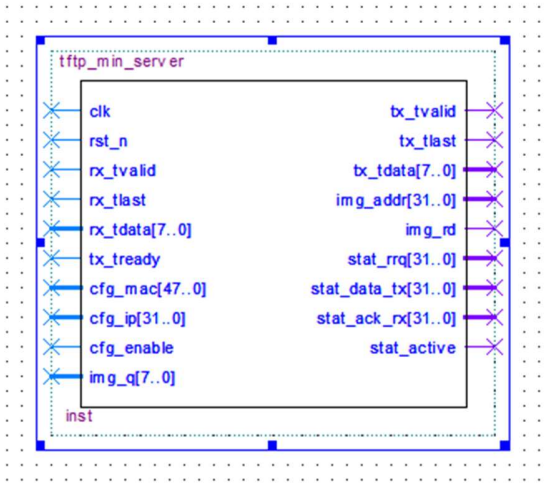


Figure 12: TFTP Mini Server

Table 5: TFTP Mini Server

Signal	Name	Direction	Type
System clock (50MHz)	clk	in	Std_logic
Reset (Active low)	rst_n	in	Std_logic
Rx data valid (handshake)	rx_tvalid	in	Std_logic
Rx last (end of frame)	rx_tlast	in	Std_logic
Rx data (payload)	rx_tdata	in	Std_logic_vector
Tx ready (handshake)	tx_tready	in	Std_logic
Server MAC address (source)	cfg_mac	in	Std_logic_vector
Server IPv4 address (source)	cfg_ip	in	Std_logic_vector
Global enable	cfg_enable	in	Std_logic
Data byte from ROM	img_q	in	Std_logic_vector
Tx valid (handshake)	tx_tvalid	out	Std_logic
Tx last (end of frame)	tx_tlast	out	Std_logic
Tx data (payload)	tx_tdata	out	Std_logic_vector
Byte address to ROM	img_addr	out	Unsigned
Read enable	img_rd	out	Std_logic
RRQ packets received	stat_rrq	out	Unsigned

TFTP data packets transmitted	stat_data_tx	out	Unsigned
TFTP ACK frames received	stat_ack_rx	out	Unsigned
Status signal	stat_active	out	Std_logic

7.5.2 Image ROM Subsystem

The Image ROM subsystem is included in the design as a future payload source for full TFTP file-transfer support. It is implemented as an on-chip, pre-initialized memory block, but in the current revision the TFTP Mini-Server does not read from it, and no payload bytes are inserted into DATA packets.

Key characteristics:

- ***Pre-initialized storage:*** ROM is synthesizable and can hold an image, but its content is unused in this revision and not required for system operation.
- ***Single-cycle access:*** Designed for straightforward byte retrieval once payload support is implemented.
- ***Reserved functionality:*** The Mini-Server FSM currently sends header-only DATA packets, so the ROM remains idle in this version.
- ***Simulation flexibility:*** The ROM contents can be changed easily for future development or testing.

In the current VHDL, the mini server never asserts *rd* and fixes *addr* to zero, so no payload bytes (*q*) are ever fetched from the ROM. All transmitted DATA frames therefore remain header-only. The ROM exists purely as a placeholder for a future payload implementation.

Although inactive in the present design, the ROM subsystem provides the required infrastructure for extending the system into a complete payload-capable TFTP file-transfer engine.

The structure of this block is depicted in the table and figure below:

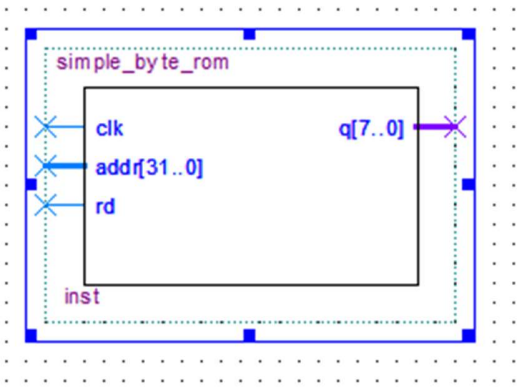


Figure 13: Image ROM system

Table 6: Image ROM system

Signal	Name	Direction	Type
System clock (50MHz)	clk	in	Std_logic
ROM address	addr	in	Unsigned
Read enable	rd	in	Std_logic
ROM data out	q	out	Std_logic_vector

7.5.3 Network Accelerator Subsystem

The Network Accelerator subsystem acts as the hardware-offload component of the design. In this revision, it functions as a real-time RX parser that observes all traffic flowing through the internal loopback and extracts key protocol information for monitoring and analysis.

Core functions:

- ***Packet classification:*** Parses Ethernet, IPv4, UDP, and TFTP headers to identify packet types and extract fields such as TFTP opcode and block number.
- ***Protocol statistics:*** Maintains counters for IPv4 packets, UDP packets, TFTP DATA frames, and TFTP ACK frames, providing a hardware-based view of protocol activity.
- ***Inline monitoring:*** Operates transparently between the client and server streams, allowing every transmitted byte to be inspected without altering the data.
- ***Standalone mode:*** In Accelerator-Only Mode (SW0 = 1), the module runs independently to validate header parsing and counter logic without requiring client–server interaction.

Although it does not modify, delay, or generate traffic in this version, the accelerator establishes a practical foundation for exploring hardware offload strategies, such as header verification, automated ACK handling, or flow-control assistance in future extensions.

The accelerator includes a small ROM, but only because the internal parser module exposes ROM ports. In this version, the parser never reads from it, so the ROM has no functional role. It is simply included as a dummy to satisfy the module interface and keep the design compatible with future payload-inspection features.

The structure of this block is depicted in the table and figure below:

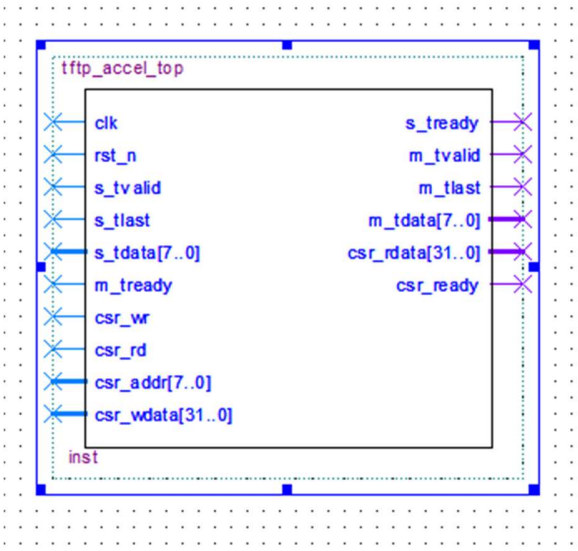


Figure 14: Network Accelerator system

Table 7: Network Accelerator system

Signal	Name	Direction	Type
System clock (50MHz)	clk	in	Std_logic
Reset (Active low)	rst_n	in	Std_logic
Source stream valid (handshake)	s_tvalid	in	Std_logic
Source stream last (end of frame)	s_tlast	in	Std_logic
Source stream data (payload)	s_tdata	in	Std_logic_vector

Master stream ready (handshake)	m_tready	in	Std_logic
CSR write	csr_wr	in	Std_logic
CSR read	csr_rd	in	Std_logic
CSR address	csr_addr	in	Std_logic_vector
CSR write data	csr_wdata	in	Std_logic_vector
Source stream ready (handshake)	s_tready	out	Std_logic
Master stream valid (handshake)	m_tvalid	out	Std_logic
Master stream last (end of frame)	m_tlast	out	Std_logic
Master stream data (payload)	m_tdata	out	Std_logic_vector
CSR read data	csr_rdata	out	Std_logic_vector
CSR ready	csr_ready	out	Std_logic

8 FINITE STATE MACHINE ARCHITECTURE

This chapter describes the internal control structures of the PXE-style TFTP system, which relies on hardware-implemented finite state machines (FSMs) to drive all protocol behavior. Each major subsystem, the TFTP Client, the TFTP Mini-Server, and the Network Accelerator—implements its own FSM to control packet generation, header sequencing, and real-time stream parsing.

These FSMs provide deterministic timing, strict ordering of protocol events, and predictable interaction with the system's streaming interfaces. Their coordinated operation forms the control backbone of the design.

The following sections outline the role of each FSM, its state transitions, and the way these control engines work together to emulate a simplified PXE/TFTP exchange entirely in FPGA logic.

8.1 TFTP Client FSM – Request/Acknowledgement Engine

This sub-component is the control state machine for the TFTP client. It is responsible for generating the outbound network traffic that drives the validation loop: first a single Read Request (RRQ), then a series of Acknowledgement (ACK) packets for each incoming data block. All instructions are expressed as states in the FSM, and the transitions are governed by a small set of conditions such as "RRQ already sent", "ACK pending", "maximum blocks reached", and the streaming handshake (*tx_tready*).

A state diagram of the TFTP client FSM can be seen in the figure below.

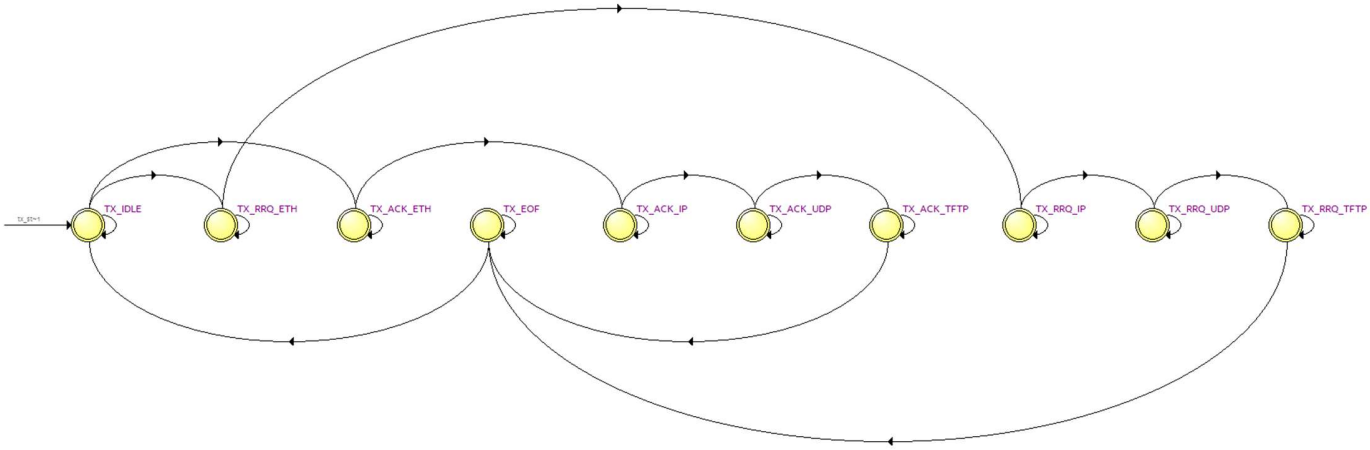


Figure 15: TFTP Client FSM

The different states can be described as follows:

i) **TX_IDLE – Idle/Decision State**
The FSM begins and returns to this state between packets. It decides what to transmit next:

- If RRQ has not yet been sent, it starts the RRQ transmit sequence.
- If a DATA block has been received, the ACK transmit sequence starts.
- Otherwise, it remains idle.

From this point, the FSM branches into either the RRQ or ACK transmit path.

RRQ transmit sequence (first packet only)

ii) **TX_RRQ_ETH – RRQ Ethernet header**
Outputs the 14-byte Ethernet header (destination MAC, source MAC, EtherType = IPv4). When complete, the FSM advances.

iii) **TX_RRQ_IP – RRQ IPv4 header**
Transmits the fixed 20-byte IPv4 header for the RRQ. Fields such as Version/IHL, Total Length, Protocol = UDP, and source/destination IPs come from generics. Checksum is set to zero.

iv) **TX_RRQ_UDP – RRQ UDP header**
Outputs the 8-byte UDP header: source port, destination port (typically 69), length, and checksum (zero).

v) **TX_RRQ_TFTP – RRQ TFTP payload**
Emits the TFTP RRQ content:

- Opcode = 1.
- Filename string hardcoded as constant byte vector.
- Null terminator.
- Mode string ("octet").
- Final null terminator.

An internal index selects each byte from constant vectors.

vi) **TX_EOF – End of RRQ frame**
Asserts tx_tlast for one cycle to terminate the frame. The RRQ is marked as sent, its counter is incremented, and the FSM returns to TX_IDLE.

ACK transmit sequence (for each DATA block)

vii) **TX_ACK_ETH – ACK Ethernet header**
Outputs the Ethernet header for the ACK frame. The destination MAC is taken from the static server configuration (G_SERVER_MAC), not from the received DATA packet.

viii) **TX_ACK_IP – ACK IPv4 header**
Transmits the IPv4 header for the ACK packet. Its structure mirrors the RRQ IP header, but the destination IP is taken from the configured server address (G_SERVER_IP) rather than extracted from the incoming DATA frame.

ix) **TX_ACK_UDP – ACK UDP header**
Outputs the 8-byte UDP header for the ACK: client port → server port, matching the expected TFTP format.

x) **TX_ACK_TFTP – ACK TFTP payload**
Outputs the 4-byte TFTP ACK:

- Opcode = 4.
- Block number from the last DATA packet.

After sending the payload, the ACK counter is updated.

Finally, TX_EOF is used again to complete the frame before returning to TX_IDLE.

In summary, the TFTP Client FSM operates as a compact packet generator that alternates between RRQ and ACK transmission based on live conditions (*rrq_sent, ack_pending, blocks_seen, and G_MAX_BLOCKS*). Its deterministic sequencing ensures that each header layer is emitted in the correct order and timing.

8.2 TFTP Mini-Server TX Generator FSM

The TFTP Mini-Server FSM is a compact transmit engine that generates a continuous sequence of TFTP DATA packets. Unlike a full server implementation, this simplified version produces header-only frames (no payload) and increments the block number for each packet. It acts as a deterministic traffic source for validating the client FSM and the Network Accelerator.

A state diagram of the TX FSM can be seen in the figure below.

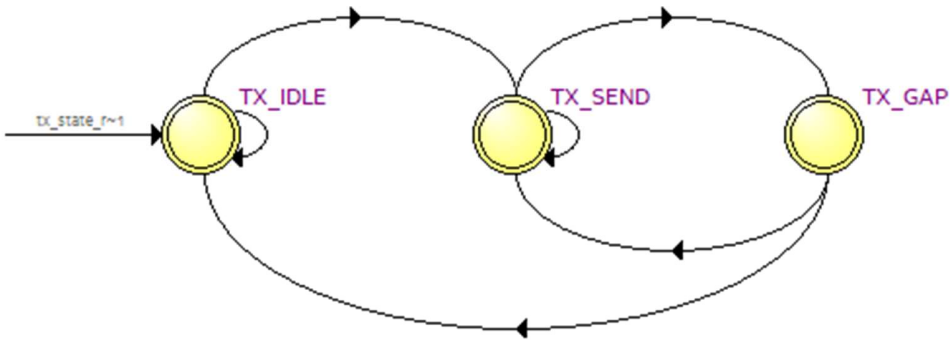


Figure 16: TFTP Mini Server TX Generator FSM

The states can be described as follows:

i) **TX_IDLE – Idle/Start State**

The FSM waits in this state until the server is enabled. When active, it does the following:

- Resets its internal byte counter.
- Loads the current block number into a register.
- Advances to the Ethernet header transmit state.

ii) **TX SEND – Header Streaming State**

In this state, the FSM outputs the entire DATA frame, one byte per cycle, whenever *tx_tready* = '1'.

A fixed sequence of header bytes is emitted:

I. Ethernet header (destination/server MAC, source/server MAC, EtherType).
II. IPv4 header (fixed fields; checksum bytes set from constants).

III. UDP header (ports, length = fixed value, checksum = 0)
IV. TFTP header
- Opcode = 3 (DATA).
- Block number = incrementing counter.

There is no payload in this revision; the frame ends immediately after the 4-byte TFTP header. An internal counter tracks how many bytes of the predefined header sequence have been transmitted. Once the last byte has been sent, the FSM transitions to TX_GAP.

iii) **TX_GAP – Inter-frame Gap**
This short state inserts a one-cycle pause between frames to simplify downstream parsing and ensure deterministic separation of packets.

During this state, the following happens:

- *block_counter* is incremented.
- The FSM waits for one clock cycle.
- Control returns to TX_IDLE, where the next DATA frame begins.

The Mini-Server therefore outputs a continuous stream of DATA packets with sequential block numbers.

8.3 Accelerator RX Parser FSM

The Network Accelerator FSM functions as a lightweight hardware offload block that inspects all traffic flowing from the Mini-Server. In this revision, only the RX parser FSM is active; the transmit path is disabled, and the accelerator focuses entirely on header parsing and protocol statistics.

A simplified conceptual diagram of the parser FSM is shown in the figure below:

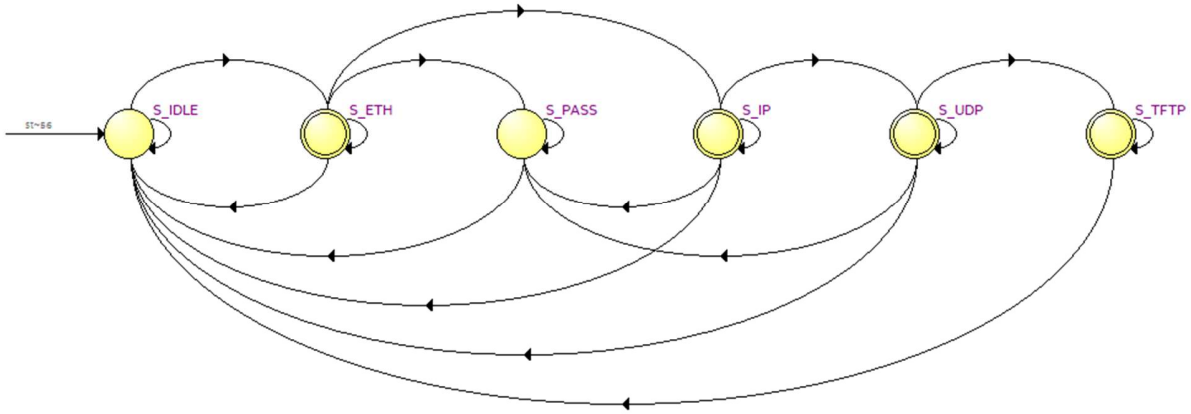


Figure 17: Accelerator RX Parser FSM

This FSM reads the incoming stream that originates from the TFTP mini-server's TX interface and is routed into the accelerator. Its purpose is to walk through the protocol stack and extract headers and TFTP fields.

The states can be described as follows:

i) **S_IDLE – Frame Wait State**

The FSM remains idle until it detects the first valid byte of a new frame (s_tvalid = '1').

At this point it does the following:

- Resets its internal byte counter.
- Moves to the Ethernet-parsing state.

ii) **S_ETH – Ethernet Header Parsing**

The accelerator consumes the 14-byte Ethernet header:

- Destination MAC.
- Source MAC.

- EtherType.

Once all Ethernet bytes have been read, the FSM verifies that the EtherType indicates IPv4 and proceeds accordingly.

iii) **S_IP** – **IPv4 header parsing**
Here the FSM processes the IPv4 header fields:

- Version and header length,
- Total length,
- Protocol identifier,
- Source and destination IP addresses.

The FSM uses a byte counter to locate each field and may verify that the protocol is UDP before advancing. After the IP header bytes have been read, the FSM proceeds to the UDP header state.

iv) **S_UDP – UDP header parsing**
In this state, the FSM reads and stores the UDP source and destination ports. These ports are later used to decide whether the packet belongs to the TFTP flow of interest. When all 8 bytes of the UDP header have been processed, the FSM transitions to the TFTP header.

v) **S_TFTP – TFTP header parsing**
In this state, the system captures and stores the 16-bit TFTP opcode and block number from specific, predetermined positions within the data. Whenever it sees opcode 0x0003 (DATA) or 0x0004 (ACK), it increments the corresponding TFTP counters and updates the last-block register. After these fields have been captured, the FSM remains in S_TFTP until the end of the frame, simply draining any remaining bytes (payload) without additional decoding.

vi) **S_PASS – Frame Completion**
In this state, where no TFTP traffic is executed, the system sinks the rest of the frame without touching any counters. Once *tlast* is seen, outer control resets the byte counter and returns to S_IDLE for the next frame.

In this revision, the accelerator focuses exclusively on header-level statistics (IPv4, UDP, TFTP opcode, and block number). Payload bytes are observed but not counted or interpreted, leaving room for future extensions such as block-size or throughput measurement.

9 TEST PROTOCOL

For the complete protocol, see appendix B

10 VERIFICATION AND VALIDATION

The verification phase was executed on ModelSim and the validation phase was executed on Signal Tap according to each test case, all which are depicted in the following figures.

10.1 Test Bench and Signal Tap Results

The following results are the product of several simulations, where each case has been thoroughly analyzed. For the results and assumptions made on every case, refer to the test protocol on Appendix B.

Case 1 - System resets to IDLE, no traffic, counters cleared, LEDs off:

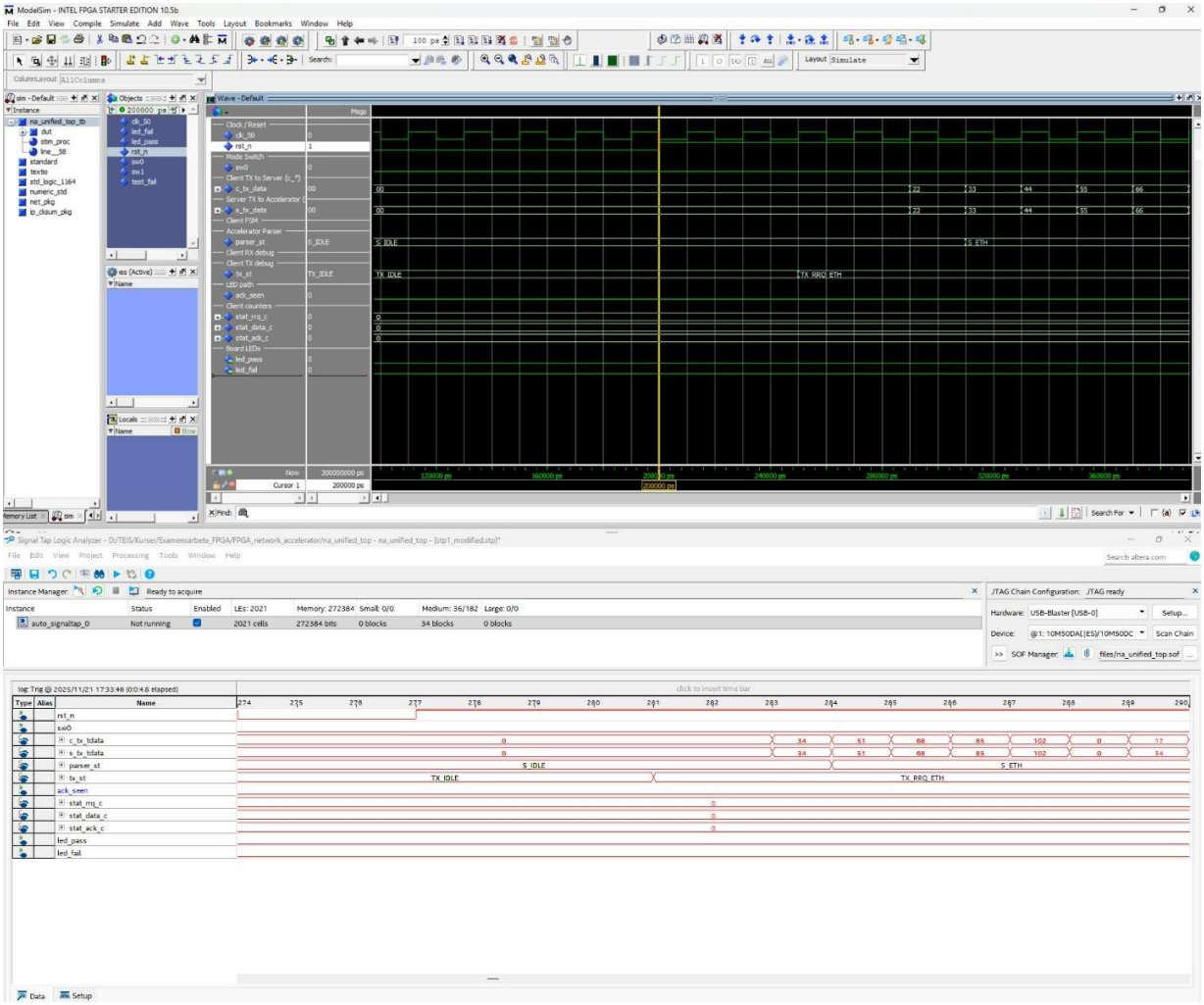


Figure 18: Case 1 – Modelsim (above) and SignalTap (below)

Case 2 - Client and Server both active, accelerator receives data from server:

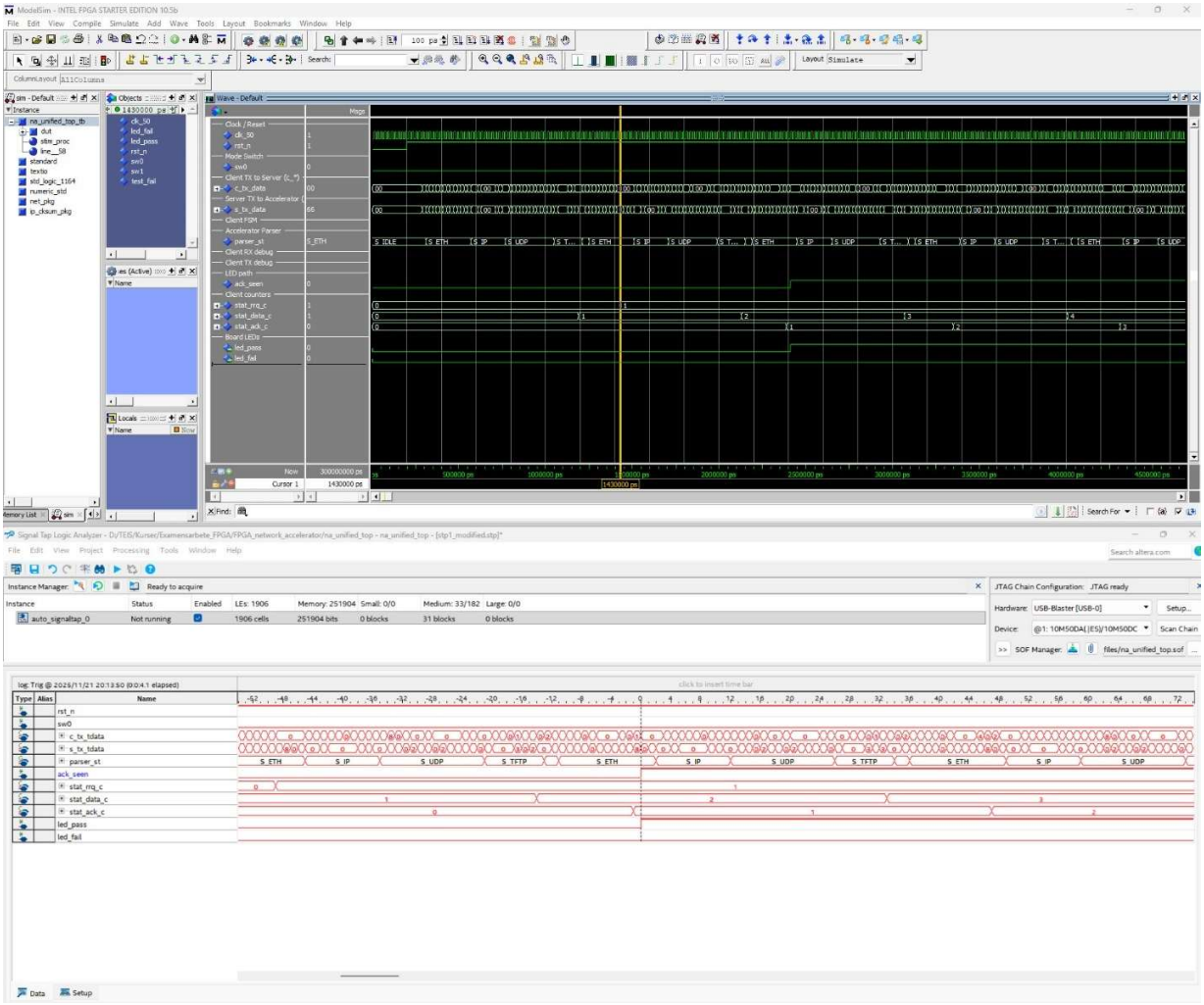


Figure 19: Case 2 – Modelsim (above) and SignalTap (below)

Case 3 & 8 - Client and Server held in reset:

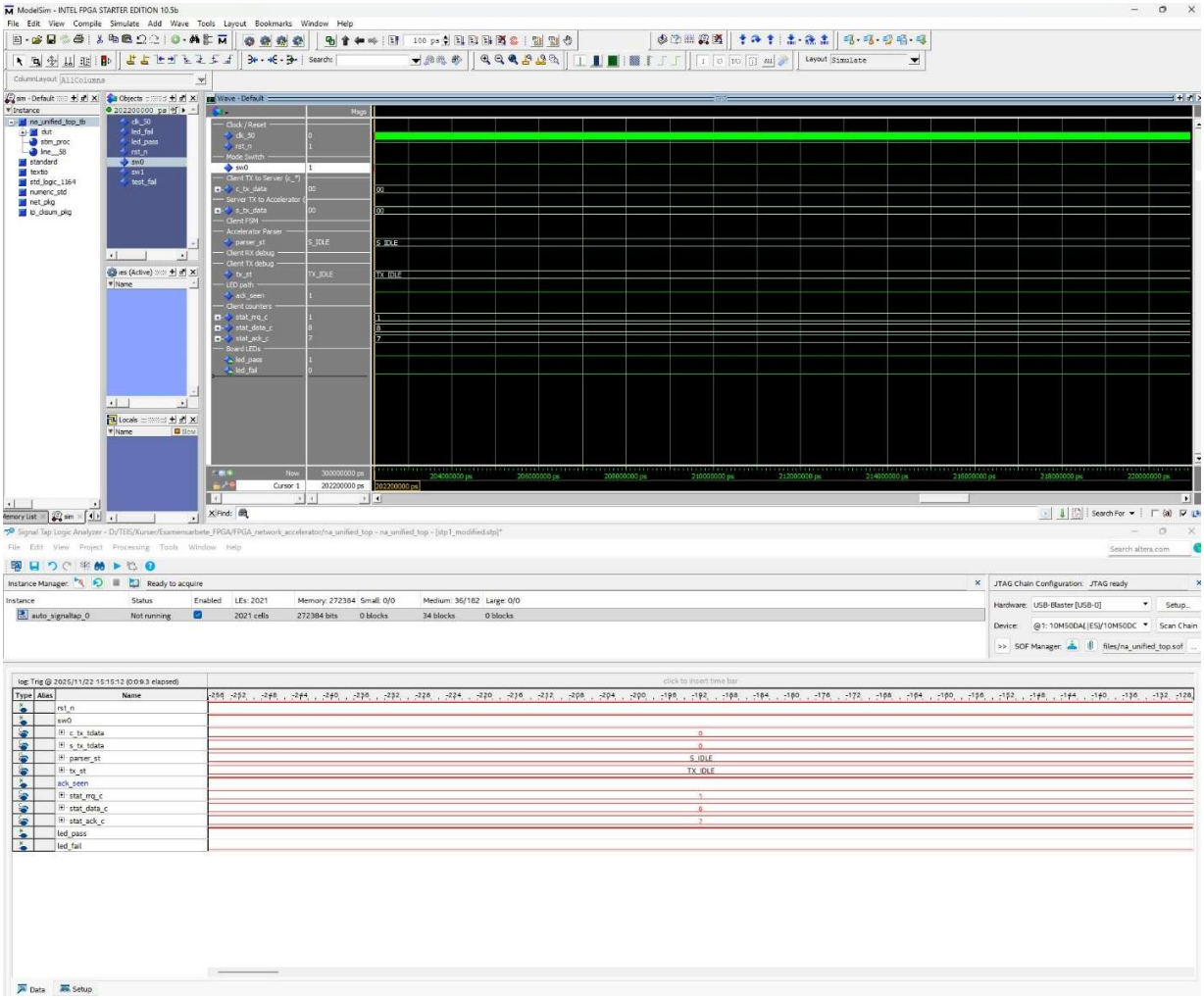


Figure 20: Case 3 & 8 – Modelsim (above) and SignalTap (below)

Case 4 - Client sends RRQ:

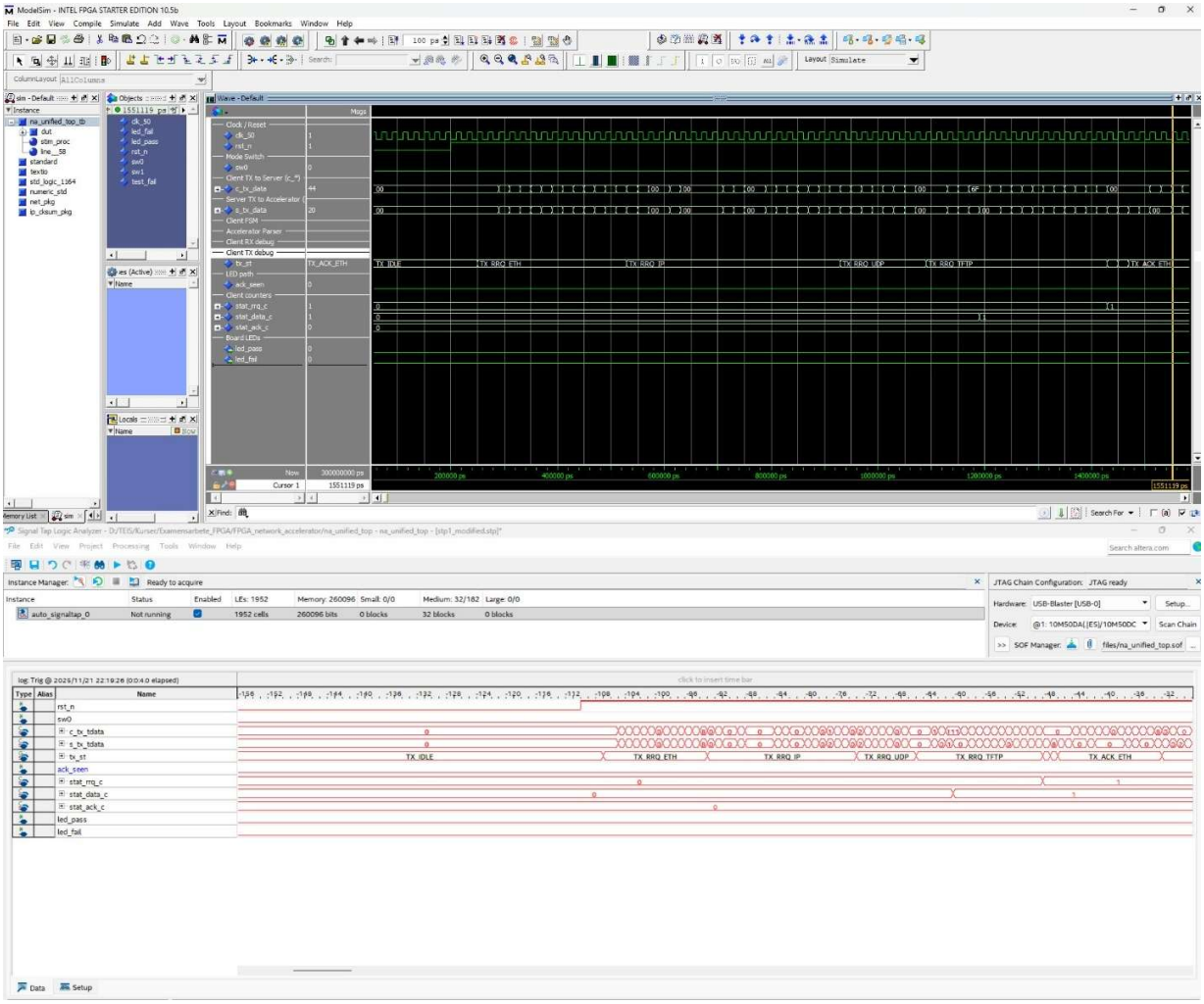


Figure 21: Case 4 – Modelsim (above) and SignalTap (below)

Case 5 - DATA frame detected:

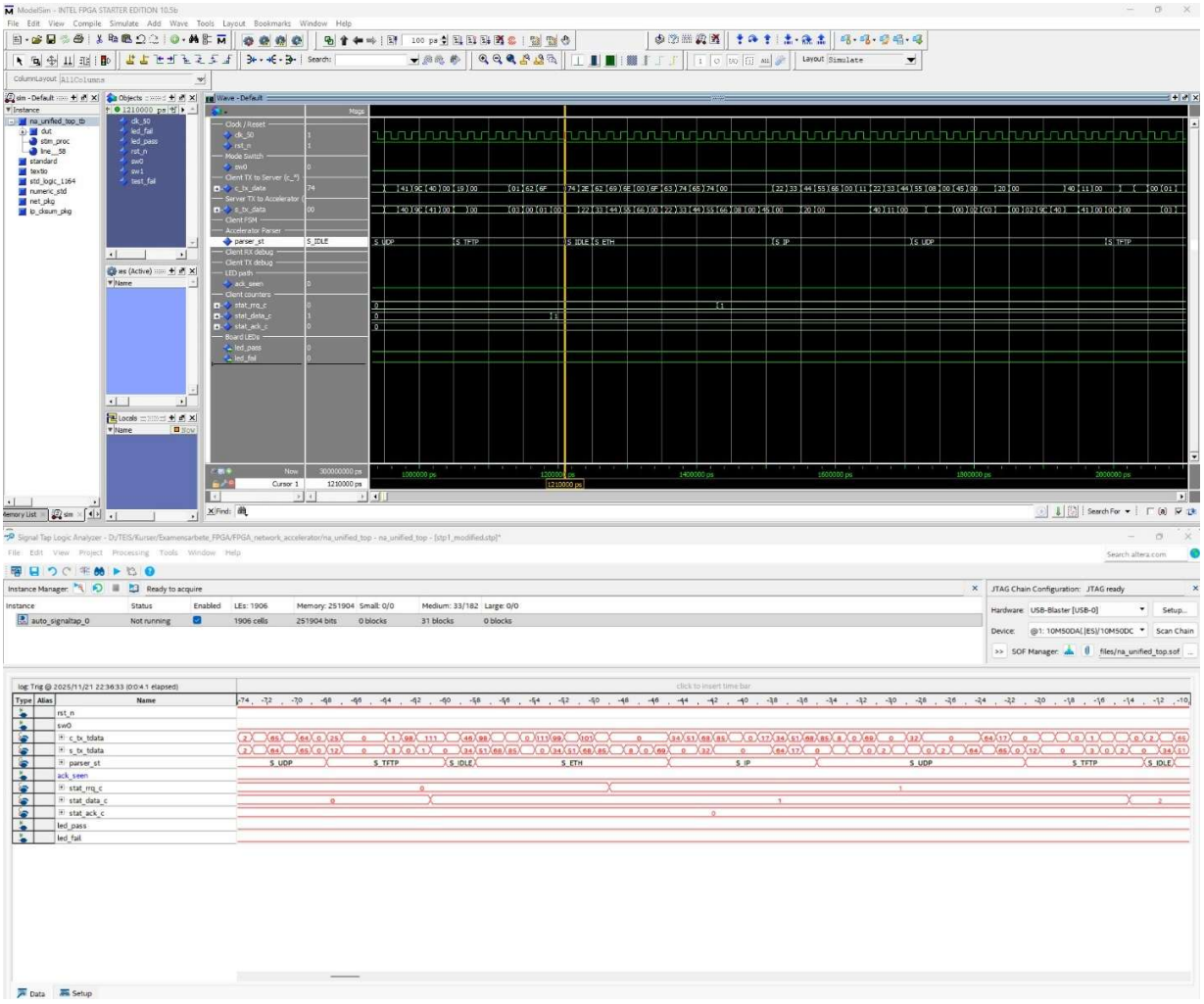


Figure 22: Case 5 – Modelsim (above) and SignalTap (below)

Case 6 - Client sends ACK:

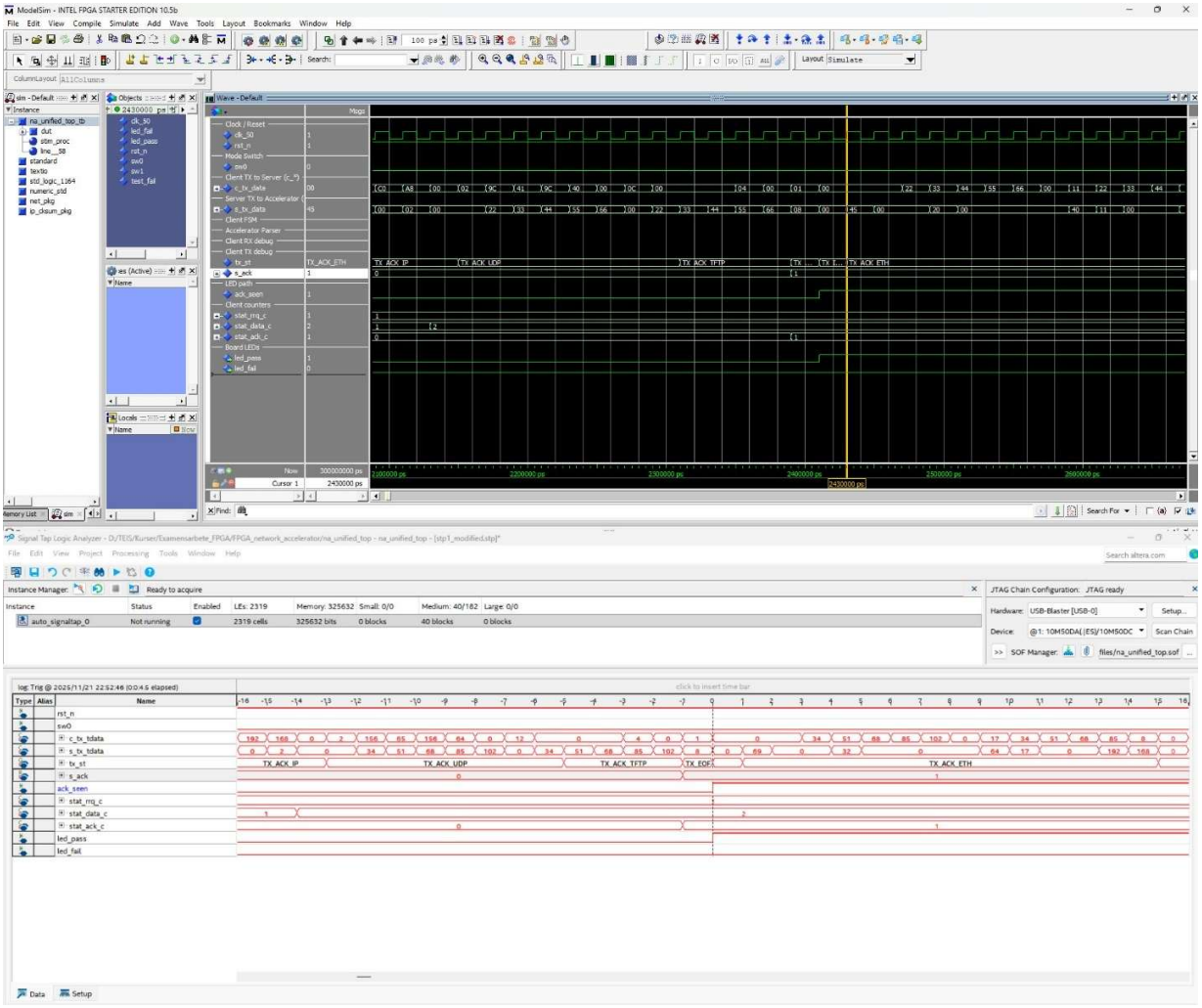


Figure 23: Case 6 – Modelsim (above) and SignalTap (below)

Case 7 - Session ends:

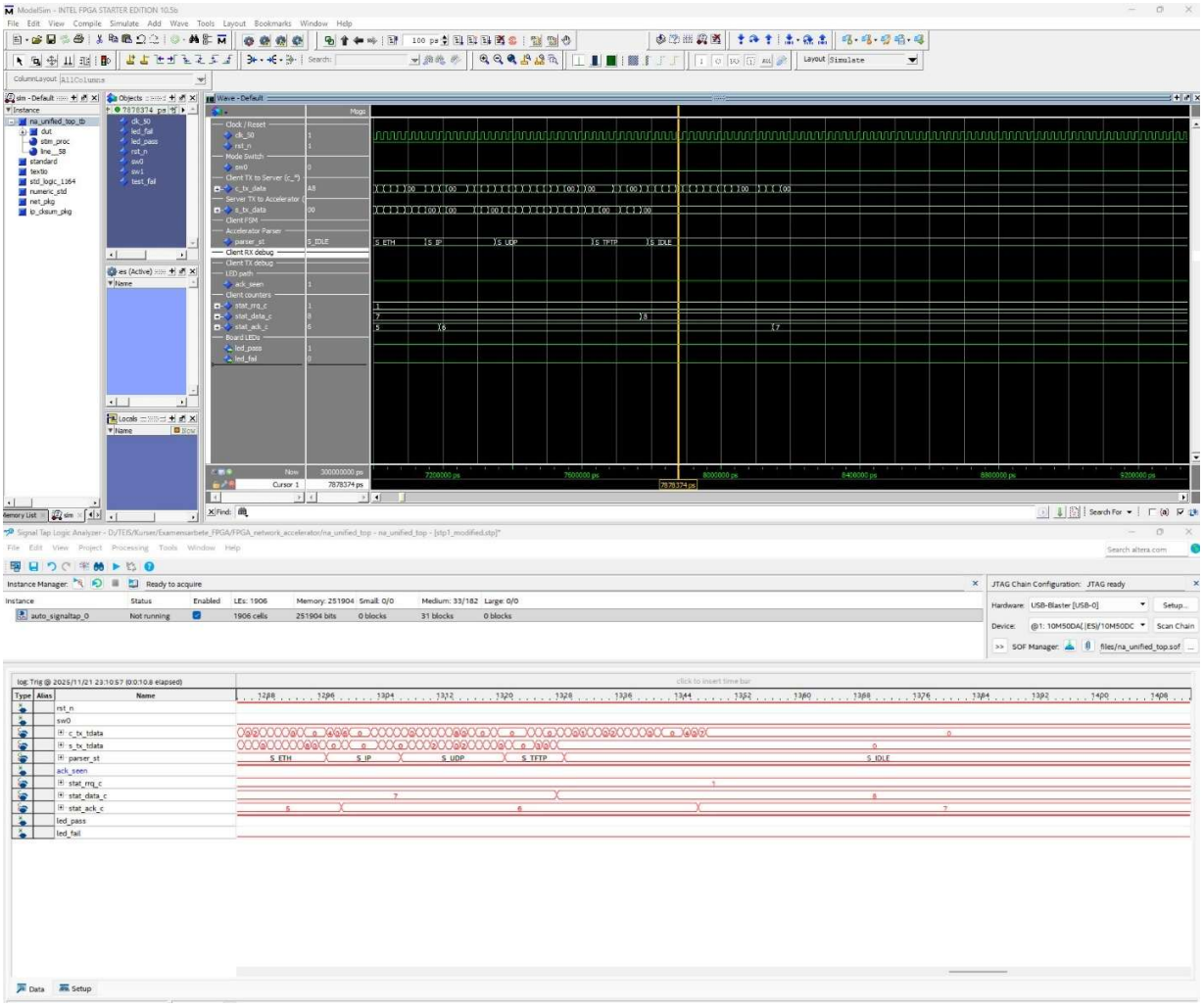


Figure 24: Case 7 – Modelsim (above) and SignalTap (below)

Case 9A – Clean restart, before reset:

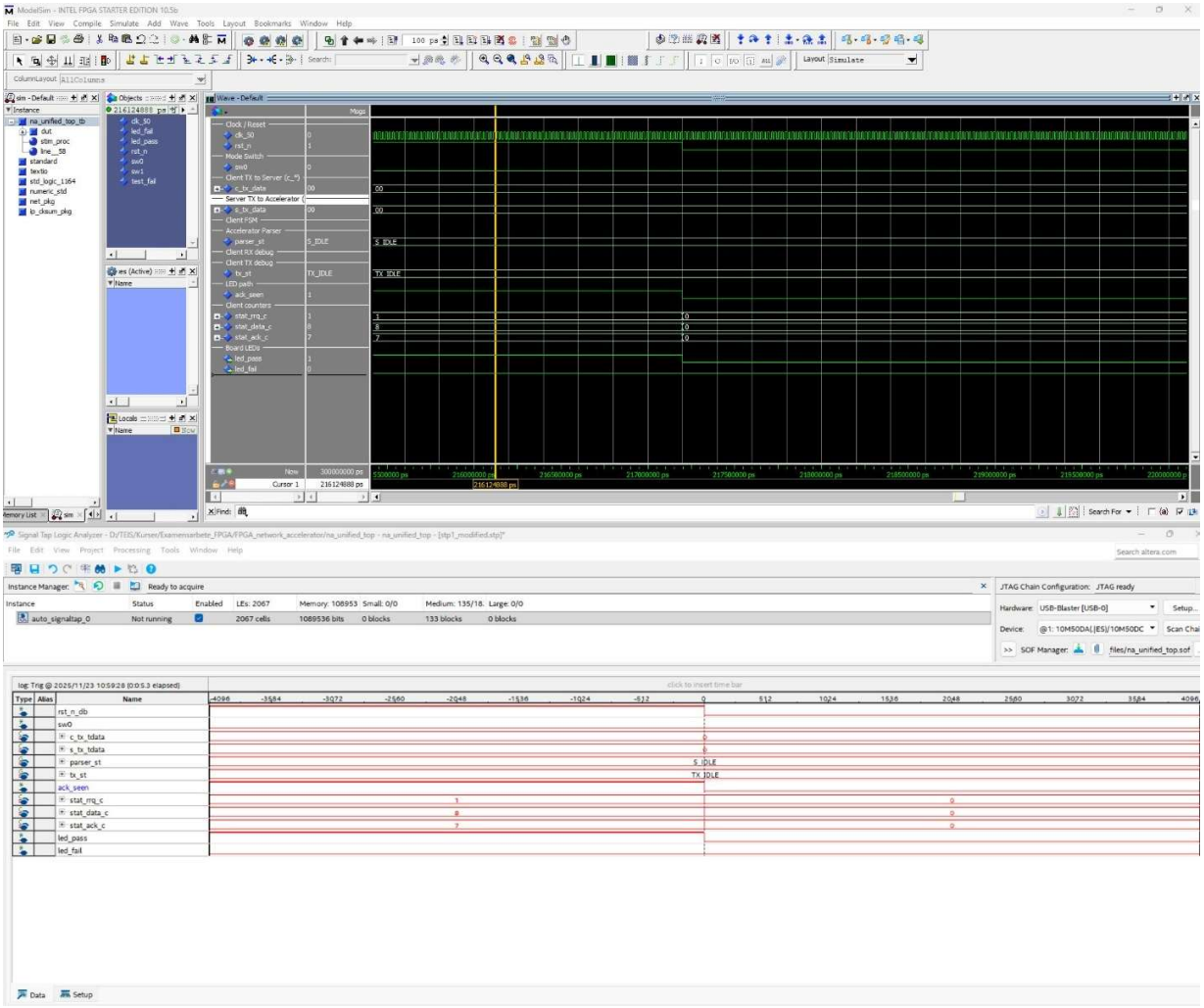


Figure 25: Case 9A - Modelsim (above) and SignalTap (below)

Case 9B – Clean restart, after reset:

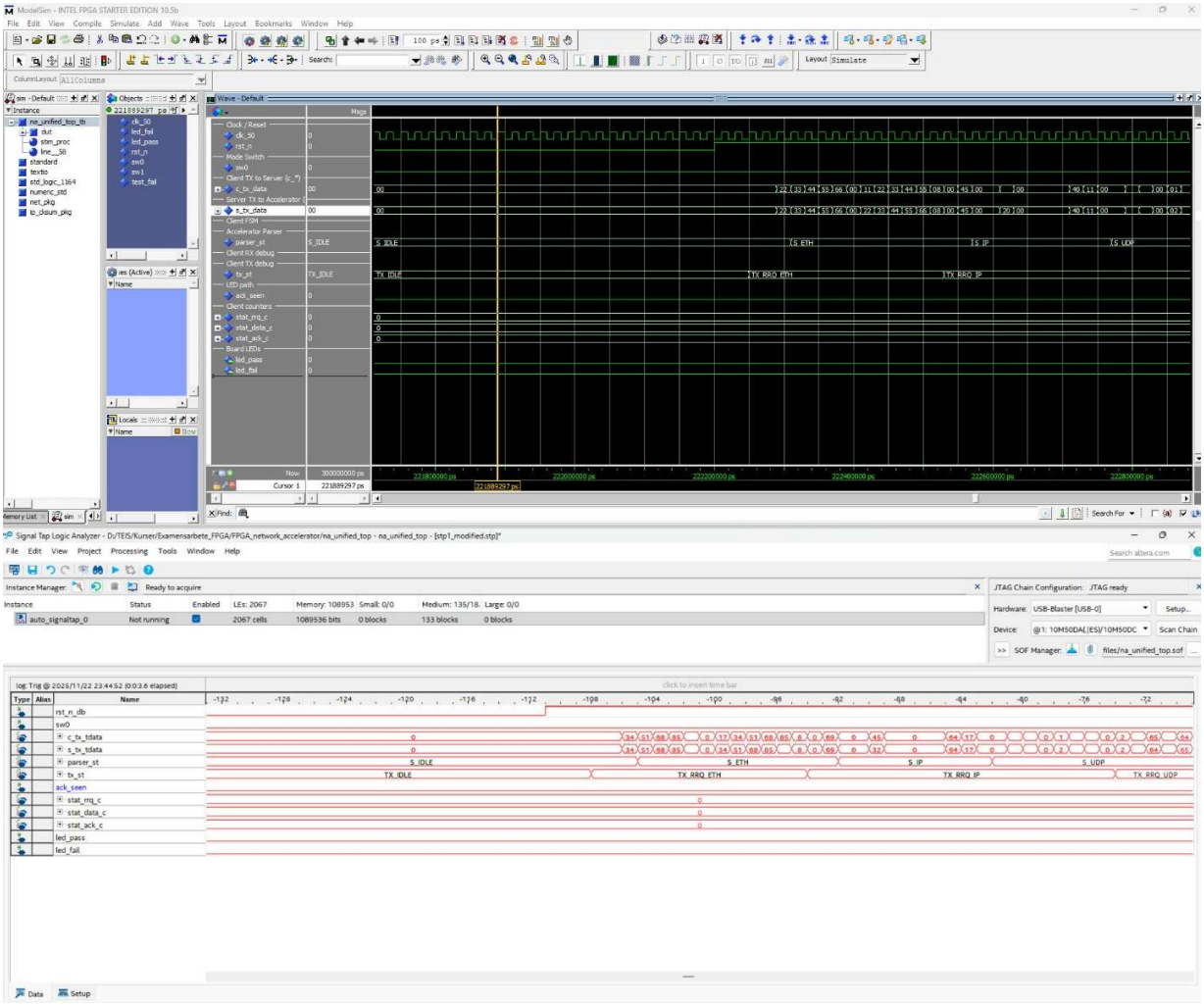


Figure 26: Case 9B - Modelsim (above) and SignalTap (below)

Case 10 - Watchdog timeout:

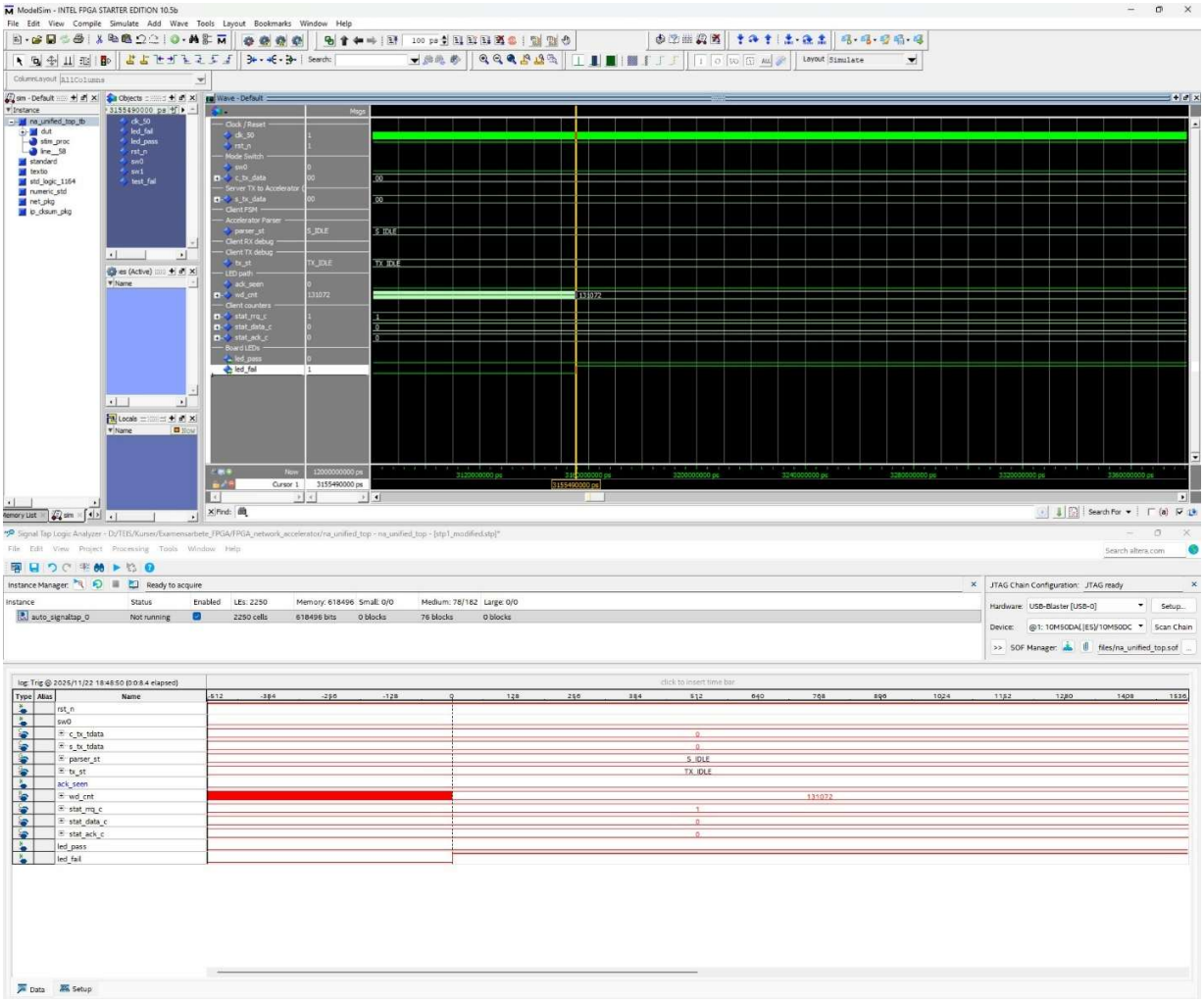


Figure 27: Case 10 – Modelsim (above) and SignalTap (below)

Additional proof Case 9 – Modelsim:

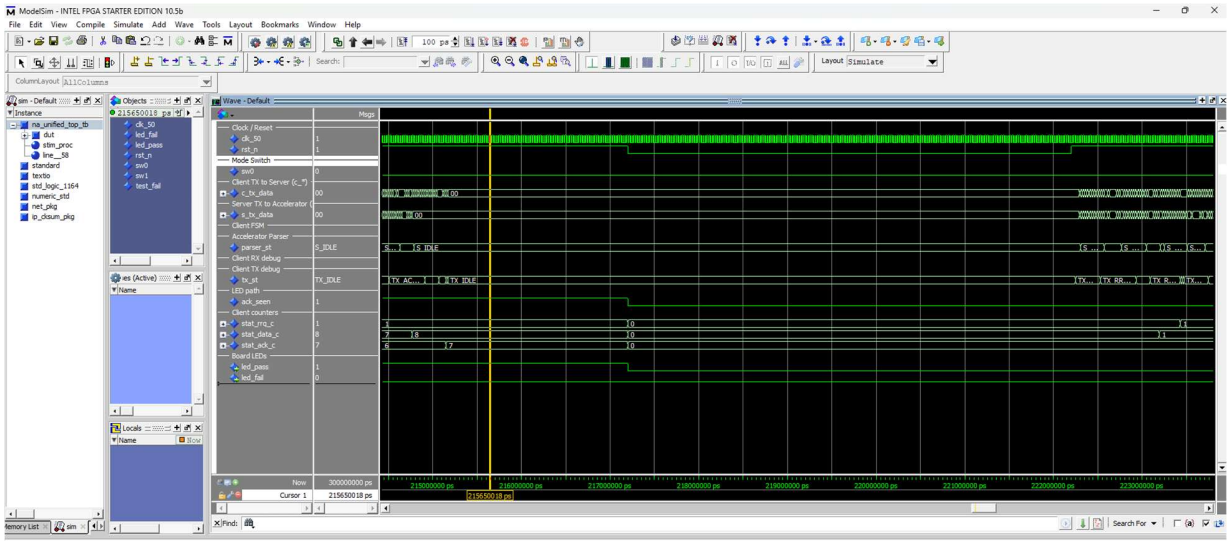


Figure 28: Case 9 extra shot - Modelsim

Validation on board:

PASS LED (normal mode, network traffic working, on the left)
FAIL LED (watchdog timeout, network traffic interrupted, on the right)

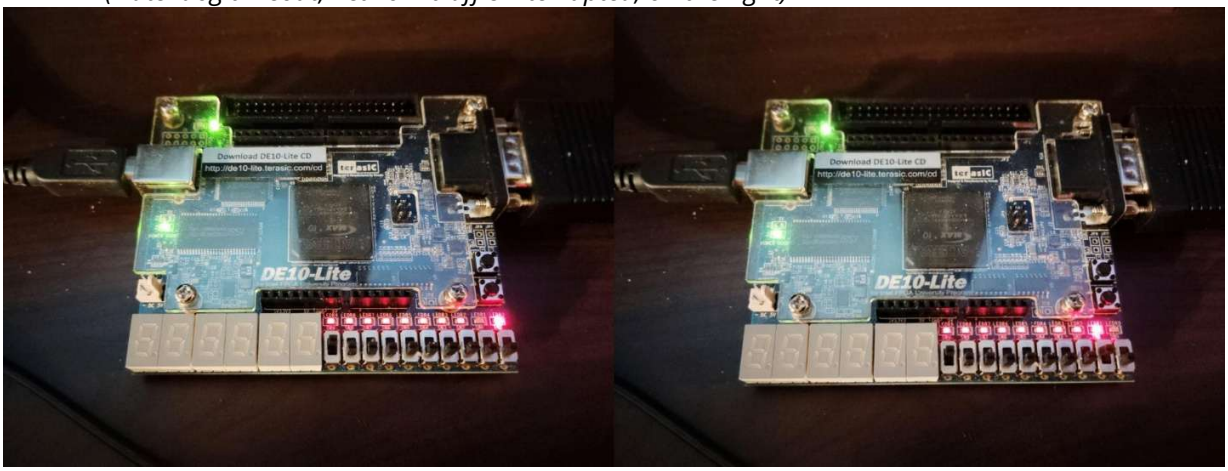


Figure 29: Normal mode and fail mode

11 TIMING ANALYSIS

This section presents the results obtained from the Timing Analyzer in Quartus Prime. The Timing Analyzer (from hereon called the tool) provides detailed timing characterization of the design by evaluating clock constraints, generated clocks, and all associated timing paths. The primary metrics of interest are the maximum achievable frequency, the minimum frequency, and the shortest slack, all of which indicate how comfortably the design meets timing requirements.

11.1 Frequency Analysis

The tool automatically determines the highest clock frequency the implemented design can sustain by analyzing the critical path delays. This is reported as the Fastest Achievable Frequency. It reflects the maximum clock rate at which all setup constraints can still be satisfied without timing violations.

In contrast, the slowest frequency is also displayed in the analysis, which serves as a comparison. The following figures show the fastest and slowest allowed frequency on the system:

Fastest frequency:

Fast 1200mV 0C Model

	Fmax	Restricted Fmax	Clock Name	Note
1	130.89 MHz	130.89 MHz	internal_clock_50MHz	
2	186.53 MHz	186.53 MHz	altera_reserved_tck	

Figure 30: Fastest frequency

Slowest frequency:

Slow 1200mV 85C Model

	Fmax	Restricted Fmax	Clock Name	Note
1	73.36 MHz	73.36 MHz	internal_clock_50MHz	
2	93.35 MHz	93.35 MHz	altera_reserved_tck	

Figure 31: Slowest frequency

11.2 Net Analysis – shortest slack

Slack represents the difference between the required arrival time and the actual arrival time of a signal. A positive slack indicates that the timing requirement has been met, while a negative slack signals a violation.

The shortest slack reported by the tool shows the most critical timing path in the system — the path closest to violating timing. Even a small positive slack is acceptable, but a larger margin provides extra robustness and indicates headroom for further logic or frequency increases. The following figure shows the most prominent signals/register/nodes in the system regarding slack time:

Shortest slack:

Fast 1200mV 0C Model

Command Info | Summary of Paths

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	12.360	fail_l	led_fail	internal_clock_50MHz	internal_clock_50MHz	20.000	-1.811	3.759
2	12.804	pass_l	led_pass	internal_clock_50MHz	internal_clock_50MHz	20.000	-1.789	3.337
3	15.251	sld_sign...e_reg[0]	sld_sign...a_we_reg	internal_clock_50MHz	internal_clock_50MHz	20.000	0.040	4.561
4	15.340	sld_sign...e_reg[0]	sld_sign...ess_reg0	internal_clock_50MHz	internal_clock_50MHz	20.000	0.049	4.481
5	15.341	sld_sign...e_reg[0]	sld_sign...a_we_reg	internal_clock_50MHz	internal_clock_50MHz	20.000	0.042	4.473
6	15.341	sld_sign...e_reg[0]	sld_sign...ain_reg0	internal_clock_50MHz	internal_clock_50MHz	20.000	0.042	4.473
7	15.350	sld_sign...e_reg[0]	sld_sign...ess_reg0	internal_clock_50MHz	internal_clock_50MHz	20.000	0.053	4.475
8	15.351	sld_sign...e_reg[0]	sld_sign...a_we_reg	internal_clock_50MHz	internal_clock_50MHz	20.000	0.046	4.467
9	15.351	sld_sign...e_reg[0]	sld_sign...ain_reg0	internal_clock_50MHz	internal_clock_50MHz	20.000	0.046	4.467
10	15.352	sld_sign...e_reg[0]	sld_sign...ess_reg0	internal_clock_50MHz	internal_clock_50MHz	20.000	0.054	4.474

Figure 32: Shortest slack

As shown above, *fail_l* and *pass_l* are the nodes with the shortest slack, and their RTL form is shown in the figure below:

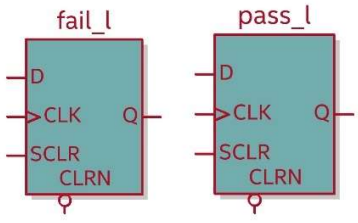


Figure 33: RTL view

11.3 Design Analysis

The tool performs a full timing check of the design under all applied SDC constraints, covering clocks, inputs, outputs, and derived signals. This analysis verifies that all synchronous paths meet setup and hold requirements, that no clocks or I/O pins are left unconstrained, and that clock domain relationships and false paths are correctly defined.

Overall, the results show that the design is fully constrained and meets timing at the target frequency, confirming stable and reliable operation under the specified conditions.

Slack Hold and Setup:

The positive setup and hold slack values on the table below show that all timing paths meet requirements across all temperature and voltage corners.

Table 8: Slack analysis

na_unified_top	**Slack Setup Internal clock 50MHz**	**Slack Hold Internal clock 50MHz**
Slow 1200mV 85C	6,369	0,239
Slow 1200mV 0C	7,291	0,242
Fast 1200mV 0C	12,360	0,093

Fmax analysis:

The Fmax results on the table below exceed the required 50 MHz, confirming that the design can operate safely above the target frequency under all conditions.

Table 9: Frequency analysis

na_unified_top	**FMax Internal clock 50MHz**	**Restricted FMax Internal clock 50MHz**
Slow 1200mV 85C	73,36 MHz	73,36 MHz
Slow 1200mV 0C	78,68 MHz	78,68 MHz
Fast 1200mV 0C	130,89 MHz	130,89 MHz

12 OPTIMIZATION

This chapter presents the optimization steps performed on the design and the resulting improvements in both area usage and power consumption. Optimization was carried out after the initial implementation to reduce resource utilization, improve efficiency, and ensure that the system operates within the desired power and performance constraints.

12.1 Area Optimization

Area optimization focuses on reducing the number of logic elements, registers, and other hardware resources used by the design. This can improve fitting results, reduce routing congestion, and potentially increase maximum operating frequency. The following figures show this concept:

Before optimization:

Fitter Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Logic Cells	Dedicated Logic Registers	I/O Registers	Memory Bits	M9Ks	UFM Blocks	DSP Elements	DSP 9x9	DSP 18x18	Pins	Virtual Pins	LUT-Only LCs	Register-Only
1	|na_unified_top	2409 (72)	1876 (25)	0 (0)	1090112	134	1	0	0	0	6	0	533 (47)	1088 (3)
1	|sld_hub:auto_hub|	152 (1)	91 (0)	0 (0)	0	0	0	0	0	0	0	0	61 (1)	24 (0)
2	|sld_signaltap:auto_signaltap_0|	1501 (140)	1381 (138)	0 (0)	1089536	133	0	0	0	0	0	0	120 (2)	972 (138)
3	|tftp_accel_top:u_accel|	149 (0)	95 (0)	0 (0)	576	1	0	0	0	0	0	0	54 (0)	28 (0)
4	|tftp_client_fsm:u_client|	383 (383)	231 (231)	0 (0)	0	0	0	0	0	0	0	0	152 (152)	59 (59)
5	|tftp_min_server:u_server|	152 (152)	53 (53)	0 (0)	0	0	0	0	0	0	0	0	99 (99)	2 (2)

Figure 34: Fitter resource list - before optimization

After optimization:

Fitter Resource Utilization by Entity

<<Filter>>

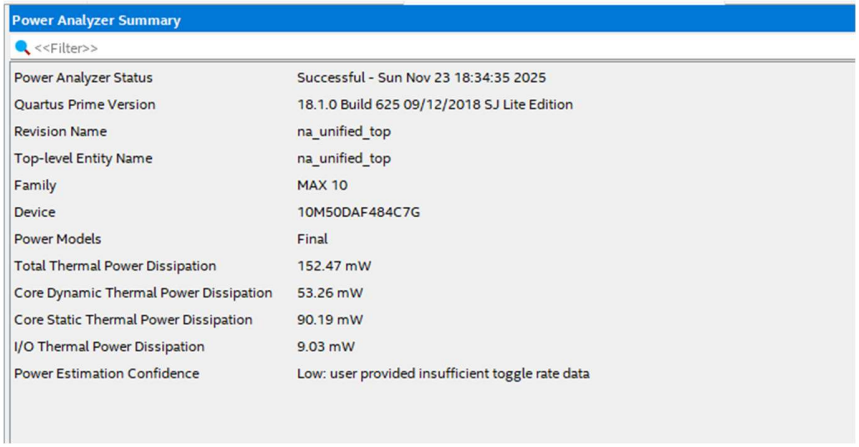
	Compilation Hierarchy Node	Logic Cells	Dedicated Logic Registers	I/O Registers	Memory Bits	M9Ks	UFM Blocks	DSP Elements	DSP 9x9	DSP 18x18	Pins	Virtual Pins	LUT-Only LCs	Register-Only
1	|na_unified_top	2381 (68)	1876 (25)	0 (0)	1090112	134	1	0	0	0	6	0	505 (43)	1060 (3)
1	|sld_hub:auto_hub|	146 (1)	91 (0)	0 (0)	0	0	0	0	0	0	0	0	55 (1)	18 (0)
2	|sld_signaltap:auto_signaltap_0|	1497 (140)	1381 (138)	0 (0)	1089536	133	0	0	0	0	0	0	116 (2)	968 (138)
3	|tftp_accel_top:u_accel|	148 (0)	95 (0)	0 (0)	576	1	0	0	0	0	0	0	53 (0)	22 (0)
4	|tftp_client_fsm:u_client|	371 (371)	231 (231)	0 (0)	0	0	0	0	0	0	0	0	140 (140)	47 (47)
5	|tftp_min_server:u_server|	152 (152)	53 (53)	0 (0)	0	0	0	0	0	0	0	0	98 (98)	2 (2)

Figure 35: Fitter resource list -after optimization

12.2 Power Optimization

Power optimization aims to reduce both static and dynamic power consumption. Static power is mainly determined by the device and configuration, while dynamic power is influenced by switching activity, clock rates, and logic transitions within the design. The following figures show this concept:

Before Optimization:



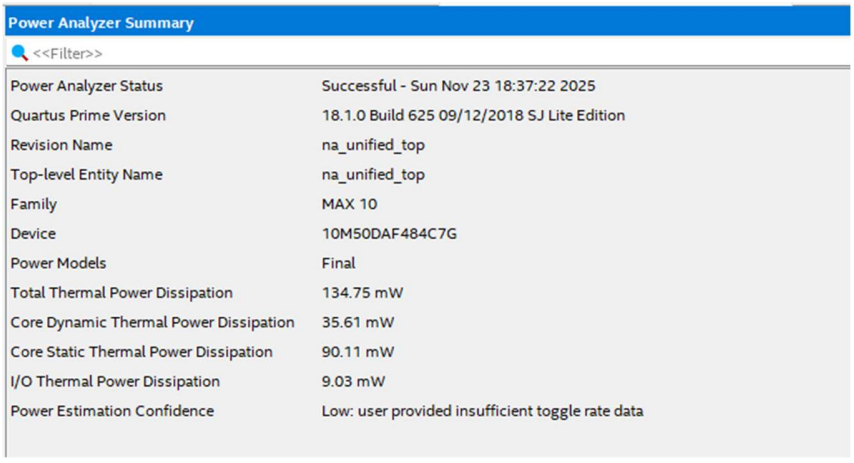
Power Analyzer Summary

<<Filter>>

Power Analyzer Status	Successful - Sun Nov 23 18:34:35 2025
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	na_unified_top
Top-level Entity Name	na_unified_top
Family	MAX 10
Device	10M50DAF484C7G
Power Models	Final
Total Thermal Power Dissipation	152.47 mW
Core Dynamic Thermal Power Dissipation	53.26 mW
Core Static Thermal Power Dissipation	90.19 mW
I/O Thermal Power Dissipation	9.03 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Figure 36: Power summary - before optimization

After Optimization:



Power Analyzer Summary

<<Filter>>

Power Analyzer Status	Successful - Sun Nov 23 18:37:22 2025
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	na_unified_top
Top-level Entity Name	na_unified_top
Family	MAX 10
Device	10M50DAF484C7G
Power Models	Final
Total Thermal Power Dissipation	134.75 mW
Core Dynamic Thermal Power Dissipation	35.61 mW
Core Static Thermal Power Dissipation	90.11 mW
I/O Thermal Power Dissipation	9.03 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Figure 37: Power summary - after optimization

13 FOOT PRINT

The following figures show a cutout from Quartus detailing the size of the entire project and how much space it takes up in the FPGA circuit. Closer observation shows that it utilizes approximately 65% of the total available memory space and approximately 2% of all available pins.

Flow Summary

<<Filter>>

Flow Status	Successful - Mon Nov 24 21:36:42 2025
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	na_unified_top
Top-level Entity Name	na_unified_top
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	2,409 / 49,760 (5 %)
Total registers	1876
Total pins	6 / 360 (2 %)
Total virtual pins	0
Total memory bits	1,090,112 / 1,677,312 (65 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 38: Foot print

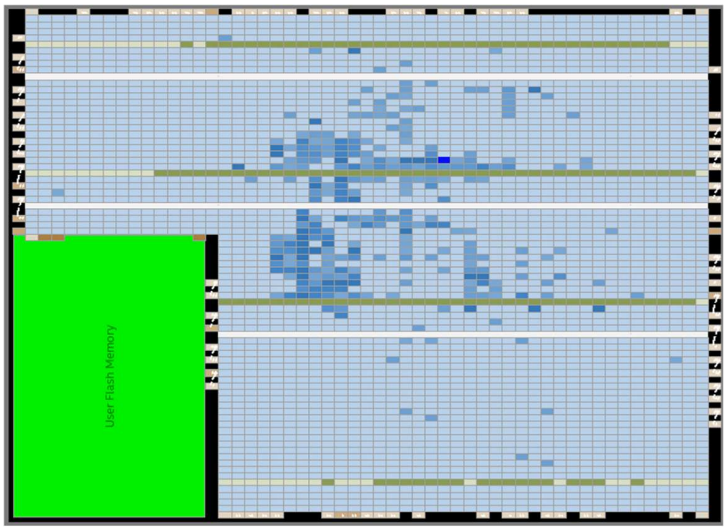


Figure 39: Chip Planner

14 ACTIVITY AND TIME PLAN

This chapter outlines the week-by-week activity plan for the development of the FPGA-based PXE/TFTP emulation system and hardware accelerator prototype. The plan describes the sequence of tasks, estimated effort, and time allocation needed to move from initial research and tool setup through module implementation, simulation, hardware testing, and final documentation. By dividing the work into clear and manageable phases, the activity plan ensures steady progress, allows time to address learning curves in VHDL, TFTP/PXE protocols, and FPGA debugging tools, and provides a structured framework for meeting project milestones within the allocated timeframe. The full plan is presented in the table and list below.

Table 10: Activity plan

Week	**Activity**	**Time (h)**
1 - 3	Background Research & Requirements Definition	20h/week
4 - 5	Scope redefinition and hardware constraints.	20h/week
6 - 11	Design, debugging and optimization	25h/week
12–13	Verification	25h/week
14–15	Validation	25h/week
16	Documentation & Final Report	25h/week
		Total: 375h

15 SERVA – OVERVIEW AND SHORT INTRODUCTION

SERVA is a small, portable program that can turn almost any Windows PC into a full PXE server with just a few clicks. In practical terms, SERVA lets other devices on the network boot without using their own storage. Instead, it sends out the files they need—such as bootloaders, kernels, or installation images—using well-known protocols like DHCP and TFTP.

What makes SERVA popular is its simplicity. There is no installation, no complicated setup, and no need for several different servers running at once. You simply start the program, point it to the folders containing your boot files, and it automatically prepares everything needed for PXE booting. For Windows deployments, it can even unpack the installation media and organize all required files automatically. This makes it a go-to tool for technicians who need a quick and reliable way to install or boot multiple systems over the network.

For Linux or custom firmware, SERVA works just as well. It can send kernel and initrd images, serve configuration files, and handle most common PXE setups without requiring deep knowledge of network services. Its portability also makes it useful as a temporary troubleshooting tool: it can run from a USB stick and provide DHCP, TFTP, HTTP, or FTP services with almost no configuration.

In short, SERVA acts as an “all-in-one PXE box” that hides the complexity of network booting. This is exactly why it plays an important role in this thesis: once the FPGA design is extended with real Ethernet support, it could connect directly to a SERVA PC and participate in a real PXE boot sequence. In this current revision, the DE10-Lite operates entirely internally without any real Ethernet hardware. The long-term vision is to use SERVA as the front-end server and let the FPGA act as a hardware accelerator—speeding up TFTP transfers, reducing boot times, and enabling storage-less devices such as speed-enforcement cameras to boot reliably over the network.

16 RESULTS AND FINAL CONCLUSION

The goal of this project was to recreate the basic steps of a PXE-style boot process inside an FPGA and explore whether parts of this flow could be accelerated in hardware. Even though the DE10-Lite board has no Ethernet hardware at all, the design still manages to imitate a small TFTP client, a simple TFTP server, and a real-time packet parser — all talking to each other inside the FPGA as if they were on a real network.

The result is a complete, self-contained model of a PXE-style RRQ → DATA → ACK exchange. Every byte, header, and state change is visible and happens with deterministic, clock-accurate timing. This makes it much easier to understand how PXE/TFTP works under the hood compared to watching packets fly across a real network.

The project did not include real file-transfer payloads. The ROM is in place and fully functional, but after testing it became clear that synchronizing payload reads with the DATA frame generator would require more time than was available. In early attempts, timing mismatches between when the Mini-Server expected data and when the ROM produced it caused misalignment. Rather than deliver a half-working file transfer, the decision was made to use clean, header-only frames with a dummy TFTP header. This kept the rest of the system stable and made the protocol much easier to validate.

Verification was done using both ModelSim and SignalTap. ModelSim gave perfect visibility, but SignalTap ran into a practical hardware limitation: the DE10-Lite board can only handle 8k samples of built-in capture memory. To show the full “transfer → reset → next transfer” sequence, at least 18–32k samples were needed. When trying to increase the depth, the system overflowed and corrupted the capture. Because of this, some of the more complex sequences had to be shown in simulation instead of on hardware.

Despite these limitations, the core system works exactly as intended. The accelerator successfully parses outgoing packets, tracks IP/UDP/TFTP headers, and maintains counters in real time. The client FSM and server FSM behave deterministically, and all timing results show that the design comfortably meets the 50 MHz target frequency.

In the end, the project achieves what it set out to do: create a clear, controlled FPGA model of a PXE-style network boot exchange, while laying the groundwork for a practical hardware accelerator that could eventually be used in real systems — including the original goal of a centralized PXE solution for speed-enforcement cameras, removing the need for fragile micro-SD cards in every unit.

17 FUTURE WORK AND DEVELOPMENT PATH

The current design forms a solid foundation, but there are several realistic improvements that would make it far more capable and closer to a true PXE/TFTP system.

i) **Real Payload Transfer:** Right now, DATA packets only contain headers and an empty TFTP header. The next logical step is to:

- Read the ROM in sync with the Mini-Server
- Insert 0–512 bytes of real data into each DATA packet
- Handle the final, shorter block
- Store received data on the client side

This was originally planned, but timing and synchronization issues made it too risky to rush. With more time, this can be solved.

ii) **Real TFTP Logic (RRQ Parsing, Retries, Timeouts):** The Mini-Server currently ignores incoming requests and always sends its own DATA blocks. A full version would:

- Parse the RRQ properly
- Track session state
- Support retries and timeouts
- Handle error codes
- Match real TFTP behavior
- This would allow the FPGA to talk to real TFTP clients and servers.

iii) **Smarter Hardware Acceleration:** The accelerator now analyzes packets but doesn't interfere with them. Future steps could include:

- Automatically generating ACKs without the client FSM
- Checking header correctness
- Detecting mismatched block numbers
- Measuring transfer speed or latency
- Inspecting payload content

This pushes the FPGA closer to a real hardware offload engine.

iv) **Connect to an Actual Network (MAC/PHY Integration):** The biggest limitation today is the lack of Ethernet hardware on the DE10-Lite. Adding an external MAC/PHY (via expansion board) or using a board with built-in Ethernet would allow:

- Direct communication with Serva

- Real PXE boot sequences
- Real transfer timings
- Compatibility with existing IT tools

This was the original goal: a user-friendly hardware accelerator that plugs into a PXE environment and speeds up booting for embedded systems.

v) **Real-World Use Case in Speed Enforcement Systems:** The long-term idea behind this project is to eliminate the need for local storage (like micro-SD cards) in speed cameras. Instead of booting from slow or failure-prone flash, each device could:

- Boot over the network
- Load firmware quickly and securely
- Reduce maintenance
- Improve reliability

The FPGA-based accelerator could drastically speed this up by handling the “slow parts” of TFTP and PXE in hardware, providing a centralized solution for several systems simultaneously, handling rebooting in a reliable manner.

vi) **Better Debugging Tools and System Visibility:** SignalTap limitations made some tests hard to show. Future improvements could be:

- Using boards with larger capture memories
- Streaming debug data over UART or USB
- Adding custom on-chip scopes
- Logging traffic into internal RAM for later readout

This would remove the sampling-depth problems seen during this project.

18 ADDITIONAL PROJECT RESOURCES

To support the understanding and reproducibility of this work, all project files and additional materials are publicly available. This includes the complete VHDL source code, simulation testbenches, SignalTap captures, diagrams, and the final 10-minute video presentation summarizing the architecture and results of the project.

GitHub Repository

The full project, including all hardware modules and documentation, is available at:

https://github.com/fredd1975/Examensarbete_FPGA_TEIS

The repository is organized into folders for:

- Source code (VHDL)
- ModelSim simulation environment
- SignalTap setup files
- Figures and diagrams used in the thesis
- The final presentation material

Video Presentation

A short 10-minute video summarizing the design goals, architecture, implementation, and validation results accompanies this thesis.

https://www.youtube.com/watch?v=4qyvS_eM6Wo

19 REFERENCES

Wikipedia – Preboot Execution Environment (Sept.05, 2025):
https://en.wikipedia.org/wiki/Preboot_Execution_Environment

Heimdal Security – What is PXE Boot and How Does it Work (Sept.07, 2025):
https://heimdalsecurity.com/blog/what-is-pxe-boot

GeeksforGeeks – Introduction to Basic Networking Terminology (Sept.12, 2025):
https://www.geeksforgeeks.org/computer-networks/introduction-to-basic-networking-terminology

Terms you Need to Know in Networking (Sept.14, 2025):
https://www.youtube.com/watch?v=kn7ei2ENJbI

Vercot – Serva PXE Server for Windows (Sept.21, 2025):
https://www.vercot.com/~serva

Serva Configuration for PXE Network Boot (Sept.22, 2025):
https://www.youtube.com/watch?v=w6RBPFcwUME&t=347s

Chat GPT - AI helper (Sept. 25, 2025): *https://chatgpt.com*

FPGA Implementation of IP Packet Header Parsing Hardware (Sept.28, 2025):
https://icaiit.org/proceedings/5th_ICAIIT/S1_5_Efnusheva.pdf?utm_source=chatgpt.com

ThalesGroup – UDP Offload Engine (Oct.09, 2025):
https://github.com/ThalesGroup/udp-offload-engine?utm_source=chatgpt.com

Simgunz – UDP IP Stack (Oct.12, 2025):
https://github.com/simgunz/udp_ip_stack?utm_source=chatgpt.com

Zynq 7000 TFTP Server (Oct.15, 2025):
https://github.com/efetunca/Zynq-7000-TFTP-Server/?utm_source=chatgpt.com

20 REVISION HISTORY

Revision	Comments	Responsible	Date
1.0	First release	Freddy A.	2025-11-27

21 APPENDIX A: RTL STRUCTURE

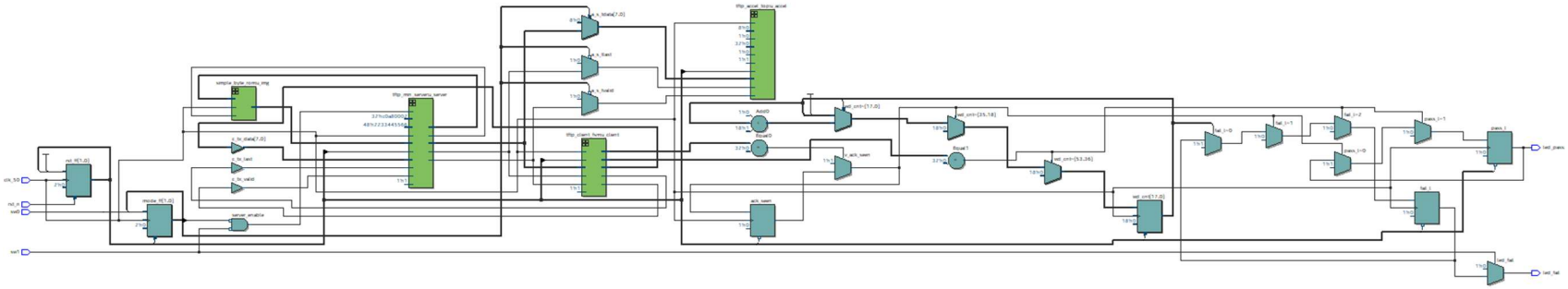


Figure 40: RTL view

22 APPENDIX B: TEST PROTOCOL

Case	Input description	Acceptance description	Signals affected/ expected values	Verification (Simulator)	Validation (Board)
1	rst_n activated (low → high)	System resets to IDLE, no traffic, counters cleared, LEDs off	**Traffic:** **_tx_data* idle. **Counters:** *stat_rrq_c = 0, stat_data_c = 0, stat_ack_c = 0.* **FSM:** *tx_st = TX_IDLE*, *parser_st = S_IDLE*. **LED:** *led_pass = led_fail* =0.	OK	OK
2	SW0=0 (normal mode)	Client and Server both active, accelerator receives data from server	**Mode:** *SW0 = 0.* **Traffic:** **_tx_data* starts sending in bursts. **Counters:** *stat_rrq_c = 1* and stays at that value, *stat_data_c* increments, *stat_ack_c* increments. **FSM:** *parser_st* on each DATA frame: *S_IDLE -> S_ETH -> S_IP -> S_UDP -> S_TFTP.*	OK	OK
3	SW0=1 (accelerator-only mode)	Client and Server held in reset, only accelerator runs	**Mode:** *SW0 = 1.* **Traffic:** **_tx_data = 0* (no bursts). **Counters:** *stat_rrq_c, stat_data_c* and *stat_ack_c* do not increment. **FSM:** *tx_st = TX_IDLE*, *parser_st = S_IDLE*. **LED:** *led_pass = 0.*	OK	OK
4	Start in SW0=0	Client sends RRQ	**Mode:** *SW0 = 0.* **Reset:** *rst_n = 1.* **Traffic:** *c_tx_data* comes in short bursts (RRQ bytes). **Counters:** *stat_rrq_c and stat_data_c* increments from 0 to 1, *stat_ack_c = 0.* **FSM:** *tx_st* cycles (*TX_IDLE -> TX_RRQ_ETH -> TX_RRQ_IP -> TX_RRQ_UDP -> TX_RRQ_TFTP -> TX_EOF)*.	OK	OK

5	Server responds with first DATA	DATA frame detected	**Traffic:** *s_tx_data* sends data. **Counters:** *stat_rrq_c = 1*, *stat_data_c* increments, *stat_ack_c = 0.* **FSM:** *parser_st* cycles (*S_IDLE -> S_ETH -> S_IP -> S_UDP -> S_TFTP*).	OK	OK
6	Client receives DATA block	Client sends ACK	**Traffic:** *s_ack = 1* right after *stat_ack_c* increments to 1. **Counters:** *stat_ack_c* increments from 0 to 1, *stat_data_c* increments. **FSM:** *tx_st* cycles (*TX_IDLE -> TX_ACK_ETH -> TX_ACK_IP -> TX_ACK_UDP -> TX_ACK_TFTP -> TX_EOF*). **LED:** *led_pass = 1* after first ACK (*ack_seen = 1* and *stat_ack_c /= 0*)	OK	OK
7	Transfer reaches G_MAX_BLOCKS	Session ends	**Traffic:** *s_tx_data* stops sending data. **Counters: :** *stat_rrq_c = 1*, *stat_data_c = G_MAX_BLOCKS (=8)*, *stat_ack_c* incrementing until *c_tx_data* stops sending. **FSM:** *parser_st* returns to *S_IDLE* and stays there. **LED:** *led_pass = 1.*	OK	OK
8	No incoming DATA	Client stays idle (same as case 3)	**Mode:** *SW0 = 1.* **Traffic:** **_tx_data = 0* (no bursts). **Counters:** *stat_rrq_c = 0, stat_data_c = 0, stat_ack_c = 0* (no increment). **FSM:** *tx_st = TX_IDLE*, *parser_st = S_IDLE*. **LED:** *led_pass = 0.*	OK	OK
9	rst_n asserted during transfer	Clean restart	**A - On Reset:** - **Traffic:** **_tx_data* idle. - **Counters:** *stat_rrq_c = 0, stat_data_c = 0, stat_ack_c = 0.* - **FSM:** *parser_st = S_IDLE*, *tx_st = TX_IDLE*.	OK	OK

			- **LED:** *led_pass* = 0 during reset. - **Reset:** *rst_n = 0.* **B - After releasing reset:** - **Mode:** *SW0 = 0* - **Traffic:** c_*tx_data* shows RRQ data, *s_tx_data* shows TFTP data. - **Counters:** *stat_rrq_c* increments to 1*,* *stat_data_c* increments on every data block, *stat_ack_c* increments on every ACK. - **FSM:** *parser_st* cycles (*S_IDLE -> S_ETH -> S_IP -> S_UDP -> S_TFTP*), *tx_st* cycles (*TX_IDLE -> TX_RRQ_ETH -> TX_RRQ_IP -> TX_RRQ_UDP -> TX_RRQ_TFTP -> TX_EOF)*. - **LED:** *led_pass* = 1 on first ACK*, led_fail = 0.*		
10	Set SW0=0 and SW1=1 and reset shortly after	Watchdog timeout – client never receives DATA/ACK because server is disabled	**Traffic:** *ack_seen* stays at 0. **Counters:** *stat_rrq_c = 1, stat_ack_c = 0, wd_cnt* starts counting. **LED:** *led_pass* = 0, *led_fail = 1*.	OK	OK