POLITECNICO MILANO 1863

FPGA-Based Interface for a High-Speed Custom Chip: Design and Implementation Report

Simone Ranfoni - 10934656

May 2025

Abstract

This report details the design and implementation of a VHDL-based FPGA system. The primary goal of this project is to create a robust interface for a custom chip developed by the Electronics Department of Politecnico di Milano. The chip generates data at a high frequency (500 MHz), buffers it in an integrated SRAM, and streams it out via a PSI (Parallel Stream Interface) at a lower, configurable frequency (up to 100 MHz). The FPGA system is responsible for configuring the chip via SPI (Serial Peripheral Interface) and receiving the high-throughput PSI data, which is then packetized and transmitted over Ethernet. This document covers the project's objectives, architectural design, implementation choices, experimental validation, and conclusions.

Contents

1 Introduction

The core objective of this project is to develop an FPGA solution capable of reliably interacting with a custom high-speed data acquisition chip. This chip's unique characteristic is its internal 500 MHz data generation, far too fast for direct external streaming. It employs an 1152 kbit SRAM to capture data segments ("chunks") which are then streamed out via the PSI at a more manageable frequency (up to 100 MHz).

1.1 Project Goals

The main goals for the FPGA HDL code were:

- **SPI Writer:**
  - Soft-programmable address and data bit lengths.
  - Soft-programmable clock frequency.
- **PSI Reader via Ethernet:**
  - Data throughput up to 100 Mbps.
  - Capability to capture data with variable length.
- **Reliability:** The overall transmission stack must be robust and avoid random disconnections, a critical improvement over previous setups.
- **Data processing:** Providing a Python/Matlab-based interface capable of unpacking Ethernet data originating from the FPGA, interpreting it into words, and plotting these to visualize the results.

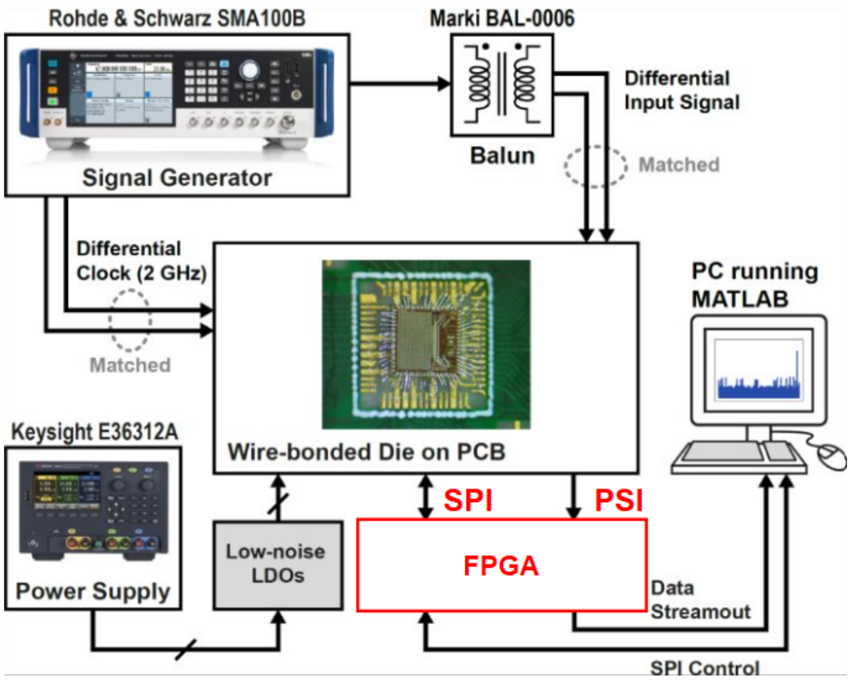


Figure 1: Project overview diagram

1.2 System Overview and Utilized Entities

To achieve these goals, a modular VHDL design was implemented, comprising several key entities:

- **`TOP_entity`:** The main top-level module that instantiates and connects all other components and IP cores. It also includes the test data generation logic for PSI.
- **`test_DUT`:** A mock-up of the custom chip, developed to facilitate testing without the physical chip. It simulates SPI command reception and PSI data stream initiation.
- **`AXI_interface_PSI`:** The core logic for handling the incoming PSI data stream, buffering it, and interfacing with the AXI Ethernet Lite IP for L2 packet transmission.
- **`SPI_interface`:** Manages the bit-by-bit transmission of SPI commands to the (mock-up) chip.
- **`axi_ethernetlite_0`:** A Xilinx IP core providing MAC functionality for Ethernet communication.
- **`clk_wiz_0`:** A Xilinx IP core for generating the necessary clock signals from a system clock.
- **`fifo_generator_0`:** A Xilinx IP core for creating a FIFO, used for clock domain crossing and data buffering between PSI input and AXI read logic.

1.3 Adherence to Specifications

The design aimed to meet the specified data throughput of up to 100 Mbps for the PSI interface. Key aspects such as soft-programmability for SPI and handling variable-length data were incorporated. The reliability of the Ethernet communication was a paramount concern, addressed through careful buffer management and robust AXI interfacing.

2 Design Choices

Several critical design choices were made to ensure functionality, testability, and performance.

2.1 Mock-up DUT (`test_DUT`)

Given the absence of the physical custom chip during development, a VHDL mock-up named `test_DUT` was created.

- **SPI Command Reception:** The `DUT_spi_process` within this entity is designed to receive serialized SPI data. For the current test setup, it specifically checks if the received SPI address is '50' (decimal).
- **PSI Stream Trigger:** Upon detecting SPI address 50, the `test_DUT` asserts a signal (`start_psi`) that emulates the custom chip starting its PSI data transmission. This signal is then used by the `TOP_entity` to initiate the data flow into the `AXI_interface_PSI` module.
- **PSI State Emulation:** The `DUT_psi_process` changes its internal state upon the `start_psi` trigger, signaling that the PSI data stream has commenced.

2.2 PSI Data Handling and Ethernet Packetization (`AXI_interface_PSI`)

The `AXI_interface_PSI` module is central to receiving PSI data and preparing it for Ethernet transmission, communicating with the AXI bus of the IP core "AXI Ethernet Lite". Therefore, module incorporates two main core processes. The **PSI process** manages the data received from the DUT: its primary objective is to ensure no data loss while maintaining continuous operation without incurring delays for data processing. The **AXI process** coordinates the writing phase in the AXI bus, guaranteeing that the psi data is correctly stored in the AXI Ethernet Lite registers. It is also responsible for configuring the MAC addresses (source and destination), the Type/Length ethernet packet field, and other configuration commands within the AXI registers.

To be sure that everything worked as expected, some important design choices were made, such as:

- **Dual Input Buffering:** To ensure no PSI data is lost due to processing latencies, a dual-32 bit-buffer mechanism is implemented for incoming PSI data within the `data` process. As data arrives on `psi_data_i`, it fills `data_buf1`. Once `data_buf1` is full (32 bits), its content is written to a FIFO, and `data_buf2` starts filling. This alternation continues, ensuring continuous data capture.
- **FIFO for Clock Domain Crossing:** A FIFO (instantiated from `fifo_generator_0`) is used to safely transfer the 32-bit data words from the PSI clock domain (`psi_clk_i`) to the AXI clock domain (`axi_clk_i`). In fact, these clocks aren't meant to be at the same frequency: the AXI clock is stable at 100MHz, while the PSI clock is variable; in this test case, it could be up to 100MHz. The write enable for the FIFO (`wr_en`) is asserted when either `data_buf1` or `data_buf2` is full.
- **AXI State Machine (`axi_process`):** An AXI state machine (`axi_state_t` with states like `IDLE`, `WRITE_ADDR`, `WRITE_DATA`, `WRITE_RESP`) manages the AXI4-Lite transactions with the Ethernet Lite IP. This protocol exposes ports that facilitate specific read and write operations, which must be executed in precise sequences.
  - It initializes Ethernet IP registers with MAC addresses, packet length, and other header information during an initial setup phase. In this way we ensure to configure immediately the ethernet packet headers, to avoid writing them in every packet.

  - For data transmission, it reads data from the FIFO (`dout`) when the FIFO is not empty (`empty = '0'`) and writes it to the Ethernet IP's transmit buffer. The `rd_en` signal for the FIFO is controlled accordingly.

- **Ethernet Packet Finalization and Timeout:** To handle scenarios where the PSI data stream ends, pauses or doesn't match perfectly the last packet's size, a timeout mechanism is implemented. If the FIFO remains empty (`empty = '1'`) for a specified number of AXI clock cycles (in this test, `empty_counter >= 127`) after data transmission has started (`started = '1'`), the system assumes the current data chunk is complete. It then proceeds to write a control word to the Ethernet IP to finalize and send the current packet. This ensures that partially filled packets are sent promptly. The `packet_data_length` generic defines the target data payload size, to which the AXI bus writing algorithm is adapted. This design makes the algorithm soft-programmable based on the current transmission's properties.

- **Endianness:** The AXI Ethernet Lite communicates via its (`tx_data`) 4-bit bus changing the order of the Bytes stored into the AXI TX registers. To ensure that the output packets sent over the LAN contain the correct data, the writing order of the 32-bit bus was accordingly inverted. This guarantees that a properly formed packet is received, ready to be segmented into the n-bit words coming from the chip.

2.3 SPI Command Generation (`SPI_interface`)

The `SPI_interface` module is responsible for sending commands to the DUT.

- **Serial Data Transmission:** The `ioINTERFACE_spi` process serializes the address (`addr_i`) and data (`data_i`) bit by bit onto the `spi_data_o` line, synchronized with `spi_clk_i`.

- **Programmable Lengths:** Generic parameters `nb_spi_addr` and `nb_spi_data` allow for flexible configuration of the SPI command structure.

- **Load Enable Control:** A load enable signal (`spi_le_i`) is asserted after all address and data bits have been transmitted. This signal is useful for the DUT to understand when the SPI stream is over and when it can finally interpret the command sent.

2.4 Test Data Generation

For testing the PSI data path, a sine wave generator is implemented within the `TOP_entity`.

- **Lookup Table (LUT):** A 64-entry LUT (`sine_table`) stores 16-bit samples of a quarter-period sine wave.

- **Scanning Logic:** The `matlab_file_process` reads these samples sequentially. It scans the table forward (index 0 to 63), then backward (63 to 0). After one full cycle (forward and backward), the sign of the output is inverted (`sign` signal), and the process repeats. The whole procedure is done for a total of 4 cycles, generating a total of 4 periods of the sine wave. These samples simulate the hypothetical content of an SRAM. In a real-world scenario, this data would have been streamed from the DUT via the PSI protocol.

- **Bit-Serial Output:** Each 16-bit sample from the LUT is output bit by bit on the `matlab` signal, synchronized with `psi_clk_pll`. This `matlab` signal serves as the `psi_data_i` for the `AXI_interface_PSI` module during testing. The process stops generating new sine data after 4 complete cycles (`cycle_count >= 4`).

3 Architecture

The overall system architecture is hierarchical, with the `TOP_entity` integrating various custom modules and Xilinx IP cores.

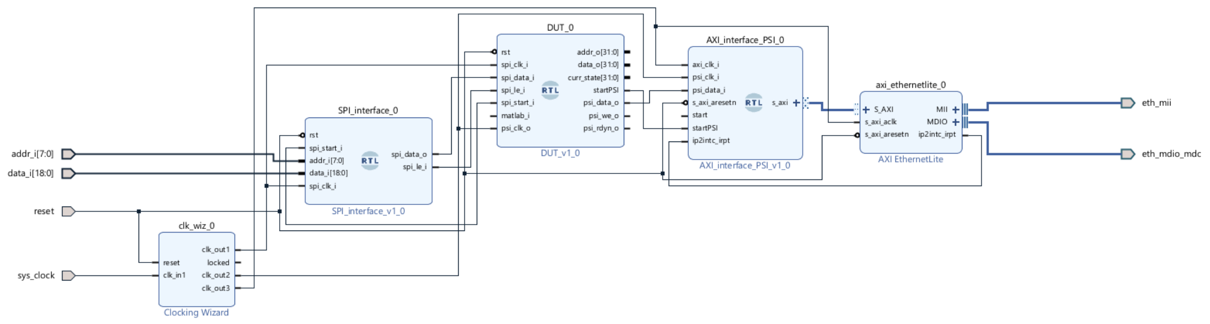


Figure 2: Vivado Block Diagram of the FPGA System.

3.1 `TOP_entity`

This module serves as the main structural component.

- **Clock Generation:** It instantiates `clk_wiz_0` to generate the required clock signals: `axi_clk` for the AXI domain, `spi_clk` for the SPI interface, and `psi_clk_pll` for the PSI data input and mock DUT operation. The `eth_ref_clk_out` is also generated here. This clock reference is essential for the FPGA's PHY component, which handles the crucial communication between the AXI Ethernet Lite IP and the RJ45 connector (the standard Ethernet port).
- **Module Instantiation:** It instantiates AXI_interface_PSI, axi_ethernetlite_0, SPI_interface, and test_DUT. All the interconnections between entities are carefully treated, ensuring that all components receive the correct clocks, resets, and input/output data ports (both for PSI and SPI). The PHY ports are directly mapped with the physical pins of the FPGA, through a constraints file.
- **Reset Handling:** It provides the necessary reset signals (e.g., `not_rst`) to the sub-modules. In the current test setup, there are two distinct reset mechanisms: the primary reset affects all components except the AXI Ethernet Lite IP. The AXI Ethernet Lite IP's reset is managed separately because resetting it repeatedly would force it to re-establish handshakes with the PC, complicating the testing process.

4 Implementation Details

4.1 AXI Ethernet Lite IP Configuration and Performance

In the final test, the `axi_ethernetlite_0` IP is configured to transmit Layer 2 Ethernet frames.

- **Packet Assembly:** The `AXI_interface_PSI` module directly writes to the AXI registers of the Ethernet IP. This includes:
  - Destination MAC Address (e.g., `s_axi_wdata <= x"FFFFFFFF"` at address "0000000000000" and part of `s_axi_wdata <= x"0000FFFF"`; broadcast, to facilitate packet reception).
  - Source MAC Address (e.g., part of `s_axi_wdata <= x"0000FFFF"` at address `"0000000000100"` and `s_axi_wdata <= x"CEFA005E"` at address `"0000000001000"`).
  - Ethernet Type/Length field (e.g., `s_axi_wdata <= std_logic_vector(to_unsigned(0, 32))`, here set to 0 to easily detect type II frames). The code later writes the total length of data written to a control register (e.g. `s_axi_wdata <= std_logic_vector(to_unsigned(packet_data_length + 16, 32))` to address "0011111110100").
  - Payload data from the FIFO (`s_axi_wdata <= dout(...)`).
  - Finally, a control word is written into a specific register to initiate the IP's transmission of the data residing in the recently written registers.
- **Dual Transmit Buffer Utilization (Ping-Pong):** The `AXI_interface_PSI` module makes use of the Ethernet IP's dual transmit buffer capability. It alternates the base address for writing packet data using the `curraddr` signal. This signal is toggled between a "ping" address (e.g., starting "0000000010000") and a "pong" address (e.g., starting "0100000010000") using the `ping_pong` signal. This allows one buffer in the Ethernet IP to be filled by the FPGA logic while the other is potentially transmitting, maximizing throughput. This choice was crucial because, without it, significant time would have been wasted waiting for transmissions to complete, not only to start a new one but also to start writing the next packet in the buffer. This would have resulted in half the overall transmission speed and necessitated a substantial data structure to store the PSI data continuously streamed from the chip.
- **Achieved Throughput:** This efficient utilization of the Ethernet IP's buffering, coupled with fast data feeding from the `AXI_interface_PSI`, enabled an average data throughput of 98 Mbps in "freerun" mode, very close to the 100 Mbps theoretical maximum of the AXI Ethernet Lite operating with a 100 MHz AXI clock.

4.2 FIFO Utilization

The `fifo_generator_0` IP plays a crucial role:

- **Clock Domain Crossing (CDC):** It safely transfers data from the PSI clock domain (`psi_clk_i` for writes) to the AXI clock domain (`axi_clk_i` for reads). This process is critical for avoiding critical path overloads and ensuring a high Worst Negative Slack (WNS).
- **Buffering and Decoupling:** It decouples the PSI data arrival rate from the AXI read rate, providing elasticity.

- **Operational Observation:** In practice, the FIFO remains nearly empty. This is because the AXI clock (100 MHz) is equal to or faster than the PSI clock (tested at 95 MHz, max 100 MHz). The `AXI_interface_PSI` reads data from the FIFO (`rd_en <= '1'`) as soon as it's available (`empty = '0'`) and the AXI state machine is ready to process it. This low latency in FIFO passthrough is beneficial for overall system responsiveness. Looking at the simulations, it's easily observable that during the writing phase into the AXI registers, the FIFO stores a maximum of a single word. The only moment it briefly holds more data is at the beginning, during the Ethernet headers configuration phase. Here, the AXI process writes data not originating from the PSI, resulting in a small number of 32-bit entries being stored temporarily. It's also clear that after a few nanoseconds, the FIFO enters this pseudo-empty state.

4.3 Ethernet Packet Length and Timeout

- For the final test, the `packet_data_length` generic in `AXI_interface_PSI` is set to 1400 bytes. This means the system aims to create Ethernet frames with a 1400-byte payload.
- The `axi_process` fills the Ethernet IP's transmit buffer. The payload writing loop continues as long as `counter <= (packet_data_length/4 + 10)` and the FIFO is not empty.
- **Timeout for Last Packet:** If the PSI data stream (simulated by the sine wave generator) finishes before a 1400-byte packet is completely filled, the system must not wait indefinitely. The `empty_counter` in `AXI_interface_PSI` monitors how many `axi_clk_i` cycles the FIFO has been empty while a packet transmission was in progress (`started = '1'`). If `empty_counter` reaches 127 (`0x7F`), it indicates a timeout. The system then forces the `counter` to advance past the data filling stage (`counter <= (packet_data_length/4 + 11)`) and proceeds to send the control word that finalizes and transmits the current (partially filled) packet. In the final test, the total byte dimension of the streamed PSI data was intentionally set to be larger than a single packet but smaller than two. This allowed for a clear observation of the timeout behavior in action.

5 Experimental Results

The functionality of the entire system was validated using a Python script (`sniff.py`) that captures and analyzes the Ethernet packets transmitted by the FPGA.

5.1 Python Sniffing Script (`sniff.py`)

The script performs the following actions:

- Uses the Scapy library to sniff network packets on a specified interface (`INTERFACE_NAME`).
- Filters packets based on the FPGA's source MAC address (in this test, `TARGET_MAC_ADDRESS = "00:00:5e:00:fa:ce"`).
- Captures a predefined number of packets (`PACKETS_TO_CAPTURE = 2`).
- For each captured packet (expected to be IEEE 802.3 or Ethernet II), it extracts the raw payload.
- The payload is then parsed as a sequence of 16-bit signed integers (big-endian, `'>h'`).
- These integers, representing the sine wave samples, are collected.
- Finally, the script plots the collected samples using Matplotlib.

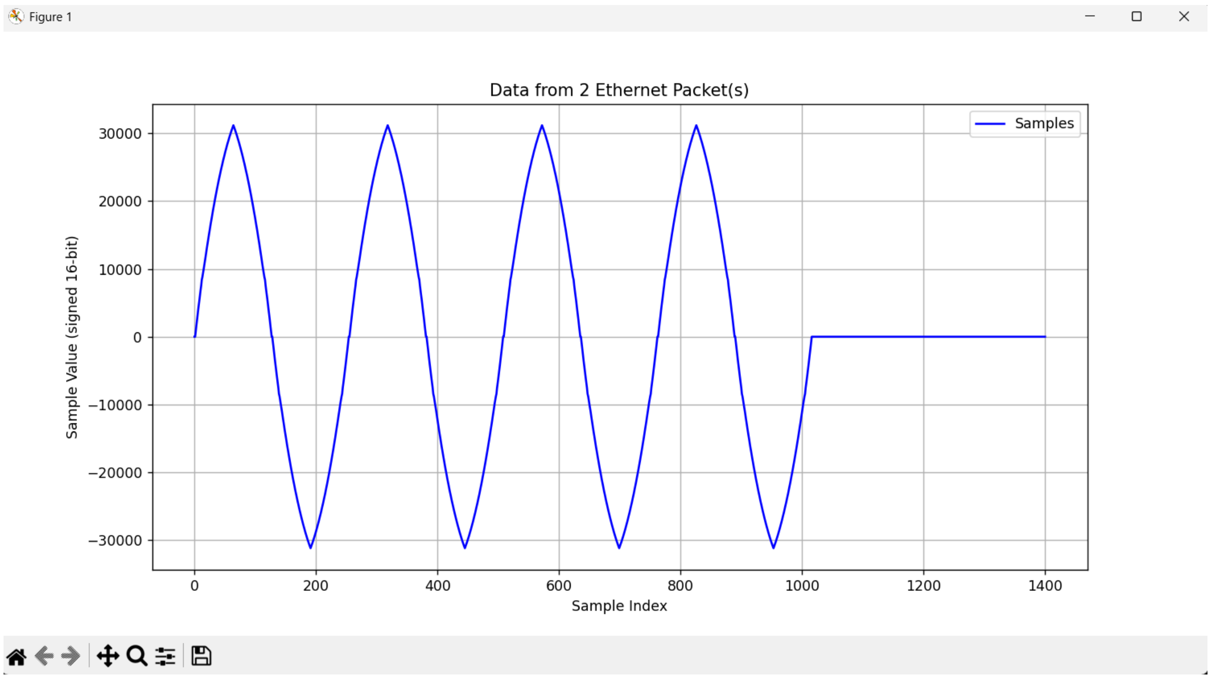
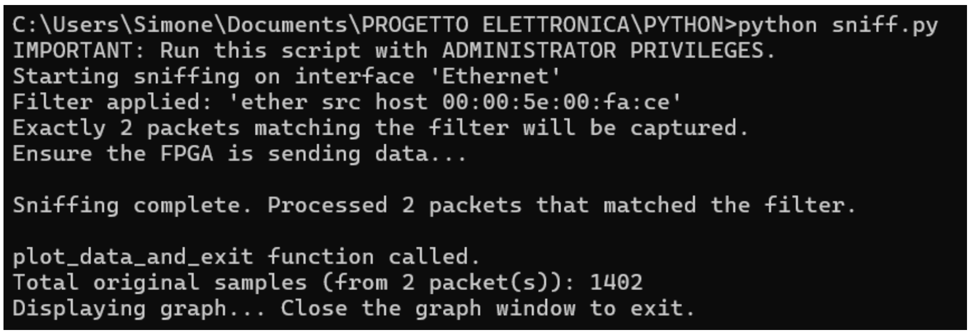


Figure 3: Final plot of the sinusoid.

5.2 Interpretation of Test Results

- **Sine Wave Verification:** The plot generated by the Python script successfully displayed four complete periods of a sine wave. This confirms the correct end-to-end transmission:
  1. Generation of sine wave data by the `TOP_entity`'s LUT and scanning logic.
  2. Correct capture of this bit-serial PSI data by `AXI_interface_PSI`'s dual input buffers.

3. Proper functioning of the FIFO for clock domain crossing (CDC) and buffering.
4. Accurate assembly of Ethernet packet headers (MAC addresses, length/type, ...) by `AXI_interface_PSI`.
5. Correct insertion of the sine wave data (after being read from FIFO and byte-swapped) into the payload of the Ethernet packets.
6. Successful transmission by the `axi_ethernetlite_0` IP.
7. Accurate reception and parsing by the Python script.



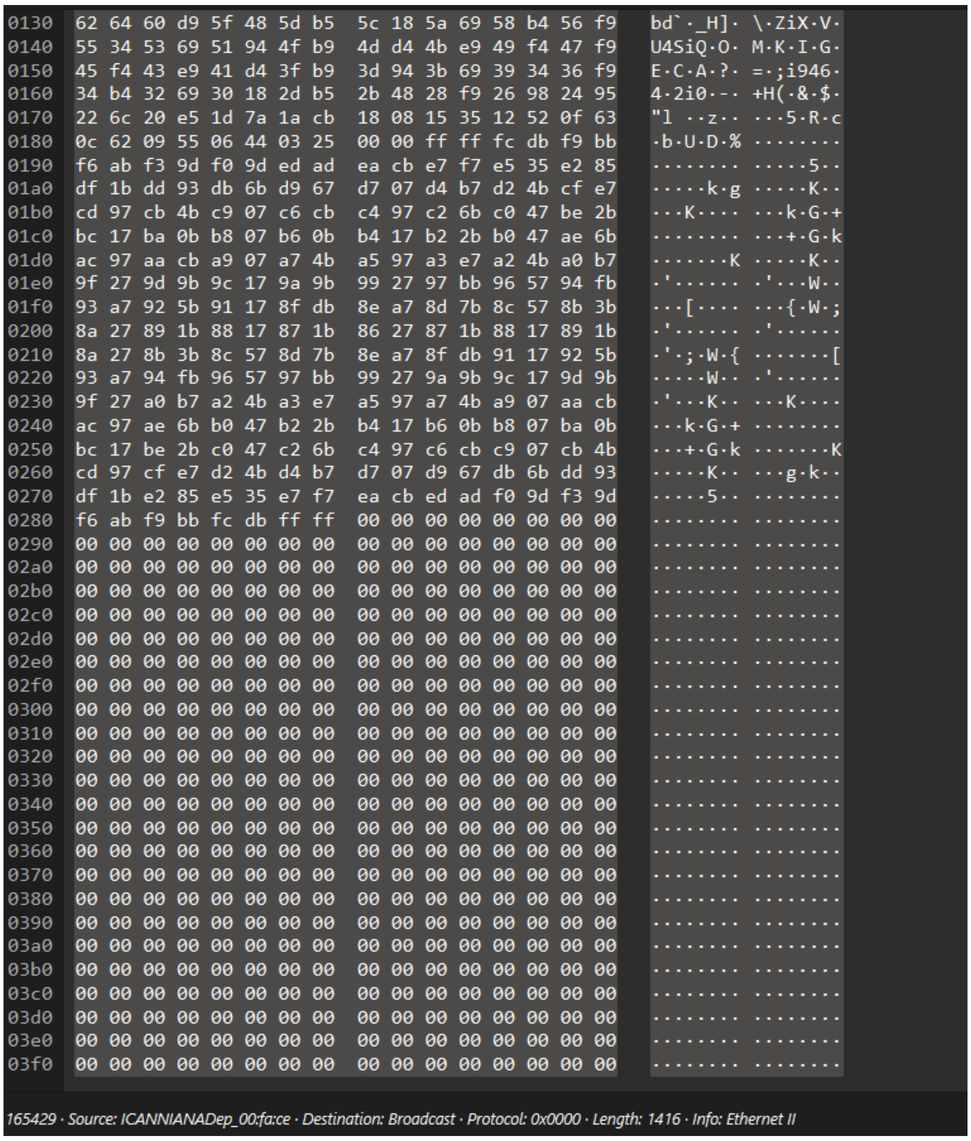
```
C:\Users\Simone\Documents\PROGETTO ELETTRONICA\PYTHON>python sniff.py
IMPORTANT: Run this script with ADMINISTRATOR PRIVILEGES.
Starting sniffing on interface 'Ethernet'
Filter applied: 'ether src host 00:00:5e:00:fa:ce'
Exactly 2 packets matching the filter will be captured.
Ensure the FPGA is sending data...

Sniffing complete. Processed 2 packets that matched the filter.

plot_data_and_exit function called.
Total original samples (from 2 packet(s)): 1402
Displaying graph... Close the graph window to exit.
```

Figure 4: Python script execution in Command Prompt.

- **Packetization and Timeout Confirmation:** The LUT contains 64 samples, representing one-quarter of the sinusoid's period. The data process streams the entire LUT in a specific sequence: forwards, then backwards, then forwards with an inverted sign, and finally, backwards with an inverted sign. This entire four-step sequence is repeated four times, resulting in a total of four complete sine wave periods. Therefore, every period contains 64 samples * 4 quarter periods, for a total of 256 samples/period. The sine wave data generated (4 periods * 256 samples/period * 2 bytes/sample = 2048 bytes) is less than the payload of two full 1400-byte Ethernet packets (2800 bytes) but more than one (1400 bytes). The test involved capturing two packets.
  - The first packet would be filled with the initial part of the sine wave data.
  - The second packet would contain the remaining sine wave data. Since the sine wave data concludes, the remaining part of the packet will contain a sequence of '0'. The fact that a second packet is sent, containing the tail end of the sine wave followed by zeros, confirms that the timeout mechanism correctly finalized and triggered the transmission of this last packet. The plot showing the sine wave followed by zeros validates this behavior.



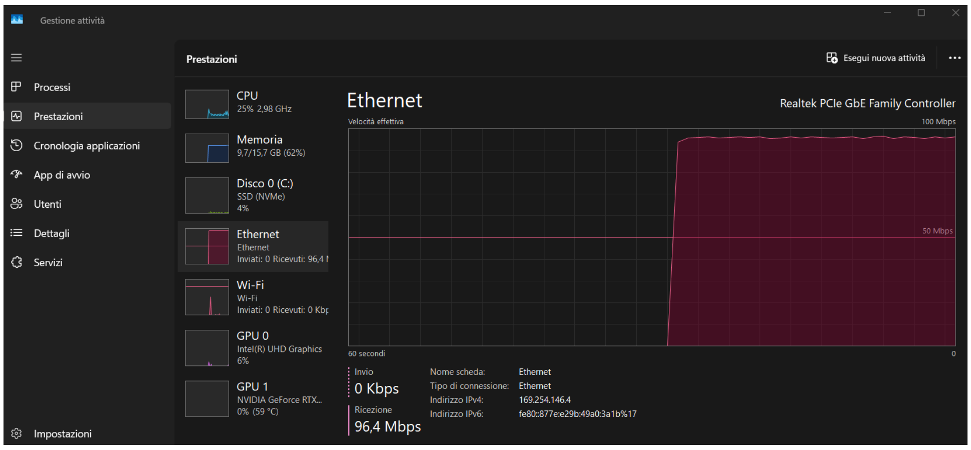
```
0130  62 64 60 d9 5f 48 5d b5  5c 18 5a 69 58 b4 56 f9   bd`·_H]· \·ZiX·V·
0140  55 34 53 69 51 94 4f b9  4d d4 4b e9 49 f4 47 f9   U4SiQ·O· M·K·I·G·
0150  45 f4 43 e9 41 d4 3f b9  3d 94 3b 69 39 34 36 f9   E·C·A·?· =·;i946·
0160  34 b4 32 69 30 18 2d b5  2b 48 28 f9 26 98 24 95   4·2i0·-· +H(·&·$·
0170  22 6c 20 e5 1d 7a 1a cb  18 08 15 35 12 52 0f 63   "l ··z·· ···5·R·c
0180  0c 62 09 55 06 44 03 25  00 00 ff ff fc db f9 bb   ·b·U·D·% ········
0190  f6 ab f3 9d f0 9d ed ad  ea cb e7 f7 e5 35 e2 85   ········ ·····5··
01a0  df 1b dd 93 db 6b d9 67  d7 07 d4 b7 d2 4b cf e7   ·····k·g ·····K··
01b0  cd 97 cb 4b c9 07 c6 cb  c4 97 c2 6b c0 47 be 2b   ···K···· ···k·G·+
01c0  bc 17 ba 0b b8 07 b6 0b  b4 17 b2 2b b0 47 ae 6b   ········ ···+·G·k
01d0  ac 97 aa cb a9 07 a7 4b  a5 97 a3 e7 a2 4b a0 b7   ·······K ·····K··
01e0  9f 27 9d 9b 9c 17 9a 9b  99 27 97 bb 96 57 94 fb   ·'······ ·'···W··
01f0  93 a7 92 5b 91 17 8f db  8e a7 8d 7b 8c 57 8b 3b   ···[···· ···{·W·;
0200  8a 27 89 1b 88 17 87 1b  86 27 87 1b 88 17 89 1b   ·'······ ·'······
0210  8a 27 8b 3b 8c 57 8d 7b  8e a7 8f db 91 17 92 5b   ·'·;·W·{ ·······[
0220  93 a7 94 fb 96 57 97 bb  99 27 9a 9b 9c 17 9d 9b   ·····W·· ·'······
0230  9f 27 a0 b7 a2 4b a3 e7  a5 97 a7 4b a9 07 aa cb   ·'···K·· ···K····
0240  ac 97 ae 6b b0 47 b2 2b  b4 17 b6 0b b8 07 ba 0b   ···k·G·+ ········
0250  bc 17 be 2b c0 47 c2 6b  c4 97 c6 cb c9 07 cb 4b   ···+·G·k ·······K
0260  cd 97 cf e7 d2 4b d4 b7  d7 07 d9 67 db 6b dd 93   ·····K·· ···g·k··
0270  df 1b e2 85 e5 35 e7 f7  ea cb ed ad f0 9d f3 9d   ·····5·· ········
0280  f6 ab f9 bb fc db ff ff  00 00 00 00 00 00 00 00   ········ ········
0290  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
02a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
02b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
02c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
02d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
02e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
02f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0300  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0310  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0320  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0330  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0340  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0350  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0360  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0370  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0380  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0390  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
03a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
03b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
03c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
03d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
03e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
03f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········

165429 · Source: ICANNIANADep_00:fa:ce · Destination: Broadcast · Protocol: 0x0000 · Length: 1416 · Info: Ethernet II
```

Figure 5: Second packet transmitted, filled with zeros. Packet captured with WireShark.

- **High-Throughput Validation:** A separate "freerun" test, where the FPGA continuously sends data, was monitored using the operating system's task manager. This test demonstrated a stable average data rate of 98 Mbps on the Ethernet port. This confirms the effectiveness of the design choices for high-speed data handling, particularly the efficient use of the Ethernet IP's dual transmit buffers by the `AXI_interface_PSI` module's ping-pong addressing scheme and the fast FIFO passthrough.

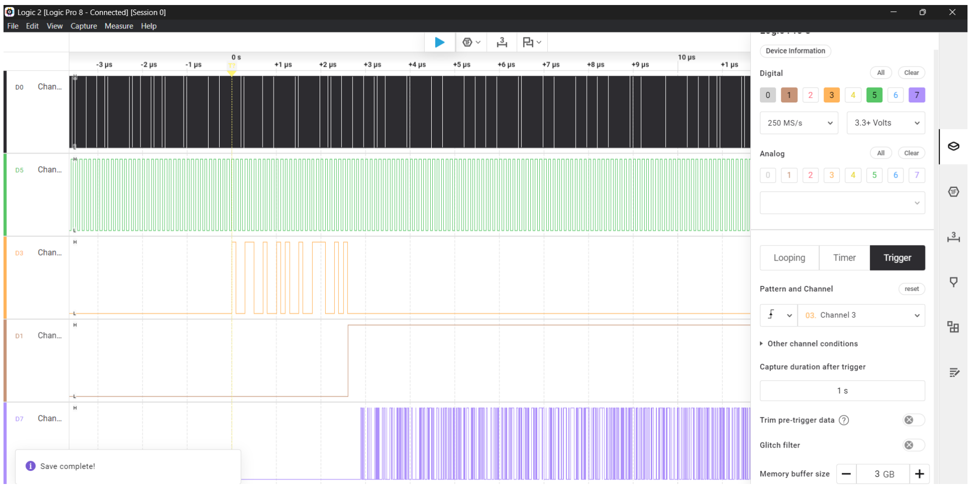
- **Logic Analyzer test:** During the final revision and project testing, a Saleae logic analyzer was used to observe the FPGA signal waveforms in detail. Specifically, the PSI and SPI clocks, SPI load enable, SPI data, and PSI data signals were mapped to dedicated pins on the Arty A7-35, which were then connected to the logic analyzer. Subsequently, we captured the signals using the "Logic 2" software, confirming that all signals were streamed correctly. Crucially, the waveforms clearly demonstrated that the SPI signal is read first, and only after the load enable signal asserts high does the PSI stream begin. This sequence confirms the intended data transfer protocol.



Invio
0 Kbps
Ricezione
96,4 Mbps

Nome scheda: Ethernet
Tipo di connessione: Ethernet
Indirizzo IPv4: 169.254.146.4
Indirizzo IPv6: fe80::877e:e29b:49a0:3a1b%17

Figure 6: Max speed transmission.



250 MS/s
3.3+ Volts
Channel 3
Capture duration after trigger: 1 s
Trim pre-trigger data
Glitch filter
Memory buffer size: 3 GB

Figure 7: SPI data (yellow), SPI le (brown), PSI data(purple) sequence.

6 References and Tools

6.1 References

Here are some references to user guides that proved useful for understanding the functionality of IP cores utilized in the project.

- **AXI Ethernet Lite:** AXI Ethernet Lite MAC v3.0 LogiCORE IP Product Guide
- **FIFO generator:** FIFO Generator v13.2 LogiCORE IP Product Guide
- **Clocking wizard:** Clocking Wizard v6.0 LogiCORE IP Product Guide
- **Arty A7:** Arty A7 Reference Manual

6.2 Tools and instruments

To achieve the success of this project, several platforms were used.

- **Vivado 2023.1:** Used to create the HDL components using the VHDL programming language. It proved invaluable for implementing a modular, entity-based project onto an FPGA.

- **Matlab:** A powerful mathematical IDE that enabled the creation, plotting, and sampling of sinusoids. The resulting samples were then used to populate the lookup table.
- **Python:** Thanks to its powerful Scapy and Matplotlib libraries, Python made it possible to sniff and monitor traffic on the PC's specific Ethernet port and to capture Ethernet (Layer 2) packets. It also allowed for slicing the 1400-byte packets into 16-bit words, saving all the data, and plotting them to visualize the final results.
- **Xilinx Arty A7-35:** The chosen FPGA for this project, on which the HDL code was implemented.
- **Saleae logic analyzer:** Useful for capturing the signal waveforms of SPI and PSI data and clocks, serving as a final confirmation that everything was functioning correctly within the FPGA.
- **Wireshark:** A powerful software that allowed for precise inspection of sent packets, including headers such as MAC addresses, and facilitated the correct adjustment of endianness.

7 Conclusions

This project successfully developed a VHDL-based FPGA interface for a custom high-speed data acquisition chip. The system reliably configures the chip (simulated by `test_DUT`) via SPI and captures high-throughput PSI data, transmitting it over Ethernet at rates approaching 100 Mbps (98 Mbps achieved).

The design tackled significant challenges, including managing high-speed data streams, ensuring data integrity across clock domains using FIFOs, and optimizing Ethernet packetization for maximum throughput. The use of a mock-up DUT was crucial for iterative development and testing. The ping-pong buffering strategy for both PSI data input into the FIFO and for utilizing the Ethernet IP's transmit buffers proved effective in achieving the performance goals. The timeout mechanism for sending the final data packet ensures that all captured data is transmitted promptly.

This project has been a demanding yet highly rewarding experience. It provided an opportunity to engage with a complex hardware design, from architectural planning to detailed implementation and verification. Seeing the system achieve such high data rates and successfully synchronize its diverse components (SPI, PSI, AXI, Ethernet) has been particularly gratifying. The skills developed in VHDL design, AXI interfacing, and high-speed data handling will be invaluable for future endeavors.

Acknowledgements

I would like to express my sincere gratitude to Professor Luca Bertulessi for providing the opportunity to work on this challenging and exciting project within their research group. His guidance and vision were instrumental. I also extend my thanks to PhD candidate Michele Rocco for his consistent support, insightful discussions, and technical assistance throughout the months of development. His expertise was a great help in navigating the complexities of the project.