Python-based Trace and Debug of FPGA Designs

Jonathan Stenroos
jonathanstenroos@hotmail.com
Hugo Sjödin
hugo-sjodin@hotmail.se

Department of Electrical and Information Technology
Lund University

Supervisor: Liang Liu

Examiner: Pietro Andreani

May 4, 2026

© 2026
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

In the hardware development space, field programmable gate arrays (FPGAs) are in a unique position as a result of their in-field re-programmability. In contrast to the traditional design process of integrated circuits, FPGAs allow iterative development workflows closely resembling software development's sprint-based prototyping. For this to work, efficient workflows for verification are key. Verification can be done through simulators, but for complex designs with intricate dependencies of surrounding hardware, these are not always sufficient. For such cases, in-circuit debugging solutions can be utilized. These represent debugging cores that are implemented on the FPGA silicon itself, capable of monitoring internal signals during system runtime.

In this master's thesis project, a debugging system has been developed. It was designed to provide a convenient and resource efficient way to perform post-implementation in-circuit verification of FPGA designs. The debugging system was made to be configurable and controlled through a Python library, allowing extensive scripting and unit-testing capabilities. The debugging system is vendor and platform agnostic and easy to integrate into a pre-existing Python-based verification workflow. We evaluated our system by comparing metrics such as ease-of-use, functionality, and resource consumption against Xilinx Vivado's Integrated Logic Analyzer (ILA).

Popular Science Summary

When building integrated circuits, it is important to have systems in place for making sure that they work as intended. For this purpose, there exists computer programs called simulators that can mimic the functionality of the circuit and show how the internal signals react when exposed to different stimulus. Simulators are very convenient to work with, controlled through intuitive user interfaces and gives complete system observability. All problems cannot be solved through simulators however, a tester needs to be able to see how the circuit behaves in the real world and how it interacts with the hardware that it is connected to. One way to do this is to install a so called embedded logic analyzer (ELA) inside the circuit itself. An ELA can be seen as an advanced multimeter, it represents dedicated verification hardware capable of monitoring signals inside the circuit. To some extent, the goal of an ELA is to provide the functionality of a simulator but working on the real implemented system rather than a software model.

This project is about implementing a custom ELA module written in the hardware description language System Verilog. It is meant to be used for verifying a type of programmable circuit board called FPGAs. There already exists commercially available ELAs today, but the system that we have developed has a few key features that distinguish it from these. Firstly, it is built to be completely controlled and configured through the Python programming language. By wrapping complex verification systems behind a user friendly Python library we aim to create a streamlined verification process. Secondly, it is extensively reconfigurable to fit a tester's needs. Our design has systems in place for dynamically specifying things such as which signals to monitor even after the circuit is fully constructed and in operation. For popular commercial ELA solutions such as Xilinx Vivado's Integrated Logic Analyzer (ILA), these types of configurations usually have to be done before they are built and placed in the FPGA circuit.

We have evaluated our test system against Vivado's ILA in terms of metrics such as how complex and resource intensive the systems are and how easy they are to interact with for a tester. We found that our system presents some key benefits, but also noteworthy drawbacks. On the one hand, our system proved to be more resource efficient than Vivado's ILA in many circumstances, in part due to the dynamic reconfiguration feature allowing a tester to cover a large amount of signals cheaply. On the other hand, Vivado's ILA offers a greater breadth of features as well as a more intuitive user interface.

Preface

The project was proposed and commissioned by Ericsson Silicon in Lund and their FPGA team. We want to thank them for the opportunity and assistance throughout the project. Especially our company supervisor Kevin Cushon, who has provided important insight regarding both the report and the system design.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

The ongoing development of 6G network equipment has shown a need for dedicated, fast, and modular hardware systems. Constantly changing requirements and goals necessitate iterative workflows with continuous prototyping and testing, where the traditional waterfall development model is too slow and costly. Field programmable gate arrays (FPGA) have proven to be an invaluable tool in this workflow. In the book *FPGAs: Fundamentals, Advanced Features, and Applications in Industrial Electronics* by Andina Rodriguez et al., the authors highlight the FPGA's role in the embedded systems market [1]. It is a sort of middle ground between dedicated Application Specific Integrated Circuits (ASICs), with strong performance and power efficiency, and general purpose micro controllers, with flexibility and programmability. The authors point out that the FPGA's characteristics become especially beneficial in applications with diverse and dynamic working environments, where requirements and specification such as communication protocols or hardware integration might be changing over time.

1.1 Importance of debugging

In the study *Error Cost Escalation Through the Project Life Cycle*, researchers at NASA explore the economic consequences of errors in projects [2]. The study focuses on how the costs of resolving errors relate to when in the project life cycle the errors are detected, and they conclude that there is an exponential relation. They define five project stages: Requirements, design, build, test, and operations. The study showed that identifying a critical error in the operations phase, after the system has been deployed, could cost up to 1000 times as much as during the design phase. From this, the authors conclude that identifying errors as early as possible is vital for a project's success.

Modern embedded systems are exceedingly complex. They consist of a multitude of interconnected hardware components that all need to work in tandem and as complexity grows, so does the risk of errors - necessitating a robust verification process.

In the case of integrated circuits, simulation software can be used during the design phase to verify functionality. Software models of the circuit give full signal observability in pre-programmed and reproducible testing scenarios, allowing early detection of errors. When the circuit is implemented on real hardware, however,

new errors could be introduced. Timing violations, faulty hardware, or environmental conditions might cause a design that behaves correctly in simulation to break post-implementation.

For an ASIC design, post-implementation errors are in many cases disastrous — possibly requiring a complete re-spin, an extremely costly endeavor in terms of both monetary resources and man-hours [3]. If instead an FPGA is used, the re-programmability facilitates ways to correct the errors seamlessly, in a process closely resembling pushing a software patch. To do this, the exact source of the bug needs to be identified, and for that purpose there exists post-implementation debugging systems. These provide functionality similar to that of software simulators, allowing a tester to observe internal signal behavior, but in contrast run the design on real FPGA hardware.

Furthermore, a strong post-implementation debugging system enables an agile workflow for FPGA prototyping. The testing and verification phase, which for ASICs necessarily needs to be extremely robust pre-implementation, can for FPGAs instead be done through a more iterative and efficient approach, shortening time to market.

1.2 Scope

The scope of this thesis is to develop an in-circuit debugging system for FPGA designs. The system should be capable of injecting test data and collecting signal samples from a device under test (DUT) implemented on hardware. The solution is built to be platform and vendor agnostic, with parameterization to fit a wide range of designs.

1.2.1 Design overview

The test system consists of two components. Firstly, a hardware defined test access point (TAP) written in System Verilog, connected directly to the DUT's interconnects. The TAP uses these connections to inject and sample data. Secondly, a Python library running on a PC (a Raspberry PI zero 2 w) controlling the TAP through a proprietary five-wire serial peripheral interface (SPI). An overview of the system is shown in figure 1.1.

The test system is designed to allow post-synthesis reconfiguration of the debugging setup. Using the Python library, named `tap_lib`, a tester can specify where and what test data to inject into the DUT as well as which signals to sample. This approach allows for extensive and varied tests on the DUT without the need for time-consuming re-synthesis and implementation.

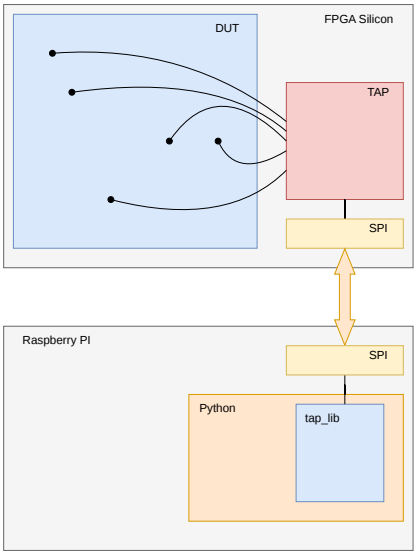


Figure 1.1: Block diagram of the TAP system.

1.2.2 Test system requirements

What follows are the core requirements for the debugging system, decided on in collaboration with Ericsson and our supervisors:

- Using a custom made Python library, a user should be able to specify which signals are to be observed and the test data that should be injected.
- The samples from the observed debug signals should be stored in FPGA block RAM during test execution, and sent back to the computer through SPI when the test finishes. SPI was decided upon because of its simplicity, higher bandwidth protocols such as UDP over Ethernet would not give any immediate benefit.
- The Python library should provide functionality for convenient visualization and analysis of the sampled signals. We will take advantage of available open source solutions for this.

1.2.3 Thesis work

The thesis work consists of: (1) Designing the TAP using System Verilog, verifying its functionality through simulations and test benches in Vivado. (2) Implementing the Python library `tap_lib` and defining a communication protocol for PC to FPGA SPI transactions. (3) Documenting an example of how the test system can be used to verify a real FPGA design. (4) Evaluating the test system and comparing it with Xilinx Vivado's own debugging system, the ILA, in regards to metrics such as e.g. silicon resource consumption and ease-of-use.

Chapter 2
Background and related work

2.1 In-circuit debugging

Debugging FPGA designs post-synthesis is an important tool in the verification process. A tester cannot rely entirely on simulation tools as growing design complexity leads to increasingly resource intensive simulation steps, and the tools can never fully account for the FPGA's final hardware integration [4], [5].

A common approach to in-circuit debugging is to use an embedded logic analyzer (ELA). This is a debug core implemented on the FPGA silicon that observes internal signals and uses memory resources like LUT or BRAM to store signal samples at runtime. Through some communication interface (e.g. SPI), the samples are transferred to a PC for analysis. In contrast to using an external logic analyzer, the embedded approach consumes less I/O resources and does not require dedicated sampling hardware. The drawback is instead an increased silicon usage as the full sampling logic needs to be defined through state-machines and memory blocks in System Verilog [4].

2.2 Integrated logic analyzer (ILA) by Xilinx Vivado

As in-circuit debugging is such a crucial step in the verification process, many FPGA vendors provide standard debug cores for this purpose. For Xilinx Vivado, their solution is called the Integrated Logic Analyzer (ILA) and this is what will be used as a reference when evaluating our test system [6].

The ILA performs signal monitoring during DUT execution according to pre-synthesis user configuration. A tester specifies which signals to monitor, resulting in an ILA instance to be synthesized and implemented on the FPGA. The conditions for when these signals should be sampled, the triggering conditions, can be altered during runtime. The ILA stores its samples in FPGA memory and continuously transmits these back to the PC over JTAG for monitoring in Vivado. The extent of the debug core's FPGA resource consumption will depend on factors such as how many signals to sample, the memory depth, etc.

The ILA is heavily integrated into the Vivado development environment. Parameterization, signal selection, and the viewing of the resulting signal traces are all done in the Vivado interface. Importantly, a tester does not need to write or modify any HDL code to accommodate the debug core. There are automatic systems in place for instantiating an ILA inside the DUT according to the GUI-specified configuration.

2.3 Limitations of ILA and runtime signal selection

The ILA is a common choice for in-circuit debugging, probably in large part because of the strong Vivado integration — making it easy to setup and work with. Despite this, the debug core has some important limitations that make it unsuitable for some testing scenarios, especially in large complex systems.

The main reason for why the ILA does not scale well with system complexity is its static pre-synthesis configuration. If a tester wants to change any of the debug core's parameters, such as e.g. which signals to monitor, the complete implementation workflow has to be re-run (from synthesis to bit stream generation). This can be a very time consuming process in complex designs [6], [7].

Reducing the need for re-synthesis in in-circuit debugging systems is a field with a lot of active research and many novel approaches have been proposed. For this master's thesis, the work of Fiala et. al. presented in [7] is of interest. In this paper, the authors describe a custom test system (called AID) that uses an external PC to communicate with the FPGA debug core at runtime for dynamic signal selection. The basic idea is that a tester will specify a large number of signals (up to 300) to connect to the debug core pre-synthesis and then during runtime the PC will be used to select a small subset (up to 4) of these for monitoring. The monitored signals are continuously sent back to the PC via UDP over Ethernet, allowing infinite signal monitoring without filling up FPGA memory resources. The controller program running on the PC was implemented in C# and is interacted with through a custom GUI.

There are two main principle differences between our system and AID. Firstly, our system utilizes FPGA memory to store samples at runtime before sending them to the PC. This allows for higher bandwidth, enabling us to monitor more signals, but increases the FPGA resource consumption. Secondly, our system runs through a scriptable Python library rather than a GUI, enabling better support for reproducible test procedures and the ability to integrate the test system into a larger Python-based workflow.

It is also important to note that our test system's ability to inject test data into the DUT is not a feature that most in-circuit debugging solutions support - neither the ILA nor the one proposed in [7] have such capabilities.

2.4 Scan chains

The debugging techniques mentioned thus far are all based on signal tracing, capturing the behavior of internal signals at specific probe points. There exists other approaches, however, that provide observability by recording system state instead [8]. The goal of these techniques is to gain access to the DUT's flip-flop modules and one way to do this is through *scan chain based state access* as discussed by Koch et al. in [9]. The idea is to chain the flip-flops together into a circular shift register chain (the scan chain), using multiplexers to toggle the shifting operation. With this setup, the full system state can be read non-destructively by shifting through the scan chain.

Compared to signal tracing tools like ILA, the scan chain technique has some key benefits. Firstly, the memory resource overhead can be made very small thanks to the fact that it is utilizing the DUT's own flip-flops. Because of this, it can be possible to achieve a high degree of observability even in high utilization designs. Secondly, the scan chain structure provides ways to manipulate the system state through external stimulus. User-defined state data can be shifted into the scan chain, giving controllability in a similar way to the data injection's of our test system [9], [8].

However, there are also severe drawbacks to the scan chain approach that ultimately led us to pursue signal tracing for this project. Most importantly, the basic scan chain design as presented in [9] severely reduces the execution frequency of the DUT. Recording the system state requires a full rotation of the scan chain, waiting for eg. an external PC to read each flip-flop value between shifts. In contrast, the signal tracing of e.g. the ILA can be run in real-time without any performance impact. Koch et al. propose *shadow scan chains* as a way to mitigate the performance impact in situations where high frequency state recordings are needed, but this instead requires duplicating all the DUT's flip-flops, leading to a large silicon overhead. That only the flip-flop states can be observed is another intrinsic drawback of scan chains. In situations where observing signal values within intermediate combinational logic is needed, the scan architecture fails to provide the necessary visibility.

2.5 FPGA to PC communication

2.5.1 SPI

Serial peripheral interface (SPI) is a duplex, synchronous, serial communication protocol originally developed by Motorola Inc. SPI follows a master-slave architecture, where one device (denoted master) controls communication between itself and one or more devices (denoted slaves). Originally SPI used four logic signals: MISO, MOSI, CLK and $\overline{SS}$, this variation is often called four-wire SPI. Other variants have also been defined throughout the years, such as three-, two- and one-wire SPI, each with their own benefits and drawbacks [10].

- **MOSI** (master out slave in), this pin is for data transmitted from the master device to the slave devices.
- **MISO** (master in slave out), this pin is for data transmitted from a slave device to the master device.
- **CLK** (clock), this pin is for the clock of the master, sent to synchronize transactions.
- $\overline{\textbf{SS}}$ (slave select), this pin is used by the master to specify which slave device to target.

The SPI communication is driven by the master device. The master uses $\overline{SS}$ for transaction initiation and slave targeting while controlling the synchronization and bit rate through CLK. During a transaction, the two devices each use a shift register to shift out their transmission while at the same time shifting in the packet being received from the other device.

In [10] Motorola defines the concept of clock polarity (CPOL) and clock phase (CPHA), two configurations that together define the SPI mode. An SPI transaction between two devices will only work if both are using the same mode. CPHA specifies on which CLK edge the devices should sample and shift their registers respectively. CPHA=1 delays the transaction by one CLK edge, as opposed to CPHA=0 where the devices start sampling immediately at $\overline{SS}$.

The SPI specification does not define any flow control mechanism for slave devices. This means that in systems where a slave's response time is unknown to the master, one needs to design some novel approach for signaling that the slave is ready for the next transaction [10].

2.5.2 Clock domain crossing

Signals passed between two registers operating on different clocks creates a condition called clock domain crossing (CDC) [11]. This situation occurs in SPI communication, where register shifting and sampling is synchronized to the master-controlled CLK signal rather than the devices' system clocks.

If a CDC is left without a supporting synchronizer circuit, timing violations can arise. In SPI for example, signals coming from the master device will be asynchronous relative to the slaves' shift registers, meaning that they might fluctuate during the setup and hold period, thereby breaking the circuit logic. Asynchronous signals causing register timing violations like this is an occurrence of metastability [11].

There are multiple ways to design a synchronizer that avoid the metastability issue. A popular solution is to let the asynchronous signals pass through a series of flip-flops before routing them to the receiving circuit, where the flip-flops stabilize the signal and reduce the risk of metastability [11].

2.5.3 SPI in Linux and Python integration

The Linux kernel has support for SPI communication through the userspace API `spidev`. This supports reading and writing to SPI slave devices through both half-duplex and full-duplex SPI transactions. The driver follows the four-wire SPI specification and can be configured to operate in any of the SPI modes mentioned in section 2.5.1 [12].

To work with SPI in Python, one can use the `py-spidev` library. This is capable of interacting with SPI controllers connected to the system through USB or PCI buses. The library runs on API calls to the `spidev` driver [13].

2.5.4 Raspberry PI and GPIO

Raspberry Pi is a computer manufacturer that makes single board computers. Importantly, the computers are equipped with general purpose input/output (GPIO) header pins. These can be controlled through Linux kernel drivers and have SPI capabilities [14].

The Python library `gpiozero` can be used to control the GPIO header pins. The library provides functions for reading pin values, waiting for a value change, etc. [15].

2.6 Signal Tracing

2.6.1 Block RAM

FPGAs are made up of programmable logic arrays referred to as slices. These slices can be used to define memory modules through e.g. flip-flops and lookup tables (LUTs). As a circuit's memory demand grows, however, using slices for memory becomes increasingly inefficient. FPGA silicon that could have been used for functional logic will be taken up by the memory modules, limiting the maximum possible system size. To mitigate this problem, modern FPGAs contain dedicated memory hardware modules called block RAM (BRAM). BRAM is more space and power efficient than slice defined memory and can work at higher frequencies without timing violations [16].

2.6.2 VCD file format and waveform viewer

Value change dump (VCD) is a file format commonly generated by logic simulation tools to specify signal dump data. The file format is defined by IEEE in [17] along with the Verilog hardware description language (HDL). Notably, the format defines the signals' timelines by including timestamps where value changes have occurred.

There are multiple available commercial and open source tools for viewing VCD dump files. These tools are called waveform viewers, and in this thesis we have been using the one integrated in Vivado as well as an open source solution developed by Linköping University called Surfer [18].

2.6.3 Trigger condition

When tracing signals, care must be taken to conserve memory resources. If an ELA were to sample every single signal change, all available memory resources would quickly fill up, and important signal data might be lost. Therefore, ELAs often have systems in place to ensure they only save signal values that are of interest, reducing memory consumption and making later analysis easier. This is generally called the trigger condition, a configuration made by the tester that specifies when the probe points should sample their signals. The limitations and granularity of the trigger condition will vary depending on ELA. For Vivado's ILA, a tester has a lot of control. Here, one constructs a boolean expression based on all the monitored signal values. When satisfied, the ILA captures a snapshot of signal activity from both before and after the event and saves it to FPGA memory [6]. For an ELA designer it is important to consider how detailed of a trigger condition is necessary as silicon consumption increases with added complexity.

2.6.4 Advanced Extensible Interface (AXI)

The Advanced Extensible Interface (AXI) from ARM is a high performance and high frequency communication protocol for communication between on-chip modules. AXI is part of the Advanced Microcontroller Bus Architecture specification (AMBA). AXI is a versatile protocol, adaptable for a wide range of applications and use-cases through optional signals that can be included depending on design requirements. Communication through AXI follows a master-slave architecture, where the transmitting component has the role of master and the receiving component the role of slave [19] .

AXI contains a handshake system for the purpose of flow control. Both the master and slave has the power to throttle the transmission speed through these handshakes. The system is composed of two signals: `VALID`, sent from the master and indicates that it is ready to send the next payload, and `READY`, sent from the slave and indicates that it is ready to process the next payload. A transfer will occur when both `VALID` and `READY` is held high, this marks a successful handshake, and the master device can then start working on the next payload [19].

Chapter 3
Implementation

As discussed in the introduction, our test system consists of two distinct parts: The Python library `tap_lib` running on a Raspberry PI and the HDL defined TAP running on the FPGA. Our initial assumption was that the design of the TAP would dictate the design constraints of the entire system. Because of this, we chose to first implement the TAP and then design the library around it.

Our HDL development process was iterative in nature, slowly introducing functionality and using Vivado's simulation tools for continuous verification. The first module designed was the TAP's DUT interface and core sampling/data injection logic. We then gradually moved outward towards the main tap controller modules and the SPI driver.

When the first iteration of the TAP was feature-complete and thoroughly verified through test benches, development focus was moved to the Python library. It was important that the API was designed to be user-friendly, and the TAP easy to integrate into the DUT's system. We decided to use the Python library for TAP configuration and TAP top module definition in an effort to streamline the testing process. In the end, the test system was designed such that a tester will more or less only interact with the Python library, requiring only minimal HDL code modification in the DUT.

After implementing both the TAP system and the Python library, we tested our solution on several different DUT configurations, comparing the functionality and ease-of-use with Vivado's ILA.

3.1 TAP

This section will provide an overview of the TAP design, describing how it works and what design choices were made. The TAP consists of four core modules:

- **Stimulus Probe:** The TAP's DUT interface, sampling, and test data injection are done through this module.
- **Outpost:** The master of a number of Stimulus Probes. Contains BRAM for storing samples and test data.
- **Base station:** The master of a number of outposts. Organizes the testing procedure based on commands received from the Raspberry PI.
- **SPI driver:** An SPI controller running in slave mode, handling packet transactions between the base station and the Raspberry PI using a proprietary 5-wire SPI protocol.

An overview of the TAP module is shown in figure 3.1. The blocks A, B, and C are example DUT blocks, P represents probes, O the outposts, and B is the base station.

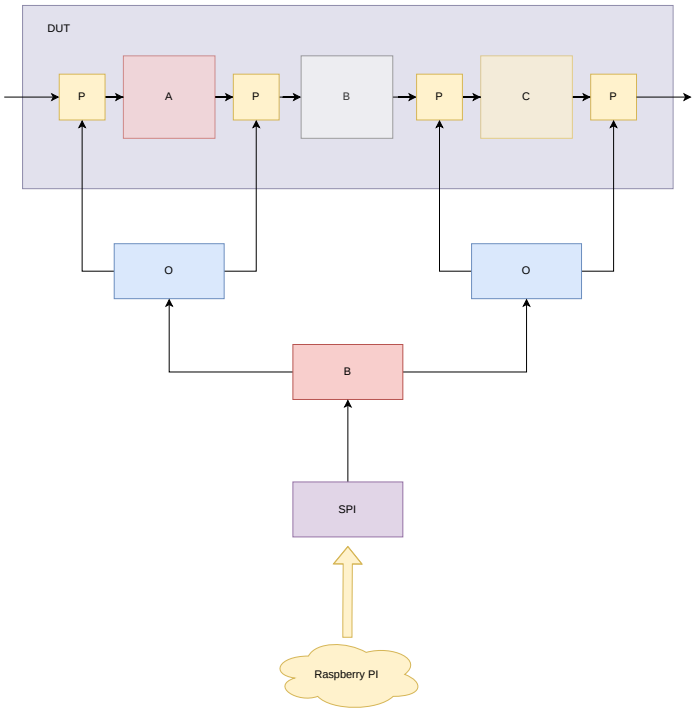


Figure 3.1: An overview of the TAP design.

3.1.1 Stimulus Probe

The probe is the TAP's interface with the DUT and is responsible for sampling and test data injection, operating through commands from its parent outpost. A block diagram of the Stimulus probe is shown in figure 3.2. The module takes two input signals from the DUT: The `s_bus`, corresponding to the data bus from which we want to sample, and the one bit wide `manual_trigger` which can be used to specify when `s_bus` should be sampled. The probe has one output into the DUT: The `m_bus`, representing the test data that we want to inject into the system. Communication between outpost and probe goes through two data buses with corresponding data valid signals denoted `probe_outpost_bus` with `sample_valid` and `outpost_probe_bus` with `do_inject` respectively. When injecting, `outpost_probe_bus` contains test data and is routed to `m_bus`. Otherwise, DUT data coming into the probe from `s_bus` is passed through to `m_bus`. This design allows the probes to remain non-intrusive when not in use.

The Stimulus Probe has optional AXI handshake support. When enabled, the `axis_ready` and `axis_valid` signals will be used for the handshake mechanism described in section 2.6.4. When injecting, the incoming `axis_ready` is sent to the parent outpost through `probe_outpost_bus` to be used for injection data flow control. Like for the other DUT signals, `axis_valid` and `axis_ready` is passed through the probe without modification when not injecting.

As shown in the block diagram, the probe contains a `sample_controller` module. This module is responsible for the sample trigger logic, for deciding when the intercepted bus should be sampled based on probe configuration specified by the tester. The controller has a burst trigger feature through the `sample_burster` module where the tester can configure how many samples should be taken when the triggering condition is met, like the shutter burst mode on a camera.

There are two ways a sample can be triggered: Either the one bit wide `manual_trigger` signal is pulled high, this signal comes from the DUT and is specified pre-synthesis by the tester, or there are bit changes in `s_bus`. The probe contains an internal bit mask `trigger_mask` that denotes which of the bits in the `s_bus` should be monitored for bit changes.

3.1.2 Outpost

The outposts are connected to several probes and are responsible for storing their samples and feeding them injection data. Importantly, an outpost can only process one probe at a time. This means that it can store samples or send inject data to one of its children only, the `probe_target`. The reason for this design is to reduce the outpost's complexity and thereby reduce its silicon consumption. The mechanism is a part of the TAP's ability to re-configure post-synthesis, mentioned in section 1.2. The tester can specify each outpost's `probe_target` through SPI commands between tests.

Also specified post-synthesis is whether the outpost is used for injecting or sampling. If the tester wants to use it for injecting, they transmit the test data through SPI to the outpost's memory module before the test starts. If no test data has been received, the outpost defaults to sampling. The injection process works by iterating over the data stored in the outpost's memory module and sending it

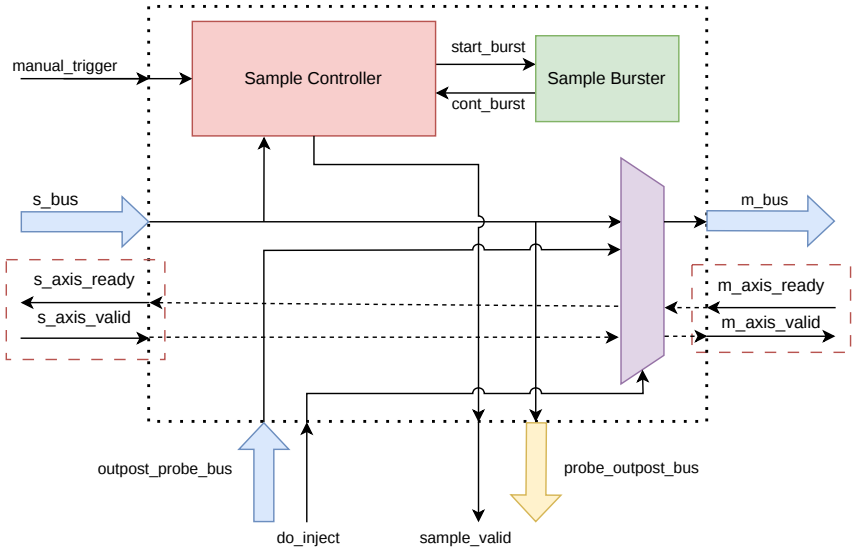


Figure 3.2: Stimulus Probe block diagram.

to its `probe_target` through the corresponding `outpost_probe_bus`. By default, each sample will be sent for one clock cycle until the memory is emptied. If AXI handshake support is enabled, however, each sample will be held until a valid AXI handshake has been established.

In figure 3.3, a block diagram for the outpost module is shown. The communication between outpost and base station is done through two buses with corresponding data valid signals: `outpost_base_bus` and `base_outpost_bus`. These are marked *Output to base* and *Input from base* respectively in the diagram. The signals denoted *Input and output from and to probes* are the previously mentioned Stimulus Probe buses `probe_outpost_bus` and `outpost_probe_bus` with the `probe_target` state specifying selection.

As we wanted our outposts to be able to store as many samples as possible, BRAM was chosen for their memory modules because of its high capacity compared to other alternatives such as LUT RAM. To instantiate BRAM in FPGAs, one can use a pre-existing licensed design. These are verified to work and probably well optimized, but as a goal of this project was to create a vendor and platform agnostic system, such designs were not applicable. Instead, we chose to write our own BRAM module in System Verilog. As the synthesis engine ultimately decides how our design is implemented, the module had to be written in such a way that predisposes it towards synthesizing the memory as BRAM. To make the memory easier to interact with, our BRAM module is wrapped in a simplified memory controller which hides some of the hardware interaction details.

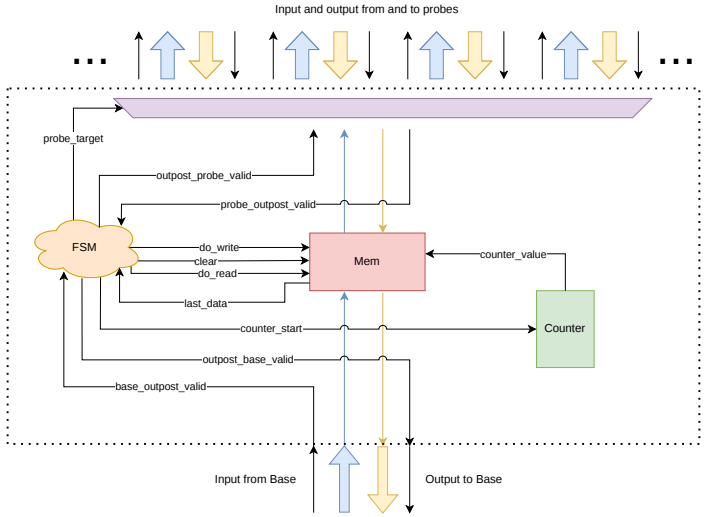


Figure 3.3: Outpost block diagram.

The functions of the outpost are accommodated by a finite state machine (FSM) of six states. The FSM is illustrated in figure 3.4 and the states are as follows:

- `IDLE_EMPTY`: The outpost is idle and does not have data in its memory module.
- `IDLE_FULL`: The outpost is idle but its memory module contains inject data.
- `DUMPING`: The outpost is dumping its memory content, sending it back to the base station to then be transmitted to the Raspberry PI through SPI.
- `PROBING`: The outpost is listening for samples arriving from its `probe_target` and storing them in its memory module.
- `INJECTING`: The outpost is sending inject data from its memory module to `probe_target`.
- `FILLING`: The outpost is receiving inject data from the base station and storing it in its memory module to be injected into the DUT at a later stage.

State changes are triggered through three distinct event types: (1) Base station commands arriving on the `base_outpost_bus`. (2) External data valid signals. These are either a probe's `sample_valid` when in the `PROBING` state or the `base_outpost_valid` when in `FILLING`. (3) Task completion checks, relevant in the `FILLING`, `INJECTING`, and `DUMPING` states as these represent a finite amount of work that should be completed. `FILLING` is deemed complete when the base station has no more injection data to transmit, the end of transmission is marked by

a one bit header on the `base_outpost_bus`. `INJECTING` is deemed complete when all test data has been injected into the DUT, this is marked by an `is_last_data` output signal from the memory module. `DUMPING` is deemed complete when the memory module has been emptied and all data has been sent to the base station, this is also marked by `is_last_data`.

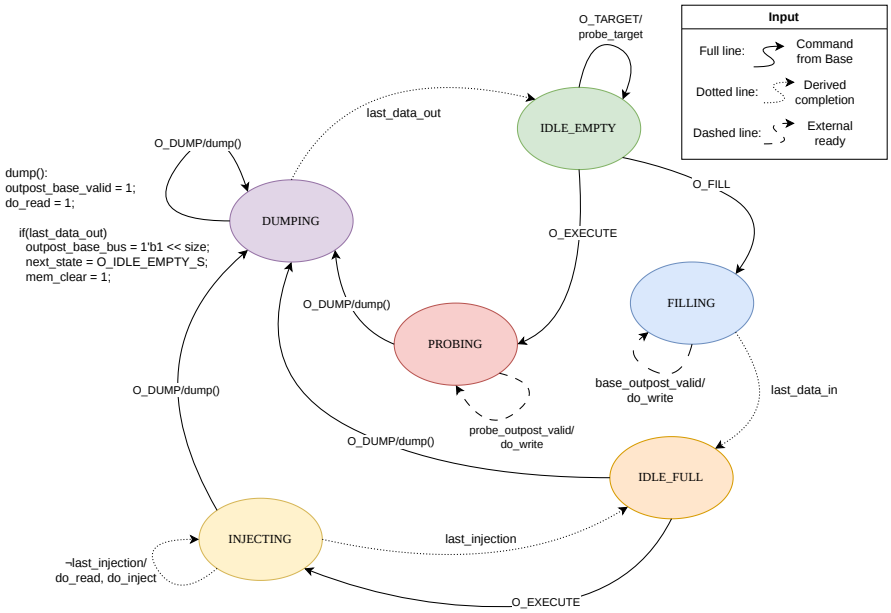


Figure 3.4: The outpost's FSM.

3.1.3 Base Station

The base station represents the brain of the TAP and controls the outpost configurations and states through Raspberry PI commands received by the SPI driver. In essence, the module works as a middle man between the SPI driver and the outposts. It contains no internal buffers or memory modules for data storage. Figure 3.5 shows a block diagram of the base station. The communication between the base station and the SPI driver is done through two buses with corresponding data valid signals: `base_spi_bus` and `spi_base_bus`. These are marked *Output to SPI* and *Input from SPI* respectively in the diagram. The signals denoted *Input and output from and to outposts* are the previously mentioned outpost buses `base_outpost_bus` and `outpost_base_bus` with the `outpost_target` state specifying selection.

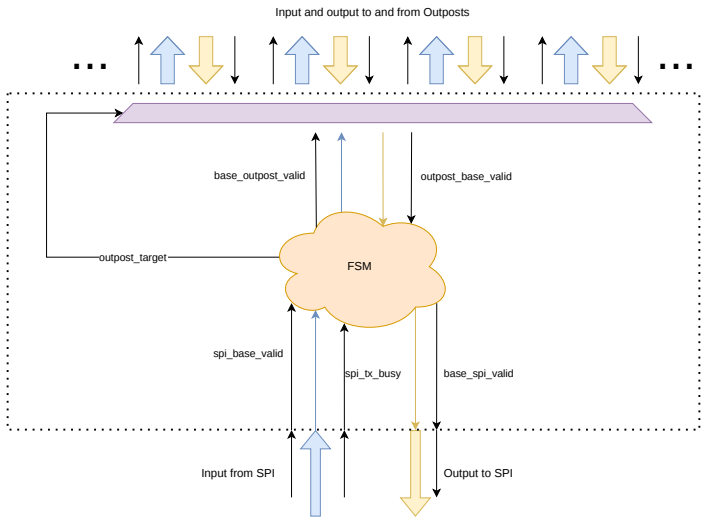


Figure 3.5: The base station block diagram.

The FSM of the base station is shown in figure 3.6 and the states are as follows:

- `IDLE`: The base station is idle and waiting for commands from the SPI.
- `FILLING`: The base station is receiving inject data from the SPI and passing this to its `outpost_target`.
- `EXECUTING`: The base station is currently conducting a test, the outposts are inject test data and sampling.
- `DUMPING_W_SPI`: The base station is iterating through its outposts and dumping their respective memory contents through SPI to the Raspberry PI. Currently waiting for the SPI driver to be ready to transmit a new packet.
- `DUMPING_W_O`: The base station is iterating through its outposts and dumping their respective memory contents through SPI to the Raspberry PI. Currently waiting for targeted outpost to begin sending its dump data.

State changes are triggered through three distinct event types: (1) Raspberry PI commands arriving on the `spi_base_bus` from the SPI driver. (2) External data valid signals. These are either the SPI driver's `spi_base_valid` when in the `FILLING` state, an outposts `outpost_base_valid` when in `DUMPING_W_O`, or the SPI driver's `tx_busy` when in `DUMPING_W_SPI`. (3) Task completion checks, relevant in `DUMPING_W_O` and in `FILLING`. `DUMPING_W_O` is deemed complete when the current outpost target has finished dumping its memory contents, indicated by a one bit header on the `outpost_base_bus`. `FILLING` is deemed complete when the SPI driver has transmitted the last test data sent from the Raspberry PI, indicated by a one bit header on the `spi_base_bus`.

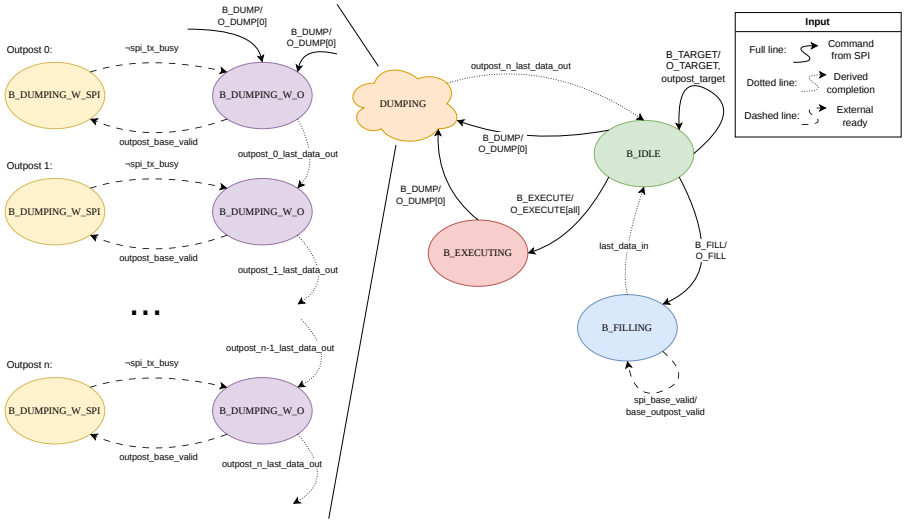


Figure 3.6: The base station's FSM.

As seen in the FSM, the base station can receive four different types of commands from the SPI driver. Of these, only `TARGET` carries a payload. A targeting packet looks like this:

$$\texttt{CMD}|\texttt{OUTPOST\_ADDR}|\texttt{PROBE\_ADDR}, \tag{3.1}$$

where `CMD` identifies the command as a `TARGET`, `OUTPOST_ADDR` is the address of the outpost of which the `probe_target` should be set, and `PROBE_ADDR` is the address of the probe.

3.1.4 SPI driver

The SPI driver was implemented to follow the four-wire serial specification, as a slave device running in the $(\texttt{CPOL}, \texttt{CPHA}) = (0, 0)$ SPI mode. The driver implements full duplex transactions, but because of how the base station's FSM is designed, data will in practice only be transmitted in one direction at a time.

A block diagram for the SPI driver is shown in figure 3.7. The driver receives the standard SPI slave input signals CLK, denoted `master_clk` in the figure, MOSI, and $\overline{SS}$ from the Raspberry PI — which is configured as a master device through `spidev`. It sends two signals back to the master, the standard MISO together with a custom flow control signal `data_ready`. `data_ready` is used to tell the master that new data is available.

The base station and the SPI driver are connected through five signals. These are the main RX and TX buses `spi_base_bus` and `base_spi_bus` with corresponding data valid signals and `tx_busy` specifying whether the driver is ready to schedule a new packet for transmission.

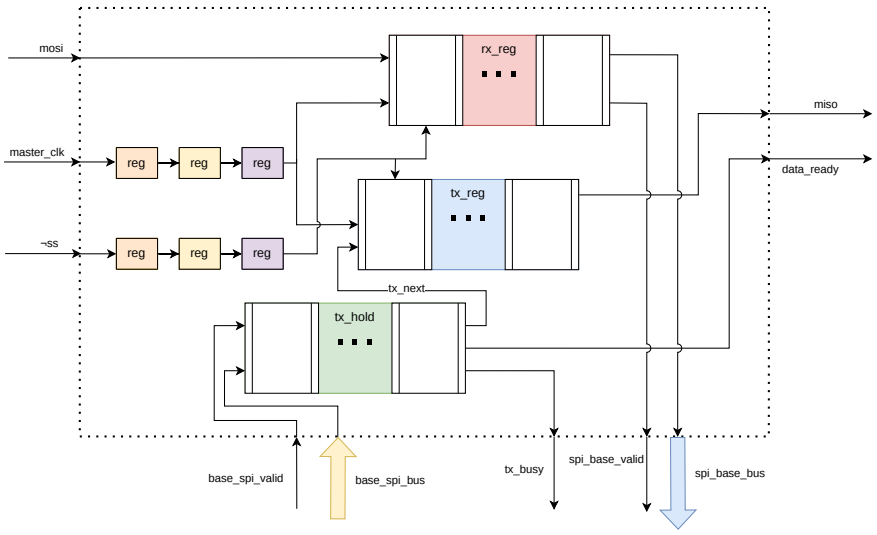


Figure 3.7: The SPI driver block diagram.

As shown in the block diagram, the SPI driver contains two shift registers, one for receiving packets (`rx_reg`) and one for transmitting (`tx_reg`). These perform serial-to-parallel conversion and parallel-to-serial conversion respectively. As described in section 2.5.2, the signals coming from the SPI master must be treated as asynchronous in the FPGA's clock domain. To reduce the risk of metastability, a three stage synchronizer circuit consisting of chained flip flops has been added to `master_clk` and `ss_n`. All registers in the SPI driver runs on the FPGA's clock rather than `master_clk` and SPI events are identified through edge detection circuits comparing the synchronizers' second and third stages.

Besides the two shift registers, the SPI driver also contains a `tx_hold` register used for storing the next packet scheduled for transmission. In practice, this will always be dump data from outposts that has been sent to the SPI driver through the base station. The flow control signal `data_ready` will be held high when there is new data in `tx_hold` and the master has not yet pulled `ss_n` low.

3.2 tap_lib

This section will provide an overview of the Python library `tap_lib`. Its design and features will be covered, as well as how to use it.

`tap_lib` has three core functions. Firstly, it contains systems for describing a TAP configuration — specifying parameters such as the number of probes and outposts, their bus sizes, triggering conditions, etc. Secondly, System Verilog generation methods that uses the TAP configuration to write a System Verilog file containing the TAP's top module as well as a package of constants describing configuration derived parameterizations. Thirdly, a controller capable of communicating with the TAP over SPI to conduct tests on the DUT.

In the following sections, we will go through the steps a tester would take to test a DUT using `tap_lib`.

3.2.1 Tap Configuration

To define the TAP configuration, a tester starts by describing all the probes that they want to connect to the DUT. These are specified by the width of their `s_bus` and `m_bus`, their `trigger_mask`, and a unique name used for referencing. Given the probes, the desired outposts are described as the subset of probes that they control, the depth of their memory modules, and a name. A complete TAP description is made up of the outpost definitions together with the number of bits to use for the sample timestamps and the number of pipeline stages to use for the data buses between the base station and the outposts, as well as between the outposts and their probes. An example TAP configuration can be seen in listing 3.1.

Listing 3.1: An example TAP configuration using `tap_lib`.

```
tap: Tap = Tap (
  outposts = [
    Outpost(
      name="injector",
      memory_depth=4,
      probes = [
        Probe("injP", 3, 0b000)
      ],
      burst_duration=0
    ),

    Outpost(
      name="sampler",
      memory_depth=4,
      probes = [
        Probe("first", 3, 0b100),
        Probe("second", 3, 0b011)
      ],
      burst_duration=0
    )
  ],
  config = Config(
    timestamp_size=20,
    outpost_probe_pipeline_stages=2,
    base_outpost_pipeline_stages=2
  )
)
```

3.2.2 System Verilog generation and DUT integration

There are two reasons why `tap_lib` was designed to be responsible for generating the TAP's top module. Firstly, so that the TAP configuration defined in Python is guaranteed to match the actual configuration implemented on the FPGA silicon. This is crucial as the controller would not be able to properly communicate with the TAP otherwise. Secondly, to make it as easy as possible for a tester to instantiate the TAP in the DUT. The TAP system is complex and depends on a large number of modules, all extensively parameterized. By generating the instantiations, the tester avoids the error-prone, time-consuming, and abstract work of doing this manually. Also, the names associated with the outposts and probes can be utilized to establish a shared naming convention between that TAP and DUT.

The module generation gives the tester the final, configuration specific, TAP that they then should instantiate in the DUT. The module contains two different types of signals. Firstly, SPI signals that should be routed to FPGA IO by specifying constraints. These are `MISO`, `MOSI`, `SS_N`, `master_clk`, and `data_ready`. Secondly, the probe signals `s_bus`, `m_bus`, and `manual_trigger` for each probe defined in the TAP configuration. The names for these will be based on the ones defined in the configuration:

```
[signal name]_[outpost name]_[probe name].
```
(3.2)

3.2.3 Tap control and conducting tests

With the TAP implemented on the FPGA silicon and the Raspberry PI connected to the constraints-specified IO pins, a tester can start running tests on the DUT. A test specifies the targeting of each outpost, the test data to inject into the system, the duration of the test, and a file path to where the dump file should be stored. Listing 3.2 shows an example test definition where `tap_config` is the current TAP configuration. In this case, the outpost called "injector" is specified to be used for test data injection by passing the test data file "test_data.csv" as an additional parameter to its target. Test data is written in a one or two column wide `.csv` format, with rows representing signal values to be injected consecutively. Two columns can be used if the injection data is complex, which is marked by a boolean parameter in the `tap_lib.Outpost` constructor. If so, the first and second column will represent the data's in-phase and quadrature components respectively. On the FPGA, this will correspond to the upper and lower half of the targets probe's `m_bus`. The signal values are formatted as unsigned integers, and it is the corresponding raw bits that will be sent over the target probe's `m_bus`. The dump files are by default stored in the VCD file format, containing signal values and timing information, with optional support for a simpler CSV format.

Listing 3.2: An example test setup using `tap_lib`.

```
test: Test = Test (
  name="myTest",
  targets = {
       Test.Target(
         tap.outposts.injector.probes.injP,
         "test_data.csv"
       ),
       Test.Target(
         tap.outposts.sampler.probes.first
       )
  },

  dump_path="my_test.vcd",

  tap = tap_config,
  duration = 1
)
```

After the test is run, the tester can import the dump file into any VCD compatible waveform viewer. Figure 3.8 shows how this can look when a dump file is loaded into the Linköping University developed waveform viewer *Surfer*. Here, the two variables inject and sample correspond to the memory contents of the two outposts with the same name.

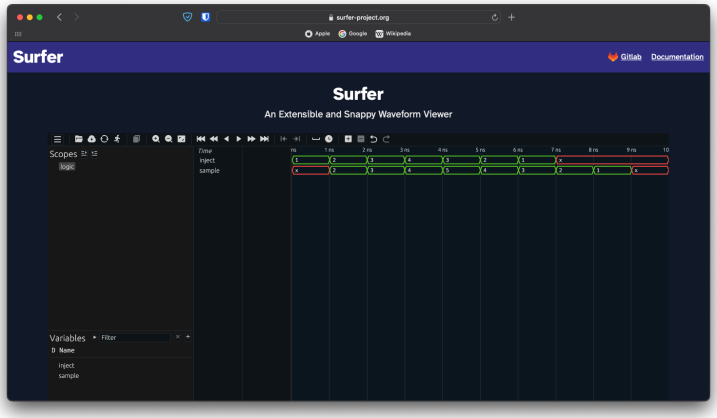


Figure 3.8: An example dump file imported into Surfer.

In addition to running singular tests, `tap_lib` also supports unit testing where tests can be associated with an expected dump file for automatic validation. Using this feature, a validation suite can be defined — a chain of consecutive tests that can be run on the DUT without re-synthesis. Figure 3.9 shows what it looks like to run the validation chain in the CLI.

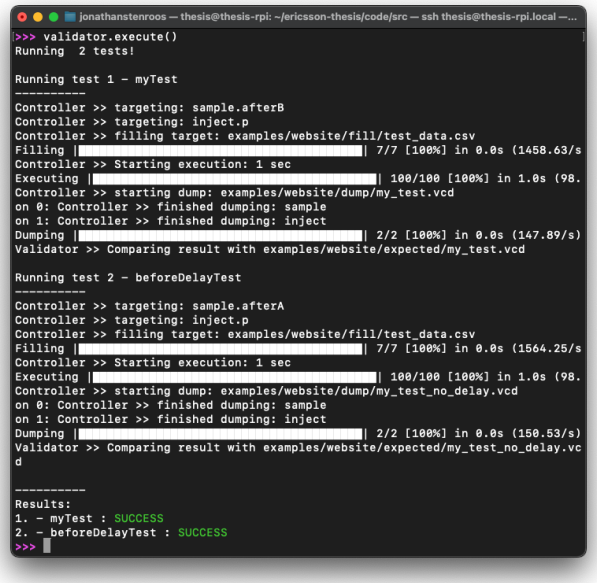


Figure 3.9: An example of a validation suite in the CLI.

3.3 Reference model integration

When designing FPGA modules at Ericsson, the engineers often build an accompanying reference model written in Matlab or Python. The reference model is a program that mimics the behavior and functionality of the FPGA module, and it is used as a part of Ericsson's verification process to ascertain that the design works as intended. A tester can in a simulator feed both the FPGA module and the corresponding reference model the same input signals and compare the outputs to spot inconsistencies.

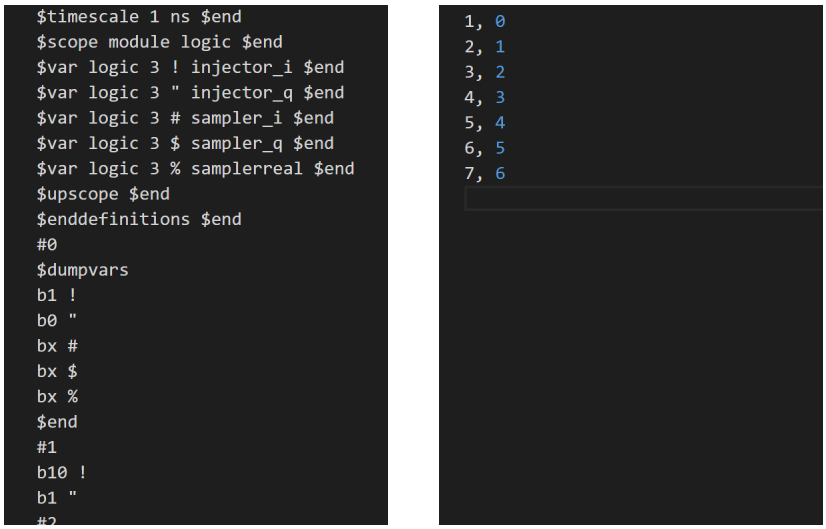
Because of our TAP's data injection capabilities, it is possible to incorporate the reference models into our in-circuit testing process in a similar way to what is currently being done with simulators. This could provide a convenient way to verify that the design still behaves correctly when running on real FPGA silicon, allowing a tester to spot eg. metastability or hardware integration issues.

In essence, the idea would be to feed the DUT's reference model the same input data as that specified in a `tap_lib.Test` and compare the reference model's output to the dump file generated from the in-circuit test execution. Ideally, we wanted to integrate the reference models into `tap_lib` to make this process as convenient as possible. We for example looked into the idea of passing the reference model as a parameter into the `tap_lib.Test` constructor. When executing the test, we could then run the reference model to automatically generate the expected output and compare this to the resulting dump file. After analyzing how these reference models were designed, it was concluded that such a strong `tap_lib` integration is most likely not possible. The issue is that there is no standard for the reference

model interface. Between projects, or even between modules in the same project, the file formats and parameter structures on which the reference models operate can look completely different from each other. This makes it impossible to design a standardized system for how `tap_lib` should interact with the models. For each project, there would have to be some degree of manual work to create a setup where the reference models and the TAP output can be compared. This might for example involve writing a script that translates the TAP input into a format that can be parsed by the corresponding reference model.

Still, there are some characteristic that the reference models do have in common with each other. Based on these, some features have been added to `tap_lib` to make it easier to synchronize the two systems. Importantly, there is usually no timing information in the reference input and output, and the data is instead often described as a list of two-dimensional vectors representing complex numbers with in-phase and quadrature components. This is in contrast with the `.vcd` dumpfiles that `tap_lib` generates, making direct comparisons between in-circuit measurements and reference output difficult — especially if one intends to design some automatic script for this.

To simplify this process, we have added support for a CSV-based dumpfile format. Contrary to the VCD dump, this contains no timing information, and the samples will simply be listed in the same order they were sampled. A separate csv-file is generated for each outpost, containing the corresponding samples. For complex outposts, the file will contain two columns representing the in-phase and quadrature components respectively. An illustrative comparison between the two dump formats can be seen in figure 3.10.



(a) VCD file. **(b)** CSV file.

Figure 3.10: CSV dump for one outpost compared to the joint VCD dump.

Chapter 4

Practical Application: Verifying the Xilinx FFT LogiCORE IP

In this chapter we will go through an example verification scenario where we document the steps a tester could take to verify the Fast Fourier Transform (FFT) LogiCORE IP by Xilinx [20]. To highlight the post-synthesis configurability of our test system, the DUT consists of two FFT modules connected together in a cascade, performing the forward and inverse FFT transforms, respectively. The modules are configured to use fixed point arithmetic, a transform size of 512 points, and a data sample precision of 16 bits. They use AXI handshakes for flow control.

Figure 4.1 shows the DUT together with the TAP that will be used to test it. Here, FFT and IFFT are the DUT blocks. The probes and outposts have been given names that can be used later for referencing. As shown in the figure, the TAP will consist of two outposts: `Edge` with two probes `A` and `C`, and `Middle` with one probe `B`. Configuring the TAP this way allow two test cases. Firstly, time-domain test data can be injected into `FFT` through probe `A` and the `FFT` output sampled through probe `B`. Secondly, frequency-domain test data can be injected into `IFFT` through probe `B` and the `IFFT` output sampled through probe `C`. The point here is that the two test cases can be run consecutively, where the dump file of the first test can be used as input for the second test. This gives strong signal observability without having to probe more than two points simultaneously.

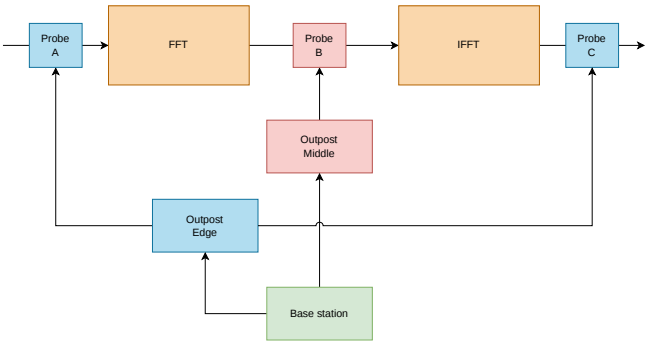


Figure 4.1: The DUT and TAP.

To specify this TAP configuration in `tap_lib`, we use the code shown in listing 4.1. The outposts are configured to perform throttled injections with AXI handshakes through the boolean `axi4` constructor parameter. The probes' `trigger_mask` parameters have all been set to zero as we want to utilize the `manual_trigger` inputs to sample only on valid AXI handshakes.

Listing 4.1: The TAP configuration.

```
tap: Tap = Tap (
  outposts = [
    Outpost(
      name="edge",
      memory_depth = 512,
      probes = [
        Probe("A", 32, 0b0),
        Probe("C", 32, 0b0),
      ],
      burst_duration=0,
      complex = True,
      axi4    = True
    ),
    Outpost(
      name="middle",
      memory_depth = 512,
      probes = [
        Probe("B", 32, 0b0),
      ],
      burst_duration=0,
      complex = True,
      axi4    = True
    )
  ],
  config = Config (
    timestamp_size                = 20,
    outpost_probe_pipeline_stages = 5,
    base_outpost_pipeline_stages  = 5
  )
)
```

After defining the desired TAP configuration in `tap_lib`, we can now call `Tap.write_verilog()` to generate the corresponding System Verilog TAP module and then manually instantiate this into the DUT module. As described in section 3.2.2, the TAP instantiation involves quite a lot of manual signal routing to make sure that the DUT and TAP are connected as desired. In this case, each probe will contain the `manual_trigger` port, six AXI ports (the master and slave pairs of READY, VALID, and LAST), and two data bus ports (`s_bus` and `m_bus`). All of these will have to be connected to the corresponding DUT signals. To achieve the desired sampling behavior, the `manual_trigger` signals will be mapped to the logical AND of `m_axis_ready` and `m_axis_valid` — making sure that the probes sample on successful AXI handshakes.

The next step is to define the two tests that were discussed earlier: (1) Injecting time domain data into the FFT block through probe `A`, we call this test `td_A`. (2) Injecting frequency domain data into IFFT through probe `B`, we call this test `fd_B`. The two tests are shown in listing 4.2 and 4.3. The test data that will be injected is specified in the two CSV files `td.csv` and `fd.csv`, containing complex reference signal samples in time domain and frequency domain respectively.

The samples are specified in 16-bit two's complement fixed point form, with the real and imaginary components split into two columns. A Python script has been written to generate synthetic periodic time domain signal samples for `td.csv`. These are shown in figure 4.2. If the FFT module works correctly, we expect to get these signals back after passing them through the FFT and IFFT blocks. The frequency domain samples in `fd.csv` come from the FFT output of `td_A`.

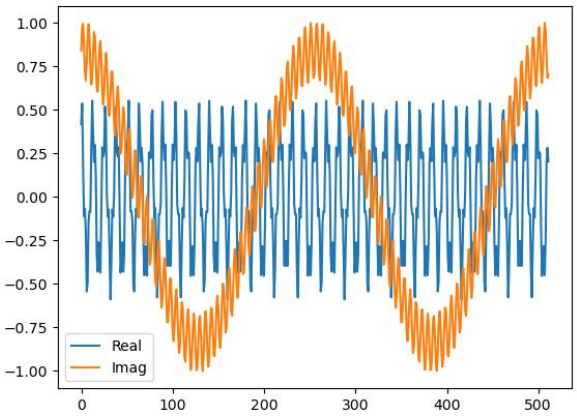


Figure 4.2: The synthetic input samples.

Listing 4.2: `td_A`.

```
td_A: Test = Test (
  name="tdA",

  targets = {
      Test.Target(
        tap.outposts.edge.probes.A,
        "td.csv"
      ),
      Test.Target(
        tap.outposts.middle.probes.B
      )
  },

  dump_path="dump",
  csv = True,
  tap = tap,
  duration= 1
)
```

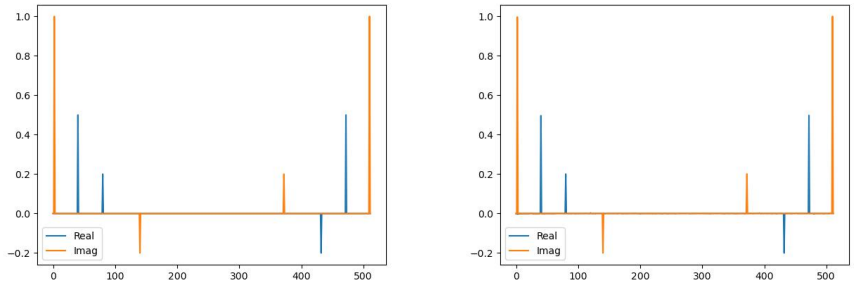
Listing 4.3: `fd_B`.

```
fd_B: Test = Test (
  name="fdB",

  targets = {
      Test.Target(
        tap.outposts.middle.probes.B,
        "fd.csv"
      ),
      Test.Target(
        tap.outposts.edge.probes.C
      )
  },

  dump_path="dump",
  csv = True,
  tap = tap,
  duration= 1
)
```

The forward FFT implementation from the SciPy library can be used as a reference model for verification [21]. Running the Python implementation on the synthetic signal samples gives a reference output for `td_A` that is compared to its dump file. In figure 4.3, the output of the SciPy FFT implementation is shown together with the `td_A` output. Clearly, the frequency bins match closely, and therefore one can conclude that the FFT block seems to behave correctly.



(a) SciPy reference. **(b)** TAP sample dump.

Figure 4.3: FFT outputs.

Figure 4.4 shows the output of `fd_B`. As expected, this closely resembles the original input samples from figure 4.2. There are some minor differences in phase, but one can assume that these come from rounding errors. In conclusion, our test system has been able to successfully verify the FFT LogicCORE IP by comparing the signal samples with a known good Python FFT implementation.

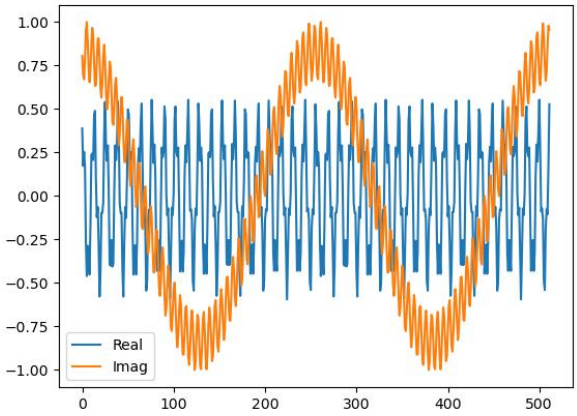


Figure 4.4: The time domain output of test `fd_B`.

Chapter 5
Evaluation

To evaluate our test system, it will be compared with Vivado's ILA. By identifying key differences in regards to performance, functionality, and ease-of-use we aim to get an understanding of the strengths and weaknesses of our system.

5.1 Methodology

The primary method of evaluation will be to instantiate either Vivado's ILA or our system on an FPGA together with a mock-up DUT. We will then use both systems to conduct the same (or as similar as possible) tests and evaluate the systems based on both quantifiable and qualitative metrics.

5.1.1 Metrics

Silicon overhead

The amount of FPGA resources a test system consumes has a great effect on its practicality. A resource intensive ELA might take up too much space to fit on the target FPGA together with a complex DUT. To measure our test system's silicon overhead, we will look at how the resource usage scales with the TAP configuration. The following variables will be studied:

i. The number of probes.

ii. The number of outposts.

iii. The sample signal width.

The exact silicon overhead introduced by the test systems will depend on multiple uncontrollable factors relating to Vivado's implementation algorithms, so these measurements will not be completely generalizable and should be seen as pointers.

Observability and controllability

What DUT signals the test systems can monitor and control. What are the limitations and are there situations where extra steps need to be taken to achieve the desired functionality?

I/O usage

In highly utilized FPGA's, there could be a limited amount of I/O resources to spare for a test system. This means that to make an ELA applicable to as many projects as possible, one might want to minimize the number of I/O pins it uses.

Ease of use

A qualitative analysis of how easy and intuitive the test systems are to use for a tester. We will compare the steps a tester needs to take to perform similar tests on the two systems and discuss which process seems most technically demanding and error-prone.

Equipment requirements

What, if any, special testing hardware does a tester need to perform tests on a DUT using the test systems. Specialized hardware dependencies will affect whether an ELA is practically usable in-field by technicians or if it is limited to lab-use. Such dependencies can also negatively impact how easy an ELA is to adopt in a large workforce.

Effect on DUT functionality

To what extent the ELA's impact a DUT's normal functions. A completely non-intrusive ELA could for example be instantiated in system deployment, possibly allowing convenient in-field verification without re-synthesis.

5.1.2 The DUT

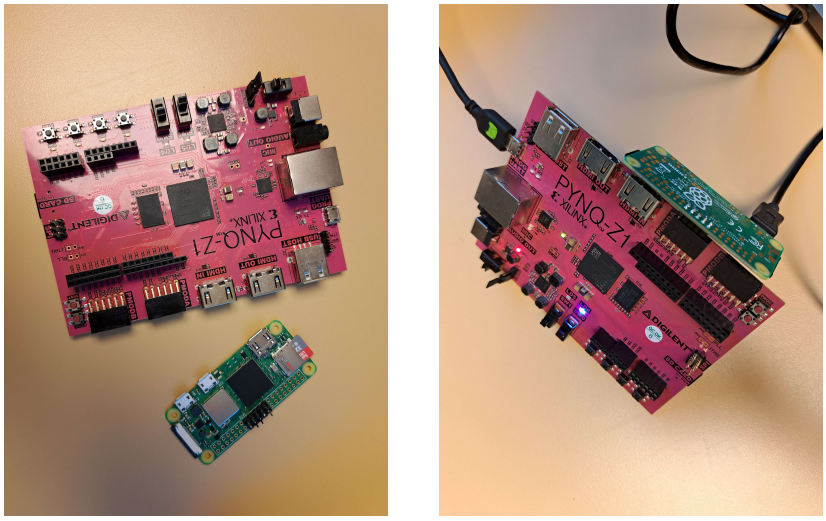
The DUT that we will be using for test system evaluation is a pipelined fast fourier transform (FFT) module published on Github under MIT license by user nanamake [22]. We used this instead of the LogiCORE IP from chapter 4 as it was easier to work with and configure. Pipelined means in this case that the FFT computation is split into multiple submodules that can work in parallel, resulting in a higher throughput. The FFT module works in fixed point complex numbers of a parameterized width and has the following signals:

- **di_en**, data in enable.
- **di_re**, data in real part.
- **di_im**, data in imaginary part.
- **do_en**, data out enable.
- **do_re**, data out real part.
- **do_im**, data out imaginary part.

As the ILA does not support test data injection, we will use two counter modules to generate arbitrary complex numbers for FFT input. To ensure that the synthesizer does not optimize away FFT logic, the sum of `do_re` and `do_im` is routed to arbitrary FPGA I/O.

5.2 Hardware setup

For system evaluation we will be using a PYNQ-Z1 FPGA development board with the Raspberry PI zero 2 w connected through one of the PMOD ports. The hardware is shown in figure 5.1



(a) The Raspberry PI and PYNQ-Z1. **(b)** The PMOD connection.

Figure 5.1: The hardware used for evaluating our test system.

5.3 Tests

A series of test setups have been designed. These will be used to take measurements of the two test systems according to the metrics defined in section 5.1.1. The test setups specify what signals to monitor in the DUT and are shown in table 5.1. For all tests, the TAP will be configured with a 1024 bit memory depth, a 32 bit timestamp size, and its probes with a full trigger mask (meaning that all bit changes in the intercepted bus will be sampled).

Table 5.1: The monitor tests.

Index	FFT width	Signal monitor
1	8	`do_re`
2	8	`do_re`, `do_im`
3	8	`do_re`, `do_im`, `di_re`
4	16	`do_re`
5	16	`do_re`, `do_im`
6	16	`do_re`, `do_im`, `di_re`

To explore the injection capabilities of our test system a final injection test is specified in table 5.2. Here, we will by-pass the counter-based complex number generator and insert custom FFT input through `tap_lib` instead.

Table 5.2: The injection test.

Index	FFT width	Signal injection	Signal monitor
7	16	`di_re`, `di_im`	`do_re`, `do_im`

Chapter 6
Results

The chapter describes the quantitative results of our evaluation. For each test specified in section 5.3, we will show the silicon consumption of our test system and compare this to an ILA instance monitoring the same signals. We will also show some examples of the resulting signal trace output from the two test systems.

Importantly, our test system's runtime signal selection capabilities means that there are more than one possible configuration of the TAP that can be used to achieve the signal monitoring specified in the tests. If it is necessary to monitor all signals simultaneously, one would need to assign one outpost to each signal. In many cases, however, it could be sufficient to use only a single outpost and look at one signal at a time — re-running the test with a new probe target between traces.

To highlight this capability, each monitor test from section 5.3 was run four times: (1) No ELA instantiated, a baseline. (2) Using Vivado's ILA. (3) Using our test system with a single outpost. (4) Using our test system with one outpost for each probe. The monitor test results are shown in table 6.1. The resulting waveforms for test 3 and 6 are shown in figure 6.1.

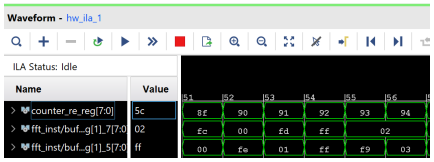
The injection test result is shown in table 6.2 and the waveform in figure 6.2. Here the TAP was configured to use one outpost for each of the four signals. Two for test data injection in `di_re` and `di_im` and two for sampling `do_re` and `do_im`.

Table 6.1: ELA resource consumption.

ELA	FFT_BITS	OUTPOSTS	PROBES	LUT	LUTRAM	FF	BRAM	DSP	IO	BUFG
Baseline										
NONE	8	0	0	1120	40	356	1.0	0	6	1
NONE	16	0	0	1020	80	593	0.0	8	6	1
Test 1										
ILA	8	0	1	2163	128	2147	1.5	0	6	2
OURS	8	1	1	1366	40	618	2.5	0	11	1
Test 2										
ILA	8	0	2	2223	137	2267	1.5	0	6	2
OURS	8	1	2	1384	40	649	2.5	0	11	1
OURS	8	2	2	1535	40	770	4.0	0	11	1
Test 3										
ILA	8	0	3	2276	146	2362	2.0	0	6	2
OURS	8	1	3	1390	40	685	2.5	0	11	1
OURS	8	3	3	1735	40	929	5.5	0	11	1
Test 4										
ILA	16	0	1	2081	176	2457	0.5	8	6	2
OURS	16	1	1	1277	80	899	2.5	8	11	1
Test 5										
ILA	16	0	2	2161	193	2601	1.0	8	6	2
OURS	16	1	2	1333	80	990	1.5	8	11	1
OURS	16	2	2	1498	80	1163	4.0	8	11	1
Test 6										
ILA	16	0	3	2224	210	2744	1.5	8	6	2
OURS	16	1	3	1357	80	1058	1.5	8	11	1
OURS	16	3	3	1732	80	1306	5.5	8	11	1

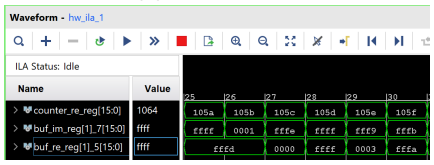
Table 6.2: The injection test results.

ELA	FFT_BITS	OUTPOSTS	PROBES	LUT	LUTRAM	FF	BRAM	DSP	IO	BUFG
OURS	16	4	4	1981	80	1512	7	8	11	1



(a) Test 3 - ILA

(b) Test 3 - ours



(c) Test 6 - ILA

(d) Test 6 - ours

Figure 6.1: The waveforms for test 3 and 6.

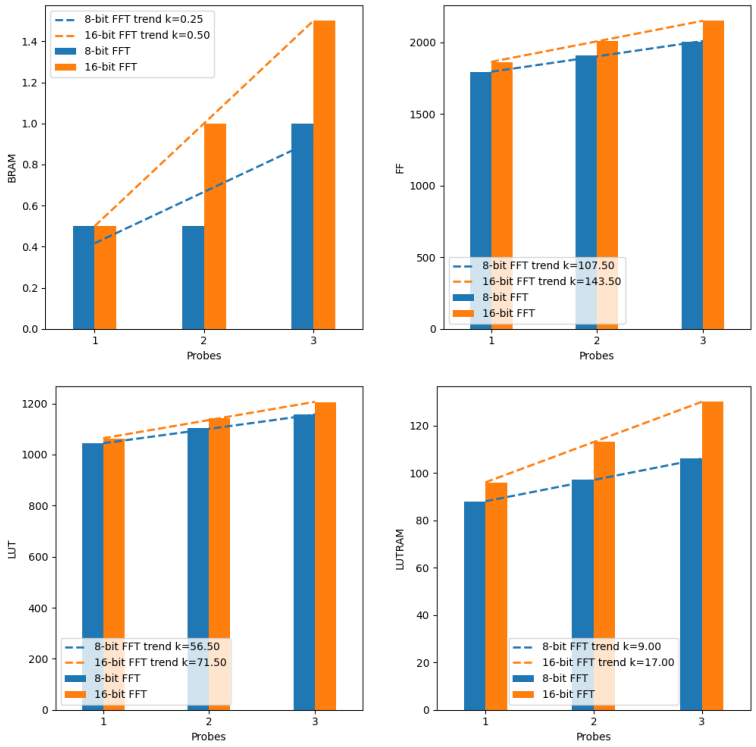
Figure 6.2: The waveform for the injection test.

6.1 Silicon overhead scaling analysis

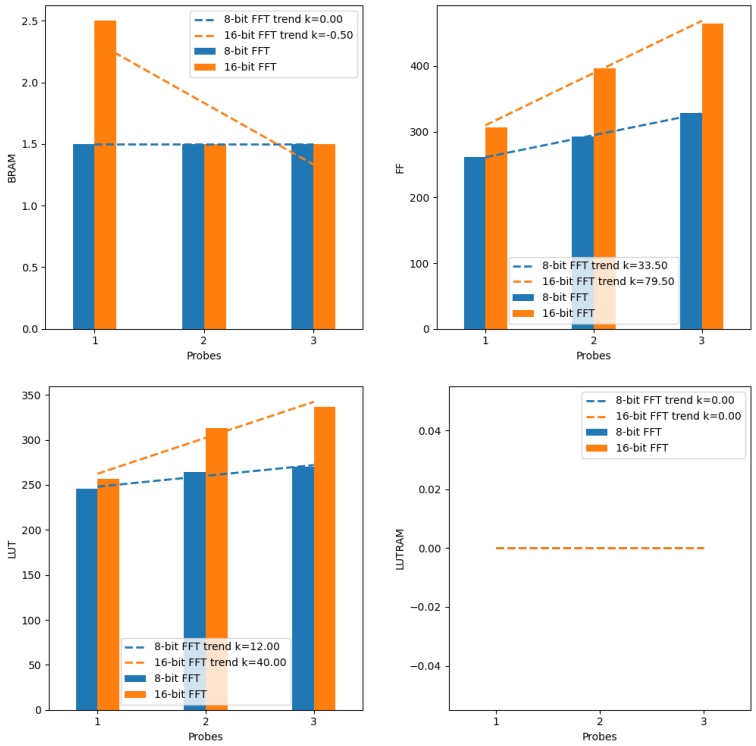
From the measurements shown in table 6.1, we can analyze how the silicon overhead scales with configuration for our test system and for Vivado's ILA. We will be using the baseline DUT measurements to get an approximation of how much extra silicon usage the ELA's introduce. As described in section 5.1.1, ELA overhead as a function of the number of probes, outposts, and the sample signal width will be studied. Linear regression will be used to identify resource trends for each variable. For Vivado's ILA, the number of signals selected pre-synthesis for monitoring will be treated as our test system's probe count variable. The outpost count variable is not applicable to the ILA.

6.1.1 The number of probes

The probes' effect on the silicon overhead can be seen by comparing the results for the single outpost configurations of tests 1-3 (8-bit FFT) and tests 4-6 (16-bit FFT) against each other. The comparison is shown in figure 6.3.



(a) ILA.



(b) Our test system.

Figure 6.3: ELA silicon overhead as a function of the number of probe points.

6.1.2 The number of outposts

The outposts' effect on the silicon overhead can be seen by comparing the results for the single outpost configurations against the multi-outpost configurations of tests 2-3 and tests 5-6. The comparison is shown in figure 6.4. In the graphs, the bar colors represent a combination of the number of FFT-bits (denoted `b`), and the number of signal probes (denoted `p`). The combination `16b, 3p` means for example 16-bit FFT with three probes — representing test 5.

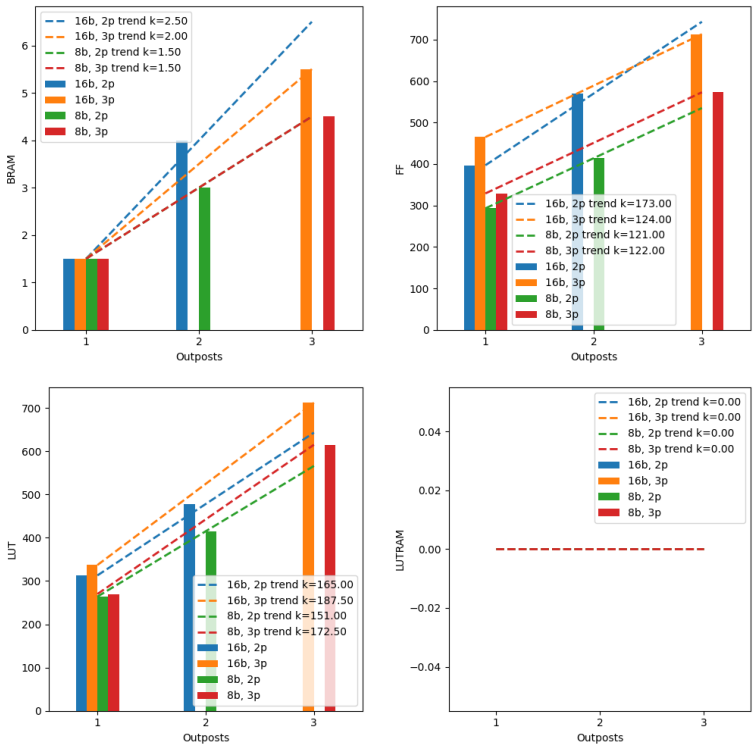
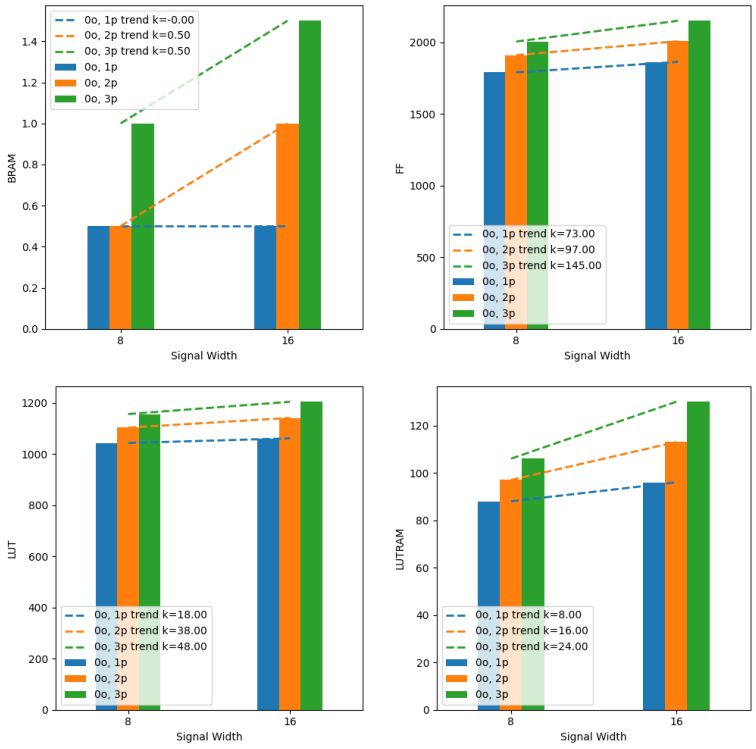


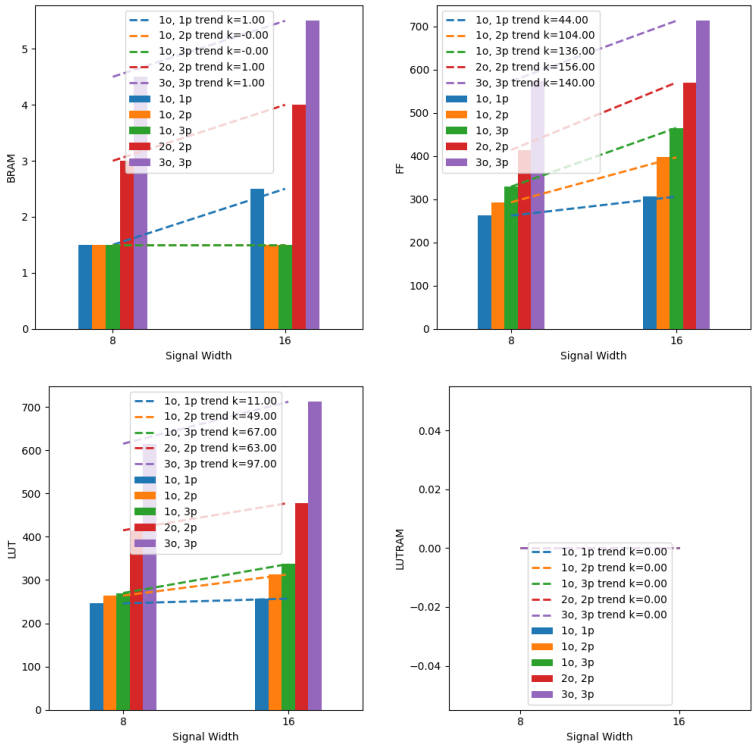
Figure 6.4: Our test system's silicon overhead as a function of outpost count.

6.1.3 The sample signal width

The effect of the monitored signals' width on silicon overhead. We compare outpost (`o`) and probe count (`p`) combinations between 8-bit and 16-bit FFT. The comparison is shown in figure 6.5. The bar marked `2o, 2p` represent for example the comparison between the two outpost configuration of test 2 against the two outpost configuration of test 5. Note that as the outpost concept is not applicable for Vivado's ILA, there is only one configuration for each test and this is marked as containing zero outposts.



(a) ILA.



(b) Our test system.

Figure 6.5: ELA silicon overhead as a function of the sample signal width.

6.1.4 I/O usage

As shown in table 6.1 our test system uses 5 additional I/O ports as opposed to Vivado's ILA.

6.1.5 Equipment requirements

Our test system is controlled through an SPI connection, requiring an external PC capable of establishing this. Vivado's ILA can run through the same cable used for FPGA programming.

6.1.6 Effect on DUT functionality

Neither our system nor Vivado's ILA have any effect on DUT functionality when not operational.

Chapter 7
Discussion

The goal of this master's thesis project was to develop an in-circuit FPGA debugging system configured and controlled through Python. This goal has been achieved, we have a complete system that has been verified to work in real debugging scenarios and that supports the required functionality specified in section 1.2.2. Extensive prototyping, testing, and debugging were needed to get to this point and we believe that the final product can serve as a strong proof-of-concept for how a Python-based in-circuit debugger can be designed.

The test system involves a large amount of sub-systems working in tandem to execute the debugging tasks, designing this complex architecture proved difficult and we found that small design choices quite often turned out to have large consequences down the line. When we started working on this project we had a very limited understanding of in-circuit FGPA debugging, we had no experience working with ELAs and therefore had a hard time figuring out what features were important and how we would want our system to be presented and interacted with by a tester. As a result of this, we had to perform quite extensive re-designs of several components during development as we had realized too late that some incorrect assumption had been made or important concepts misunderstood. With the evaluation results in hand, however, we can conclude that the end result is a competitively viable option with strong use-case arguments. Still, it has become clear to us that there are improvements to be made in possible future iterations of the design.

The following sections will cover the evaluation metrics defined in section 5.1.1 and we will discuss some key observations that we have made when evaluating our system.

7.1 Silicon overhead

For all resources except BRAM (more on this later), our test system was significantly cheaper to implement in all configurations covered in our evaluation. We consistently measured resource overheads as small as 50% or even 25% of the ILA counterpart. This can for example be seen in the LUT consumption shown in figure 6.3, where our system seemed to use around 300 LUTs and the ILA 1200. As all configurations covered in our evaluation were rather minimal — the trend curves in section 6.1, derived from linear regression, paint a better picture of the

resource characteristics than these absolute values. When taking the trend curves into account, it becomes obvious that our system does not outperform the ILA in all situations. Which of the two systems are more resource efficient seems to vary heavily depending on configuration.

On the one hand, the benefits of our system's runtime signal selection capabilities become apparent when the amount of outposts is kept constant. The more probe points a tester adds, the more resources our system saves compared to the ILA. This is shown in figure 6.3, where the ILA's trend curves are steeper than ours in every single metric. On the other hand, in situations where it is necessary to monitor a large number of probe points simultaneously, our systems resource scaling significantly worsens. In these situations, a tester would need to increase the number of outposts and from figure 6.4 one can see that this is expensive. In our system, to add a simultaneously observable signal one needs to add both a probe and an outpost — the cost of which can be seen as the sum of the resource curves in figure 6.4 and figure 6.3. Our system seems to scale around two to four times worse in these cases, depending on the resource considered.

Besides the general trends discussed above, there are some interesting differences in regards to memory use that we think should be highlighted. Firstly, our system does not seem to use any LUT RAM. This resource was never synthesized in the evaluation configurations and presumably this could bring some advantage compared to the ILA if resource availability is limited. Secondly, and more importantly, the measurements indicate that our system has significantly higher BRAM usage compared to the ILA. Both the measured absolute values and trends are worse. By comparing figure 6.4 and figure 6.3 we can see that our BRAM usage seems to increase around four times faster. One possible reason for this is that our solution does not fully consider the hardware design of the BRAM modules. A BRAM address corresponds to a physical location in memory and as such is not fully configurable, the number of bits that can be stored per address and the number of unique addresses is set by hardware constraints. By not considering these constraints, our system most likely severely underutilizes the BRAM hardware. We might, for example, only use a fraction of the available bits per address due to small sample sizes or use only a few of the addresses available. One way to mitigate this problem could be to implement a more centralized memory system, communicating with the probes directly and avoiding the outpost middle man. A centralized system would have more complete information about total memory usage and might therefore be able to make better decisions, allowing for more efficient BRAM utilization through resource sharing.

Finally, it is important to consider how generalizable our measurements are. There are two main factors that threaten their validity and that limit how certain we can be of conclusions drawn from them. Firstly, the small scope of our tests. To make the evaluation process more manageable, we had to limit the number of tests to run, and because of parameter constraints in the FFT module we could only test up to 16-bit sample widths. Secondly, the uncontrollable nature of Vivado's synthesis engine. The synthesis engine is in many aspects a black box. We do not know how it chooses to implement the logic in our System Verilog modules nor how this implementation could change depending on external factors such as synthesis settings and Vivado updates. Importantly, our simple way to calculate ELA

overhead by subtracting the baseline resource consumption is inherently flawed because of this. We can not know for certain that the DUT implementation stays the same between different test runs, Vivado might find a way to co-optimize it with our test system somehow. This can for example be seen in figure 6.3, where the BRAM usage decreases between tests in the 16-bit FFT case, contrary to what is expected.

7.2 Observability and controllability

Our system offers both benefits and drawbacks compared to the ILA in regards to observability and controllability. Neither ELA has any theoretical limitations in what signals can be monitor, a tester is free to specify any number of probe points in the DUT. In practice, however, the resource consumption must be considered as it will create an upper bound for observability. As discussed in section 7.1 — which of the two ELAs is more efficient, thereby allowing greater observability, will depend on configuration. Importantly, as long as simultaneous signal monitoring is avoided, our test system is significantly cheaper to implement than the ILA. We can add more signal probes for the same resources, leading to higher observability in this case.

The ILA's advanced and highly customizable triggering system should also be taken into account. In complex verification scenarios where a large amount of signals are monitored and analyzed, defining a fine-grained filter for when sampling should occur might be vital. For our TAP, a tester could utilize the probes' `manual_trigger` input to in some degree mimic the ILA's trigger system, but the process would be cumbersome and still not offer the same set of features. One of our reasons behind the probe's simple triggering mechanism was to minimize its resource consumption, which seemingly paid off.

The TAP's unique injection capability makes it possible for a tester to control the state of the DUT, which unlocks use cases that are not possible with the ILA. The unit test system in `tap_lib` is for example reliant on this feature.

7.3 I/O usage

Vivado's ILA has a key benefit here as it can be run through the same JTAG infrastructure that is already used to program the FPGA. The five-wire SPI communication that our system uses could be too expensive for highly utilized FPGAs. If this turns out to be an issue, however, there are multiple ways to reduce the I/O consumption of our design. One could for example either re-write the SPI driver to follow a standard two-wire SPI specification instead, and utilize polling instead of a dedicated flow control pin, or move away from SPI completely and run the communication through e.g. Ethernet as was done in [7].

7.4 Ease of use

It is not possible to make any objective arguments for which of the two systems is easier to use for a tester. The verification workflows bear little resemblance to each other and there are important differences in what features they support. Still, the GUI-based configuration and viewing process provided by ILA with its robust and seamless Vivado integration is, in our opinion, hard to compete with. We have made great efforts to make our system easy to work with, hiding complex configuration details behind high level API calls and utilizing automatic System Verilog generation, but compared to the ILA the verification process still involves significantly more manual, and sometimes unintuitive, steps.

Xilinx has managed to get ILA to a point where working with it is in many regards very similar to working with a simulator. To specify what signals to monitor, the tester is simply presented with an exhaustive list of all signals in the design where they can freely select those that are of interest. To set parameters such as trigger conditions, memory depths etc., the tester only needs to follow a guided chain of GUI windows with built in value ranges and automatic assertions. Importantly, at no step in the process do you have to make any manual modifications to either constraints files or modules to accommodate the ILA. Based on the GUI-specified configurations, Vivado automatically instantiates the ILA, connects it to the DUT, and re-writes the constraints file.

In contrast to this, the script-based approach of our system demands more from the user. The TAP configuration as described in section 3.2.1 requires one to carefully read and follow library documentation to ensure that it both compiles and accurately describes the intended functionality. Then, actually integrating this into the DUT's project comes with further annoyances. Every probe point needs to be connected to the correct place in the DUT, a top file either modified or created, the constraints file changed, and potentially signals rerouted. Each of these steps introduces additional complexity, increasing the risk of erroneous configurations.

Still, there are clear benefits to our design in regard to reproducibility. The way tests are defined through scripts and our capability to inject test data allows a tester to perform fully deterministic and well-defined system tests that require no user input besides script execution. A tester can create a fully reproducible test suite consisting of any number of such tests, enabling automated, fast, large-scale, and complex verification. Reproducing this in the ILA is not possible, especially because of the lack of runtime signal selection necessitating re-synthesis between tests.

7.5 Equipment requirements

To interact with our test system, the tester needs to establish a SPI connection between the FPGA and a PC running the `tap_lib` Python library. For our project work, this PC has been in the form of a Raspberry PI — so our setup essentially involved two computers, one for reprogramming the FPGA through Vivado and one for controlling the TAP. There is however no architectural reason for doing it this way and one could instead use a single computer for both of these tasks. The reason we chose to use a Raspberry PI was because of its GPIO, the pins being controllable from Python libraries made it easy for us to set up the SPI communication.

7.6 Effect on DUT functionality

As long as no timing violations appear because of the increased FPGA resource consumption, neither our system nor the ILA is destructive. The DUT will keep all of its functionality despite the signal probe connections and it should be able to operate as normal. As a result of this, a possible use case could for example be to add a TAP to final production-ready FPGA designs, allowing a technician to utilize the test system in-field to e.g. identify hardware malfunctions.

Chapter 8
Conclusion

The point of this master's thesis was to develop an in-circuit FPGA test system. To make it easy to integrate into a pre-existing Python-based verification workflow, the test system needed to be controlled through a Python library. It was also important that it was platform and vendor agnostic to make it applicable to as many projects and situations as possible. To measure our success in building this test system, we compared its design and user-experience against Xilinx Vivado's Integrated Logic Analyzer (ILA). From the discussion in chapter 7 we can draw some conclusions about the project results.

Firstly, our measurements indicate that our system supports resource efficient configurations through its runtime signal selection capabilities. In testing scenarios that do not require extensive simultaneous signal monitoring, it is significantly outperforming the ILA in terms of FPGA silicon consumption. Our system also has a much smaller base cost than the ILA, in small configurations where a tester only wants to look at a handful of signals, it can cost as little as 25% of a corresponding ILA implementation. Where our system does not perform as well, however, is in situations where a tester needs to monitor a lot of simultaneous signals. The TAP scales poorly with outpost count and the ILA will, despite its higher base cost, eventually outperform it. This issue is especially apparent in BRAM consumption, as it seems like the memory controllers in our outposts are underutilizing this hardware component.

Secondly, even though efforts were made to make our system easy to work with, we conclude that the robust Vivado IDE integration of the ILA gives a smoother and more intuitive user experience for a tester. Our system is built to be interacted with through a command line while the ILA is GUI-based, one could argue that this is an issue by itself but the approach also provide clear benefits in terms of e.g. test repeatability as the Python library is completely scriptable. The larger problem with our system is rather the amount of manual steps one needs to take to use it. Importantly, the process involves some manual modifications of the DUT modules where a tester needs to instantiate the TAP and connect its signals, these steps are completely automated in the ILA.

In summary, the result of our project is a functioning in-circuit test system that that contains all components necessary for practical FPGA design verification. It is proven to be resource efficient compared to commercial solutions and includes additional functionality in the form of test data injections. We believe that our project has served its purpose well as a design concept and could in its current state serve as a real tool in a verification workflow. Still, there are many design choices that could be reevaluated and there are some clear improvements that could be made, especially in regards to user experience and project integration aspects.

References

[1] J. J. Rodriguez Andina, E. de la Torre Arnaz, and M. D. Valdes Peña, *FPGAs: Fundamentals, Advanced Features, and Applications in Industrial Electronics*. CRC Press, 2017.

[2] J. M. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, and G. Moroney, "Error cost escalation through the project life cycle," National Aeronautics and Space Administration (NASA), Technical Report JSC-CN-8435, 2004. [Online]. Available: https://ntrs.nasa.gov/api/citations/20100036670/downloads/20100036670.pdf

[3] K. Kim, B. Kang, D. Kim, S. Lee, J. Shin, and H. Shin, "Low-cost design for repair with circuit partitioning," in *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010, pp. 259–261.

[4] H. Feng, W. Li, S. Wang, J. Zhou, Y. Pang, C. Tian, and Y. Zhang, "A framework of embedded logic analyzer for fpga," in *2022 IEEE 5th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, vol. 5, 2022, pp. 609–615.

[5] S. Asaad, R. Bellofatto, B. Brezzo, C. Haymes, M. Kapur, B. Parker, T. Roewer, P. Saha, T. Takken, and J. Tierno, "A cycle-accurate, cycle-reproducible multi-fpga system for accelerating multi-core processor simulation," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 153–162. [Online]. Available: https://doi.org/10.1145/2145694.2145720

[6] Xilinx, Inc., *Integrated Logic Analyzer (ILA) v2.0*, AMD, 2012. [Online]. Available: https://docs.amd.com/v/u/en-US/ds875-ila

[7] G. Fiala, T. Scheipel, W. Neuwirth, and M. Baunach, "Fpga-based debugging with dynamic signal selection at run-time," *Automated Software Engineering*, 2020. [Online]. Available: https://ceur-ws.org/Vol-2581/ase2020paper3.pdf

[8] S. Attia and V. Betz, "Stop and Look: A Novel Checkpointing and Debugging Flow for FPGAs," *IEEE Transactions on Computers*, 2022.

[9] D. Koch, C. Haubelt, and J. Teich, "Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation," in *Proceedings of the 2007*

ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays, ser. FPGA '07, 2007.

[10] Motorola, Inc., "Spi block guide, version v04.01," Motorola, Inc. / Freescale Semiconductor, Inc., Technical Report, Jul 2004, original Release: 21 Jan 2000; Revised: 14 Jul 2004. [Online]. Available: https://web.pa.msu.edu/people/edmunds/Disco_Kraken/SPI_Documents/motorola_freescale_nxp_spi_manual_2000.pdf

[11] H. S. Poornima and C. Nagaraju, "Functional verification of clock domain crossing in register transfer level," in *2023 International Conference on Recent Trends in Electronics and Communication (ICRTEC)*, 2023.

[12] Linux Kernel Documentation, "Spi userspace api," https://www.kernel.org/doc/html/latest/spi/spidev.html, accessed: 2026-02-24.

[13] doceme, "py-spidev: Python bindings for spi using linux spidev," https://github.com/doceme/py-spidev, accessed: 2026-02-24.

[14] Raspberry Pi Ltd, "Raspberry pi computer hardware," https://www.raspberrypi.com/documentation/computers/raspberry-pi.html, accessed 2026-02-24.

[15] B. Nutall and D. Jones, "gpiozero," https://gpiozero.readthedocs.io/en/stable/, accessed 2026-02-24.

[16] B.-C. C. Lai and J.-L. Lin, "Efficient designs of multiported memory on fpga," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 1, pp. 139–150, 2017.

[17] "Ieee standard hardware description language based on the verilog(r) hardware description language," *IEEE Std 1364-1995*, pp. 1–688, 1996.

[18] L. University, "Surfer," 2026, [Accessed 12-03-2026]. [Online]. Available: https://surfer-project.org/

[19] Arm Limited, *AMBA AXI Protocol Specification: AXI3*, issue l ed., Arm Limited, 2025. [Online]. Available: https://developer.arm.com/documentation/ihi0022/latest/

[20] Xilinx, Inc., *Fast Fourier Transform v9.1*, AMD, 2022. [Online]. Available: https://www.xilinx.com/support/documents/ip_documentation/xfft/v9_1/pg109-xfft.pdf

[21] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[22] nanamake, "r22sdf: Pipeline FFT Implementation in Verilog HDL," https://github.com/nanamake/r22sdf, 2019, accessed 2026-03-03.