Manta: An In-Situ Debugging Tool for Programmable Hardware

by

Fischer Jay Moseley

B.S., Electrical Science and Engineering, Physics
Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by:	Fischer Jay Moseley Department of Electrical Engineering and Computer Science May 19, 2023
Certified by:	Joseph D. Steinmeyer Senior Lecturer Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Manta: An In-Situ Debugging Tool for Programmable Hardware

by

Fischer Jay Moseley

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Designing and debugging digital hardware has traditionally used vendor-provided tools, which are large and platform-constrained. Designers expend considerable effort accommodating toolchains for Field Programmable Gate Array (FPGA) development, which include utilities for debugging logic on the FPGA itself. As an alternative, this work proposes Manta, a lightweight, modular, platform-independent, and intuitive tool for debugging digital logic on FPGAs. Manta is designed to supplement vendor tools, and includes a logic analyzer, block memory interface, and the ability to measure and control individual signals on the FPGA. These tools are shown to build faster and consume fewer on-chip resources than equivalent vendor offerings, without any restrictions on chip family or vendor. Ethernet and UART interfaces provide convenient and high bandwidth communication between the host machine and target FPGA, and an extensible Python API allows for easy development of custom applications. This complete system produces an accessible and equitable FPGA development experience for use in educational, professional, and hobbyist environments alike.

Thesis Supervisor: Joseph D. Steinmeyer
Title: Senior Lecturer

Acknowledgments

This thesis wouldn't have been possible without:

- The students I've taught across 6.002/6.2000, 6.111/6.205, 6.S092/EC.S03, and 6.900 over the last five years. You're what makes MIT special, and why I wrote this thesis (I hope you find it useful!). You've made 38-500/600 feel like home to me, and thanks for letting me derail whatever staff meeting or 6.115 checkoff I happened to walk in on. Don't ever lose your spunk, and keep sticking dead IR2125s into the ceiling tiles.

- Joe Steinmeyer, for the immense freedom granted with this thesis, and for the willingness to respond to slack messages at hours too late to divulge here. The genuine care you show for students something I look up to.

- Aditya Mehrotra, for teaching me that good design happens first in your heart, and is justified with your head. Thanks for keeping me focused through the years on the people we're trying to help, and keeping me from getting sidetracked on much else. I'm excited to see where the in-can-decent light within your heart shines.

- Jay Lang, for bringing a systems-first perspective to pretty much everything in sight, and for fighting unnecessary abstractions in systems design with a passion that could only be described as *zealous*.

- Daniel Klahn, for being both my first friend at MIT, and a fabulous teaching partner during the inaugural offering of 6.9000. We've yet to get ourselves into any trouble we can't get out of.

- MIT Formula SAE, for teaching me that no matter how cool the project is or how tight you think the deadline is, engineering is always about people, and it's them who matter most.

- Simmons Hall, and the Spongeifeareans inside. Thank you for giving me just as many windows as friends - and sorry for all the skid marks, dents, and suspiciously placed blue paint and plaster.
- Gene Fatton, for teaching me things that it took getting an MIT education to fully appreciate.
- Collin Turbert, for giving me way more responsibility than I deserved just to see what I could do with it.
- Team Pettner, for endless love, support, and paninis from day one.
- Bee, for always hearing my rants, having my back, and making time for me. I could not have made it through the last five years without you.

Finally, all mistakes in this thesis are mine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

MIT's *Digital Systems Laboratory* (6.205/6.111) course has long been a staple of the Institute's undergraduate electrical engineering curriculum since its introduction in 1968 as 6.711.[24] In the course, students build intuition for digital logic through guided laboratory exercises for the first half of the semester, before implementing a final project of their own design in the latter half.

The course underwent major restructurings in the late 1980s and early 2000s. Prior to then, the digital logic in lab exercises and final projects was implemented with discrete integrated circuits (ICs) assembled on a breadboard. This suited the class well as it reflected the way nearly all digital logic was designed at the time, but the growing adoption of programmable logic in industry prompted the course to be rewritten in the late 1980s. The new course targeted Programmable Array Logic (PAL) devices, until the course was rewritten again for more modern Field-Programmable Gate Arrays (FPGAs) in 2002.

The transition to programmable logic was an excellent change for the course. Logic was specified with source code, which was easier to inspect, debug, modify, and maintain than a bundle of jumper wires on a breadboard. This allowed for greater attention to be placed on learning and using the design principles taught in the course - and less on exorcising any loose wires before a lab checkoff.

However, this required the usage of proprietary EDA tools from chip vendors to implement the desired digital logic, which in the case of the FPGAs presently used, is Xilinx. These tools were run on dedicated machines in the lab, and were responsible for putting students' logic on their FPGAs, carrying their designs through synthesis, elaboration, implementation, placement, and routing. These processes were time-consuming, and the tools to perform them were bulky, platform-dependent, and offered an unintuitive user interface. Despite this, patiently waiting for a build to finish was greatly preferable to fiercely debugging a discrete implementation on a breadboard. A clunky tool was a small price to pay for the magic of programmable logic.

Fast-forwarding to the present day, the clunkiness of these tools has now become a distraction from the pedagogical goals of the course - much like hunting down loose wires was twenty years ago. The current iteration of Xilinx's design tools (now called *Vivado*) are still bulky and platform-constrained. Vivado's binaries exceed 100GB in size once installed, and only support Windows and Linux hosts with x86 processors. This leaves the toolchain inaccessible to students with either MacOS devices or low-spec machines. Historically the course has mitigated this by providing virtual machines for MacOS users, and dedicated machines in lab for students without the disk space to spare.

Neither of these solutions are convenient, or provide an experience equal to having the tools natively installed. The inequitably reduces the educational experience for those without the right hardware, a problem that the rising popularity of ARM processors has exacerbated. Students who own Macintoshes with Apple Silicon would need to create both a virtualized **and** emulated environment for over 100GB of EDA tools. Including this many compatibility layers does not produce a usable experience on anything but the highest-end Apple Silicon.

In response, at the beginning of the Fall 2022 semester the course staff repurposed the fleet of dedicated machines in lab into a distributed build system. Each machine became a worker in the pool, with its own set of native Vivado binaries installed. Students were provided a script (called `lab-bc`) that copied their source code to a

worker over SSH, built their code, and sent back a bitstream. This script was written in Python, enabling any machine with a SSH client and Python interpreter to become suitable for FPGA development.

The particular implementation was not perfect - it left some robustness and security to be desired - but it worked *beautifully*. Students with computing environments not natively supported by Vivado were able to make significant progress on their projects outside of lab. In the opinion of the course staff, this was responsible for a noticeable improvement in the quality of student projects.

Although most development work was done in simulation or with `lab-bc`, most projects at some point required the use of debugging tools only available through a native install of Vivado. Student designs would work flawlessly in simulation, but failed when flashed to a FPGA. To give a few examples from the Fall 2022 semester:

- A Reduced Media-Independent Interface (RMII) Ethernet controller that would place N bytes of data on the wire, only to have $N + 1$ bytes of data arrive on the receiving device.
- A Secure Digital (SD) Card controller that would request to read a 512-byte block from a microSD card, only to have the read buffer hang while servicing the request.
- A Xilinx-provided Double Data Rate (DDR) memory controller that would never finish initializing the Dynamic Random-Access Memory (DRAM) connected to the FGPA.

These issues were debugged with Vivado's Integrated Logic Analyzer (ILA) and Virtual IO (VIO) utilities. Both require a native Vivado install on the development machine - making them fundamentally incompatible with `lab-bc`. As a result, most students were back to using dedicated lab machines by the end of the semester, negating the pedagogical benefits of `lab-bc`.

In addition to being inconvenient and unequitable, debugging with an ILA or VIO on a dedicated lab machine was an overwhelming experience for most 6.205

students. Bringing up an ILA or VIO was only necessary during the latter half of the course, where students were working on their own projects. At this point, students were comfortable with a FPGA development workflow consisting of their preferred text editor, and a few staff-provided scripts for simulation, build, and upload. This meant that the process of opening Vivado's GUI, importing source code, navigating the IP configuration, and using the ILA and VIO was a completely new process, and students often became completely lost in it - even with a member of the course staff walking them through it. This consequence is particularly undesirable, as students resorting to an ILA or VIO had usually expended considerable effort debugging their design any other way they could, meaning most of their energy had been expended by the time they began debugging. This left little room for discussions of how to fix whatever issues were discovered, or creating a lasting understanding of how to use the debugging tools in the future. In short, using the vendor-provided tools exhausted whatever energy the students had, making the experience of developing their hardware exhausting, not enlightening.

Given the pedagogical benefits provided by equitable access to EDA tools via a cross-platform build client, it is believed that a cross-platform debugging utility would further increase the educational value of the course and provide a more equitable and enlightening learning experience for all.

The development of such a tool is the objective of this thesis.

1.2 Proposed Solution

Figure 1-1: The Manta logo. Canonically, the manta ray depicted is named Raymond, for obvious reasons.

This thesis proposes *Manta*, an in-situ tool for debugging programmable hardware, as sufficient for meeting the pedagogical needs of the course. Fundamentally, its goals are:

- To debug user-designed logic in a manner familiar, intuitive, and useful to both experienced hardware engineers as well as 6.205 students and staff.
- To be as simple as possible, both in user interface and system architecture. Using the tool should place as little cognitive load on the user as possible.
- To complement the vendor-provided debugging tools currently used. Manta should require fewer FPGA resources, build faster, and provide bandwidth equivalent to or greater than vendor tools.
- To provide a convenient Python interface to the debug cores once flashed to a FPGA, such that interacting with a vendor-provided Tcl console is not necessary.
- To be platform-agnostic, both to the host machine and the target FPGA. It should function identically regardless of host operating system and processor architecture, or FPGA vendor and family.

- To produce human-readable Verilog-2001 similar to what one would expect a human engineer to write, at a level understandable by the average 6.205 student. While users should never need to modify the source, this choice gives users a transparent view of the hardware placed on their FPGAs.

- To have no dependency on vendor-provided software, outside of that strictly necessary for building Hardware Description Language (HDL) source code to a bitstream. Dependencies on other external software should be used sparingly, and be made optional if possible.

Manta is named such because manta rays are an endangered and fragile species that should be loved, cherished, and protected, just like good hardware design. The name is also a nod to Mr. Ray, the jolly science teacher in *Finding Nemo*, whose enthusiasm towards teaching is fondly admired by the author of this text. His ability to transport fish to school on his back is also similar to Manta's ability to shuffle data on its bus.

Chapter 2

Related Work

2.1 Vendor Tools

Most FPGA designs are developed inside a vendor-provided environment, and as such the debugging tools included in the environment are those most often used. For designs utilizing Xilinx FPGAs with Vivado, this primarily consists of the Virtual IO Core and the Integrated Logic Analyzer. Other debugging tools are provided by Vivado, but these two cover nearly every use case encountered in 6.205.

2.1.1 Integrated Logic Analyzer

The ILA is by far the most commonly used tool in 6.205. The ILA functions much like a benchtop logic analyzer, with a set of probes connected to some logic under test. When a trigger condition is met, the ILA records the state of the probes on each clock cycle. This continues for some predefined number of clock cycles, and once complete, the captured data is sent back to the host machine and presented as a digital waveform. This tool exists as digital logic flashed on the FPGA along with a graphical interface in Vivado, and thus requires no external equipment.

For modest configurations, Integrated Logic Analyzer follows the same workflow as other Intellectual Property (IP) blocks in Vivado. The IP customizer is used to configure the ILA, setting the number and width of ports, the number of samples to

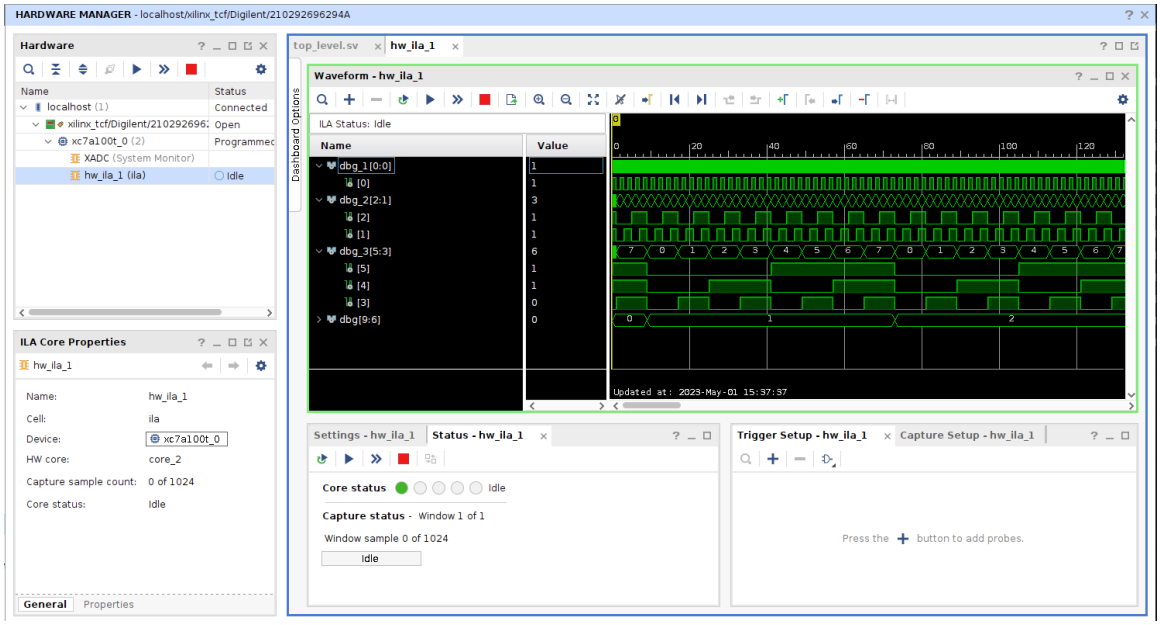

Figure 2-1: Waveform display of data captured by an ILA.

capture, and how many trigger conditions to evaluate. This is then used by Vivado to prepare a synthesized netlist - not source code. The source code used for synthesis is generated internally by Vivado from encrypted sources, and is never exposed to the user. A similar process occurs through the Tcl console for ILAs requiring a larger number of probes, or wider probe widths. This process targets post-synthesized designs, which complicates the build process.

Once flashed to the FPGA, this tool allows hardware designers to inspect the state of any net on the FPGA in pseudo-realtime. This ability has proven remarkably useful in 6.205, where student designs often work perfectly well in simulation, but fail when put into hardware. This is usually due to some difference between the test conditions provided in the simulation and the conditions provided by the real world. The ILA allows designers to debug their designs by playing spot-the-difference between the inputs provided to their devices in simulation, and the inputs provided to them in the real world.

This is most commonly seen when external devices need to be modeled - for example, it's nearly impossible to debug an I2S interface that works in simulation, but fails in hardware without an ILA. The simulation will only be able to predict the

behavior of the user's logic as long as the inputs are cycle-accurate. This is a tall order for students in a semester-long laboratory course that are learning digital logic fundamentals, languages, protocols, and toolchains all at the same time. As a result, the existence of the ILA is critical for fulfilling the present pedagogical goals of the course.

2.1.2 Virtual IO

Virtual IO behaves much like the name implies. It allows the user to define a set of signals on the FPGA whose state can be queried (in the case of an input port), or controlled (in the case of an output port). It does this in a workflow similar to the ILA, using Vivado's IP workflow to generate a synthesized netlist. This is then included in the user's design, and flashed on the FPGA. Once placed on the chip, the VIO core communicates with the host over the interface specified by the Joint Test Action Group (JTAG), setting the values of output probes reporting the values of input probes in response to communication from the host. This is done through Vivado's GUI or Tcl console.[17] [13]

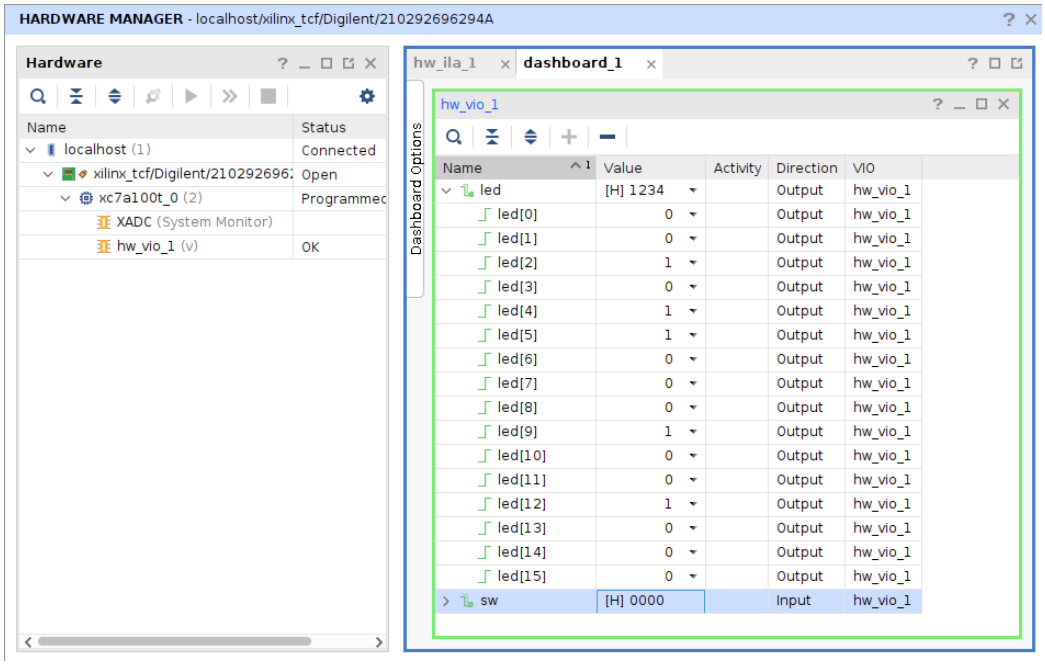

Figure 2-2: Xilinx Virtual IO (VIO) User Interface.[17]

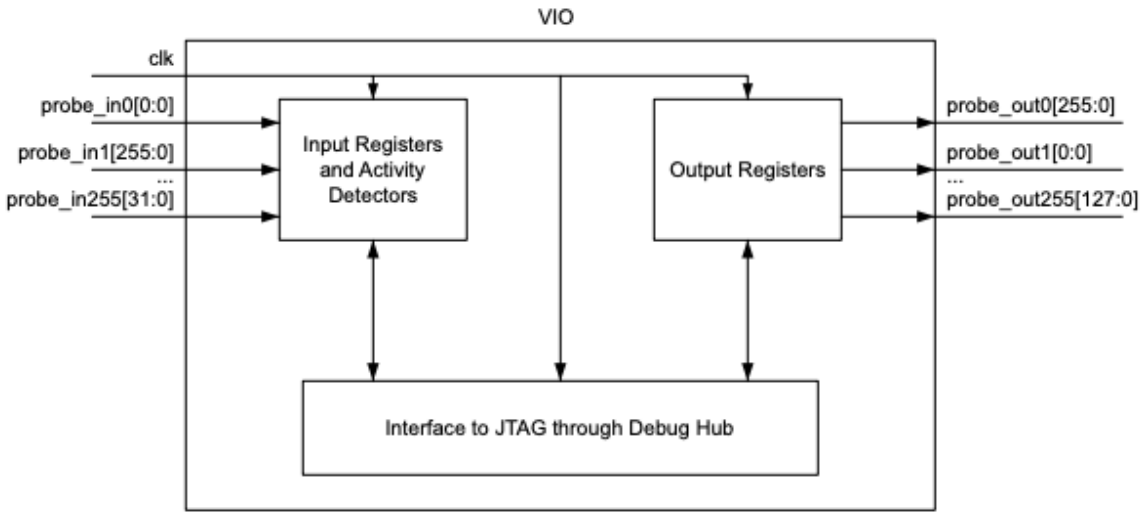

Figure 2-3: Xilinx Virtual IO Hardware Block Diagram.[13]

2.1.3 ChipScoPy

Xilinx also provides a less well-known interface to their debug cores called ChipScoPy. This exposes a Python Application Programming Interface (API) to debug cores on Xilinx Versal System-on-Chips (SoCs). These SoCs contain fixed-function silicon for debugging high-speed logic like DDR Memory, Gigabit network transceivers, and PCI Express busses, to which ChipScoPy can interface. It can also interface with debug cores implemented in programmable logic (such as the ILA and VIO cores), providing a Python API that exposes the same functionality present in Vivado's graphical interface. In the case of an ILA, the Python API can set trigger conditions, run the core, and export the acquired data as a waveform. In the case of a VIO core, the API can query the state of input probes, and set the state of output probes. This provides an interface that's more user-friendly and extensible than the Tcl scripting capabilities available in Vivado.[19] [15]

2.1.4 Signal Tap

Intel provides their own analogues to the ILA and VIO cores in the form of Signal Tap and In-System Sources and Probes (ISSP). Both are accessible only through Intel's FPGA development environment, Quartus Prime. These tools are largely equivalent

to their Xilinx counterparts, as they exist in the programmable logic of the FPGA, communicate with the host machine over JTAG, and do not expose the underlying HDL source. A programmable interface to these commands only exists via a Tcl scripting environment.

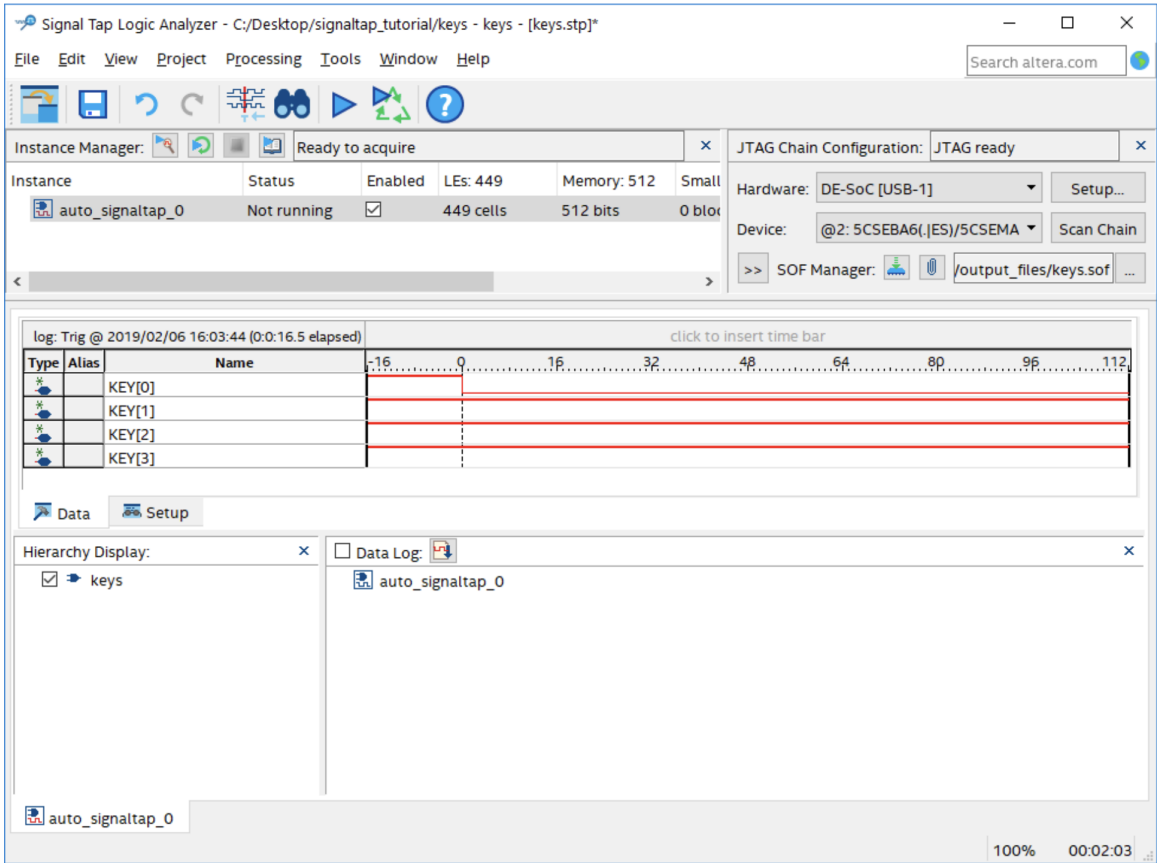

Figure 2-4: Waveform display of data captured by Signal Tap. [4]

Intel's Signal Tap is functionally equivalent to the Xilinx ILA, with a two notable extra features. First, a small scripting language is provided for specifying what Intel refers to as a *Trigger Flow*. This is a set of trigger conditions that are evaluated sequentially, such that when one condition is met the data is captured to a buffer, and depending on which condition was met, a new trigger condition is selected. This allows multiple trigger conditions to be cascaded through a user-defined state machine, with the state of the FPGA at each to be observed at each. As the scripting language is used only for specifying state machines, it implements few features other than conditional logic. [2] [3].

Second, Signal Tap contains a feature Intel refers to as *Simulation-Aware* nodes, which place capture data from a Signal Tap instance inside a simulation. This allows for running simulations on user-specified logic with data acquired from the real world. Naturally, this only supports the simulator included with Quartus Prime, Questa. At the time of writing, this feature is only out of beta in the most recent release of the full-featured version of Quartus Prime, Quartus Prime Pro Edition 22.4.[1]

2.1.5 In-System Sources and Probes

Intel's In-System Sources and Probes (ISSP) tool is the Intel equivalent to Xilinx's Virtual IO core. Signals being controlled by the host machine are referred to as *sources*, and signals driven by the FPGA are referred to as *probes*. These are observed and controlled through the Quartus GUI. The primary difference between ISSP and Xilinx's VIO is ISSP's inclusion of a waveform viewer. This shows a rolling history of any configured probes, albeit limited by the bandwidth of the FPGA's JTAG interface. No rolling waveform is provided for configured sources, only probes.

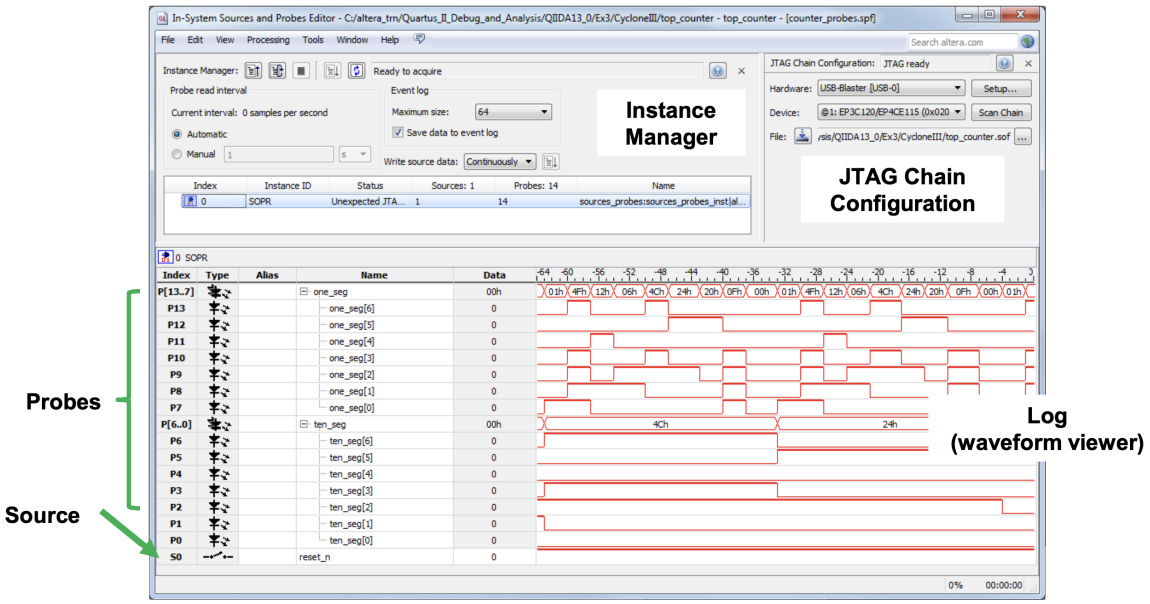

Figure 2-5: User interface to In-System Sources and Probes in Quartus.[4]

2.2 Commercial Offerings

Although the most commonly used closed-source debuggers are those provided by the vendor, one commercial tool deserves mention. While not directly marketed as a debugger, Opal Kelly's FrontPanel Software Development Kit (SDK) can be used as such. FrontPanel is designed to a provide a host computer with a real time interface to FGPA signals, and present them on a graphical "front panel". These front panels present themselves as a GUI window, and contain buttons, knobs, and indicators, mimicking those found on the front panels of benchtop electrical test equipment, hence the products name. These have a look and feel very similar to the front panels used on LabVIEW virtual instruments, but the underlying API is exposed to the user. Bindings for hosts running Windows, macOS, and Linux are provided, and target C, C++, C#, Python, Java, Ruby, and MATLAB. Communication between the host machine and target FPGA is done over either the Universal Serial Bus (USB) or Peripheral Component Interconnect Express (PCIe). On the FGPA side, the user is given a skeleton module with defined inputs and outputs, into which their HDL must fit. The user logic fits into this block, and is instantiated by the FrontPanel SDK which manages the entire module hierarchy.[12]

The FrontPanel SDK is best understood as a faster, cross-platform version of the Xilinx VIO core that uses USB and PCIe instead of JTAG, and includes an API like ChipScoPy by default. The only interface to user HDL is through the endpoints, and it does not include a logic analyzer.

2.3 Open Source Offerings

Several scripts exist on GitHub that provide similar functionality to the tools listed above.[11] [5] [25] These appear to function either by implementing logic analyzers in HDL from scratch, or by reverse-engineering output data from the vendor tools. None appear to have a significant user base, or much accompanying documentation.

However, one open-source system deserves mention. Dan Gisselquist has created

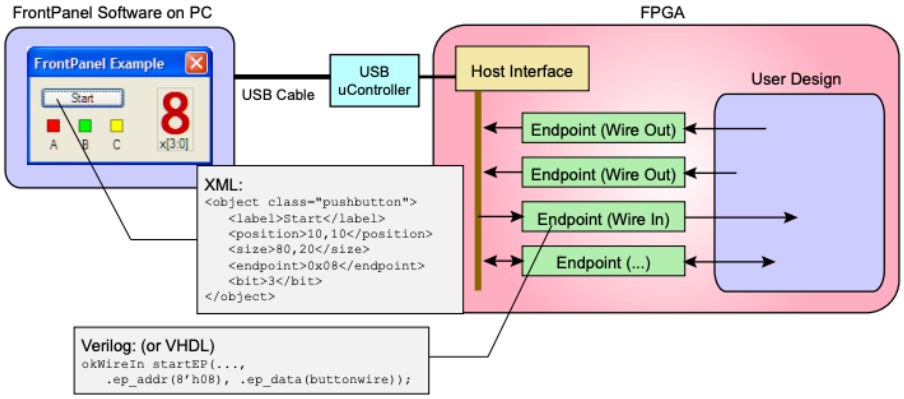

Figure 2-6: System architecture of Opal Kelly's FrontPanel SDK.[12]

an open-source bus-based debugger called `dbgbus`, which provides a C++ API on the host side, and a Wishbone bus interface on the FGPA side [9] [7] [10] [6]. The debugger itself exposes only a Wishbone bus controller to the user logic, which connects separately to other Wishbone-based debugging tools. These include user registers, block RAM, and a Wishbone scope. The registers and scope function similarly to the Xilinx VIO core and ILA respectively.

The debugging tools are provided as Verilog and C++ source only, so configuring the debugger consists of modifying them directly. In the case of the Wishbone scope, the FPGA side is configured by assigning the desired signals to a register in the scope definition, and the host machine side is set up by subclassing a C++ template class. Once both ends have been built and run, the resulting capture data is exported as a VCD file to be opened in an external waveform viewer such as GTKWave.

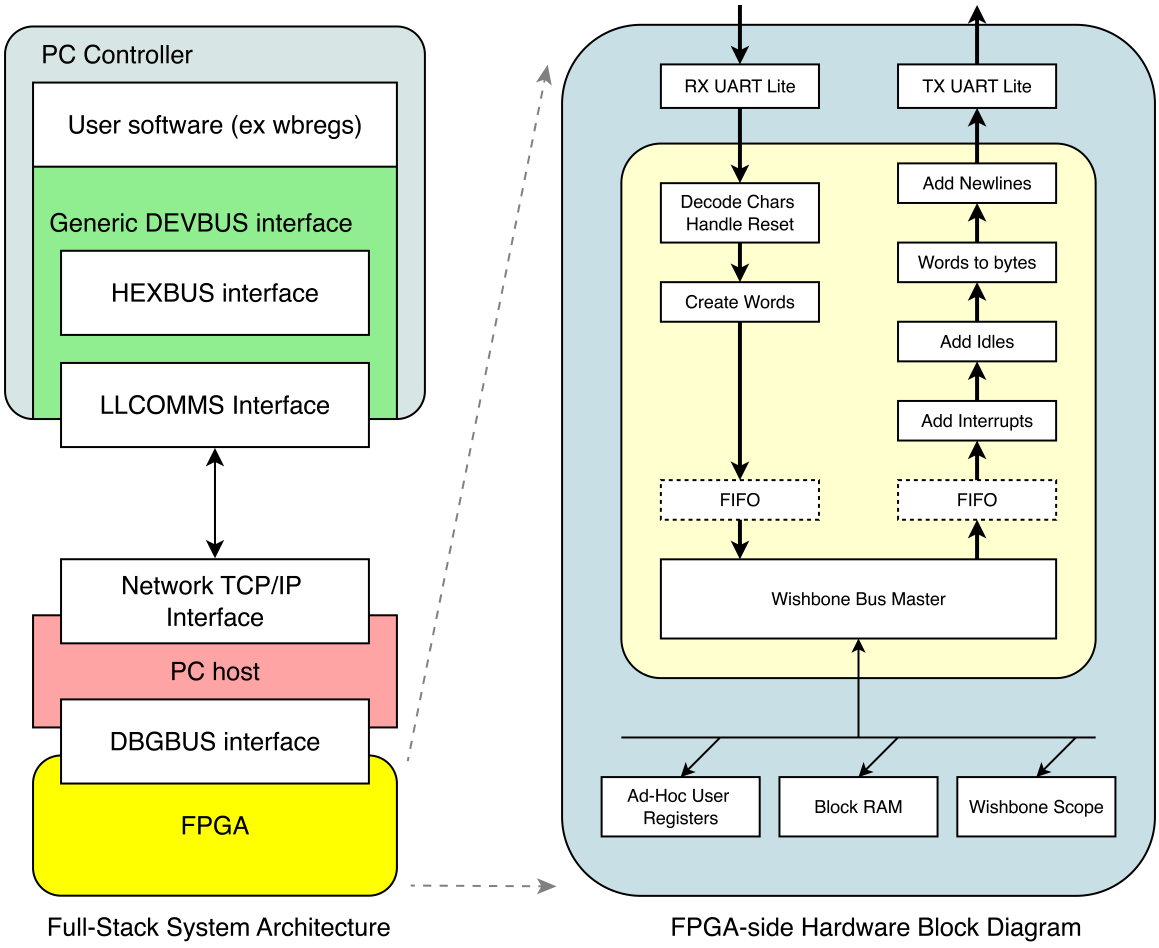

Figure 2-7: System architecture of `dbgbus`, at both the system level and FPGA-level.

Chapter 3

Design

3.1 Overview

Manta is packaged as a Python wheel, available through the Python Package Index (PyPI) and its frontend `pip`. The package performs two primary functions: procedurally generating synthesizable Verilog-2001 to be flashed to the FPGA, and communicating with the logic over UART or Ethernet once flashed. Both functions are performed in Python, which also provides an API for users to access in their own applications.

Using Manta requires a hardware setup like that shown in Figure 3-1. The host machine is connected to a FPGA development board with either a USB or Ethernet cable, over which communication takes place. The host accomplishes this communication through either a USB controller or Network Interface Controller (NIC), which the operating system provides a device driver for. These device drivers function differently between hosts running Windows or Portable Operating System Interface (POSIX)-compatible operating systems. To accommodate this, an interface API is used to abstract away platform-specific behavior and provide a single Python interface for device communication. Manta itself uses this to send UART bytes or Ethernet packets to the FPGA, but it also exposes a high-level Python API that may be used in custom applications written by the user. Taken together, the components of this stack provide a user-extensible API for communicating with the FPGA across

multiple operating systems and interfaces.

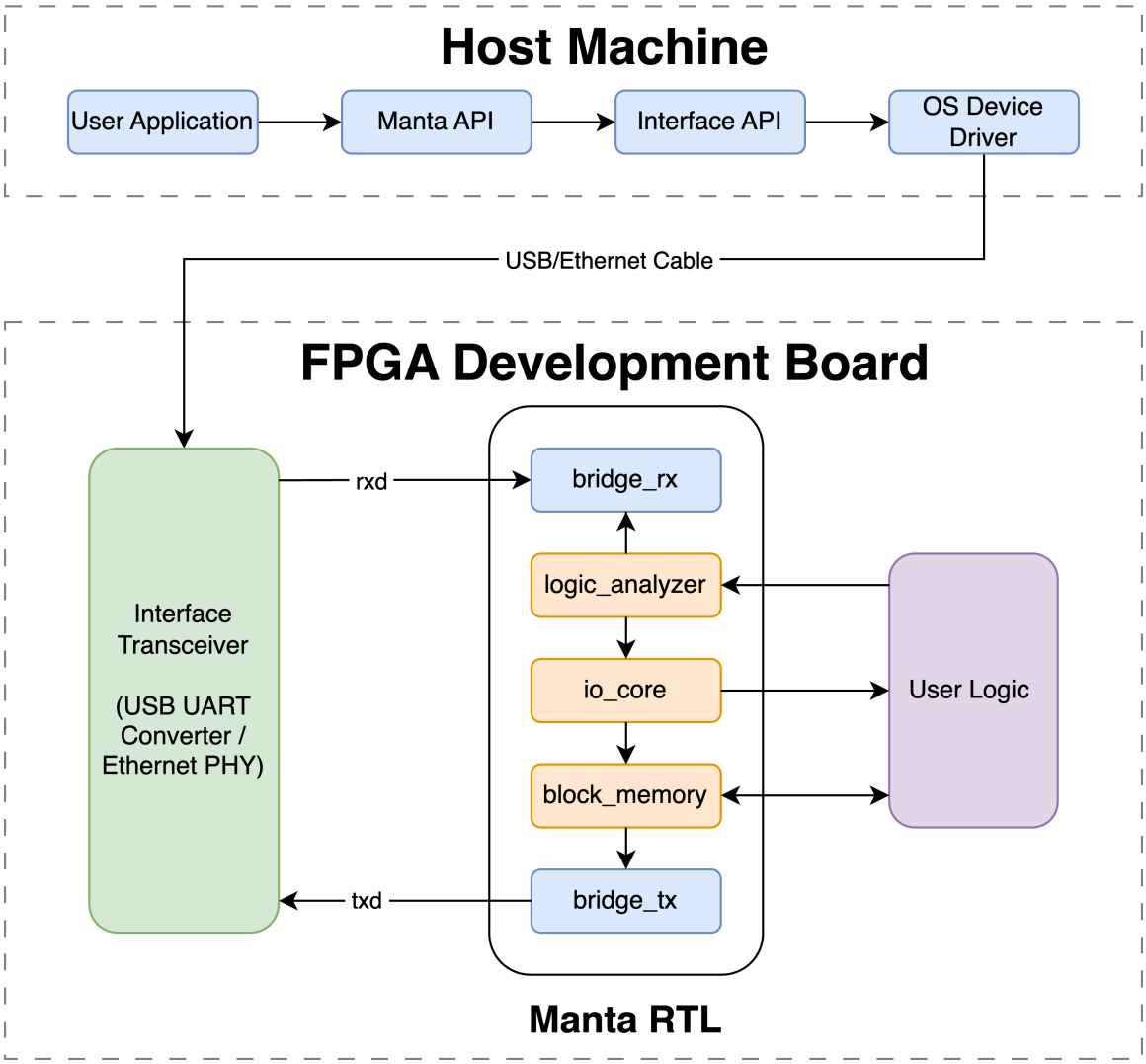

Figure 3-1: Block diagram of Manta.

Communications sent by the host are carried across the cable to the FPGA development board. There they are handled by an interface transceiver, an IC that converts the signals on the line to something interpretable by the FPGA. When UART is used, this transceiver is the FGPA development board's USB/UART converter, which provides the `tx` and `rx` signals to the FPGA. When Ethernet is used, this transceiver is the development board's Ethernet PHY, which provides a Media-Independent Interface (MII) to the FPGA. The transceiver's output is connected through an IO pin to the FPGA's fabric, where the hardware design specified by the user's HDL is im-

plemented. By including the Verilog generated by Manta in this HDL, the cores are implemented in the FGPA fabric. There, it exposes the user's logic to the host machine through the provided interface signals. This is accomplished by daisy-chaining the cores across an internal bus, and is described in Section 3.1.3.

3.1.1 Usage

Once the connection between the host machine and development board has been made, using Manta consists of the following:

- The user specifies a set of debug cores they wish to include in their design. This is represented as a configuration file, formatted as either JavaScript Object Notation (JSON) or Yet Another Markup Language (YAML). Both are human-readable and can be quickly written and modified, but the latter is preferred for its more minimal syntax. An example configuration, specified as YAML, is presented in Figure 3-2.

- The user invokes Manta to generate Verilog-2001 source code from the configuration provided. This is done via Manta's command line interface, which generates a single `.v` file containing a complete definition for a Verilog module named `manta`. This file is generated from a set of Verilog templates stored in the Python package, which are modified and combined to produce RTL matching the configuration specified by the user.

- The user instantiates `manta` in their design, and connects it to the logic they wish to debug. The user must also connect `manta` to the interface transceiver via the IO pins on the FPGA. This permits communication with the host machine.

- The user builds the design with a tool of their choice, and uploads it to the FPGA. Manta is designed to support as many toolchains as possible, and does not use any vendor, toolchain, or chip-specific features. Manta also exports only Verilog-2001, allowing it to maintain compatibility with most FPGA toolchains

released in the last twenty years, and avoid SystemVerilog's inconsistent interpretation across EDA tools. This allows for its use by maintainers of legacy systems, or budget-constrained educators using older FPGAs.

- The user then operates the debug core(s). Each core has a unique set of operations available, which can be performed either through the Python API, or the command line (which merely provides a convenient wrapper for the API). The exact syntax is provided in the API documentation, but users can capture data from a Logic Analyzer core, set and measure probes on an IO core, and read and write to a Block Memory core.

- Once the user has identified a bug or wishes to update their logic, they repeat this process. No changes are required to the configuration file or the generated Verilog, unless the user wishes to modify the debug cores placed on the device. This is continued until the user's design is functioning as intended.

3.1.2 Dependencies

Manta depends on a few pieces of external software. Some are only required when developing the package, while others are required while using it. The latter category consists of a handful of Python modules, including:

- pyYAML, which is used for parsing configuration files written in YAML.
- pySerial, used for communicating with the FPGA over UART.
- Scapy, used for communicating with FPGA over Ethernet.
- pyVCD, used for writing waveforms captured by the Logic Analyzer Core to standard Value Change Dump (VCD) files.

All of these dependencies are technically optional. A user comfortable writing configurating files as JSON does not need pyYAML, and a user using exclusively UART for communication does not need Scapy. However, Manta will try to install

```yaml
---
cores:
  my_io_core:
    type: io

    inputs:
      probe_0_in: 6
      probe_1_in: 12

    outputs:
      probe_2_out: 20
      probe_3_out: 1

uart:
  port: "auto"
  baudrate: 3000000
  clock_freq: 100000000
```

Figure 3-2: Example Manta configuration file. Here, an IO core with two input and two output probes is specified. These probes have widths of 6, 12, 20, and 1 bit respectively. The FPGA communicates with the host over an autodetected serial port running at 3Mbps, while the FPGA itself is clocked at 100MHz.

(or use an existing copy of) pyYAML, pySerial, and pyVCD during its own installation to cover most use cases.

Aside from the dependencies required to use Manta, a few dependencies are required to develop it. The package itself is built with Python's setuptools, and uses Black for linting it's Python source. Verilator is used for linting the Verilog templates, which are functionally tested in Icarus Verilog. This testing is performed automatically in a CI/CD pipeline created in GitHub Actions, which also builds example designs for the Nexys A7 and iCEstick. This is done with Vivado and the open-source iCE40 toolchain (Yosys/nextpnr/icepack) respectively.

Lastly, the most up to date documentation and exact usage instructions are hosted at `manta.mit.edu`, which is generated with Material for MkDocs.

3.1.3 System Architecture

The logic Mantas places on the FPGA consists of a series of modules connected in a chain along a common bus, as shown in Figure 3-3. Each module, called a *core*, provides a unique method for interacting with the user's logic, and as such the means by which the connection is handled varies per core. However, these connections are made by routing signals, called *probes*, between the user's logic and the cores that interface with it.

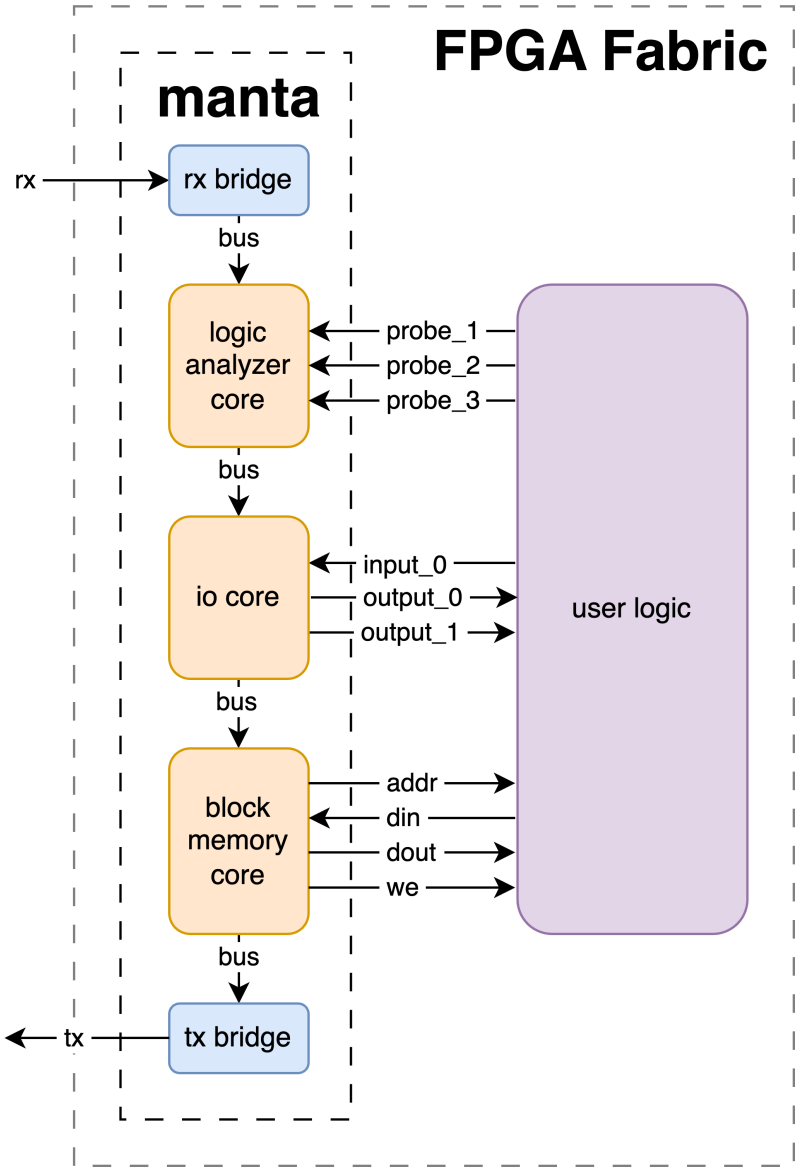

Figure 3-3: Functional block diagram of the logic Manta places on the FPGA.

These probes are then presented as addressable memory, and may be interacted with by reading and writing to them - not unlike registers on a microcontroller. Each core is allotted a section of address space at compile time, and operations addressed to a core's address space control the behavior of the core. These cores are then daisy-chained along an internal bus, which permits a chain arbitrarily many cores to be placed on the bus.

At the beginning of this chain is a module called a *receive bridge*, which converts incoming UART/Ethernet communication from the host into read and write requests, which are placed on the bus. These are called *bus transactions*, and once placed on the bus, they travel through each core before reaching the *transmit bridge* at the end of the chain. This module places the result of the bus transaction back on the UART/Ethernet interface, and sends it back to the host. This produces a request-response style of communication between the host machine and the FPGA.

Manta's architecture can be described as a set of probes connected to the user's logic, which are mapped to memory through a set of cores, all daisy-chained along an internal bus that provides responses to requests from the host. The design of these constituent elements - the bus, interfaces, and cores - is the subject of the remainder of this chapter.

3.2 Data Bus

3.2.1 Description

The data bus is designed for simplicity, and consists of five signals used to perform reads and writes on memory:

- `addr [15:0]`, indicating the memory address targeted by the current transaction.
- `rdata [15:0]`, containing the result of a read from memory.
- `wdata [15:0]`, containing data to be written to memory.
- `rw`, indicating a read or write transaction if the signal is low or high respectively.
- `valid`, which is driven high only when the operation specified by the other signals is to be executed.

Each core has a bus input and output port, which permits daisy-chaining by connecting the output of one core to the input of another. Upon receiving an incoming bus transaction, the core checks the address present on the wire against its own memory space. If the address lies within the core, the core will perform the requested operation, writing the data at the address to `rdata`, or writing the data on `wdata` to the address. However, if the address lies outside of the memory of the core, then the transaction is copied from the input port to the output port, and simply passes through the core. This is shown in Figures 3-4 and 3-5 for a read and write transaction, respectively.

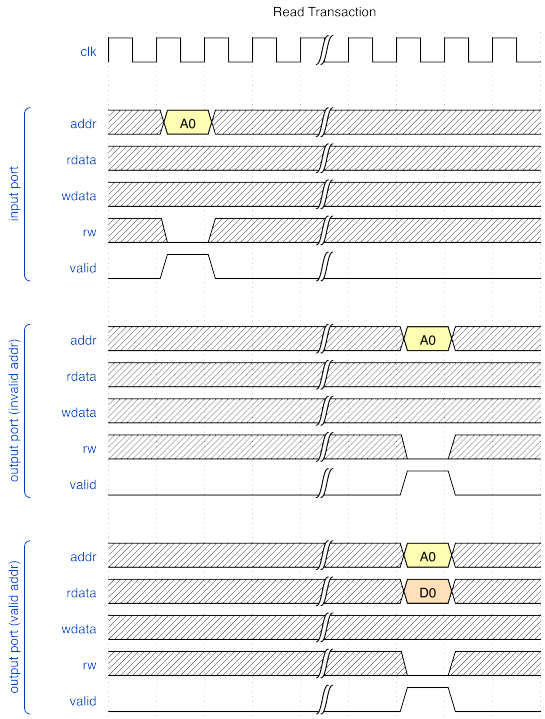

Figure 3-4: Waveform diagram of a read transaction on the bus.

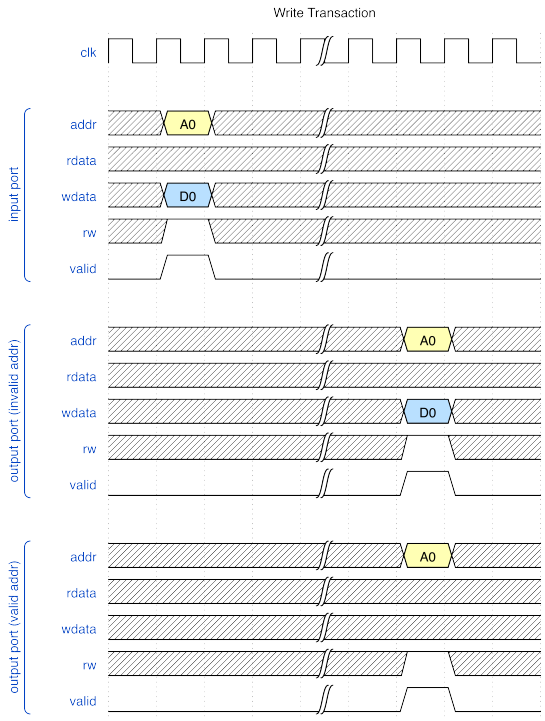

Figure 3-5: Waveform diagram of a write transaction on the bus.

3.2.2 Functional Simplicity

The design of this bus is intentionally very simplistic. This is done by applying two principles already present in the design:

- *All external logic is separated from Manta's internals.* Originally, using a "brand name" bus such as Wishbone, Avalon, or AXI was considered. This had the advantage that it would simplify interfacing to IP in user logic that used the same bus - at the expense of making it more difficult to interface to IP on any other bus. This conflicts with the design goals of simplicity and platform-agnosticism: AXI is popular amongst Xilinx IP, Avalon is used by Intel's IP, and Wishbone is preferred by some in the open-source hardware community. Choosing one meant alienating the others, which was unacceptable.

  Instead, it was chosen to separate Manta's internal bus from user logic, isolating it from any IP. The connection between user logic and the internal bus is not direct, and is moderated by the cores. This allows for greater safety and flexibility when interacting with logic that is, by nature, not behaving as expected.

  This puts Manta at an advantage against bus-based debuggers, like the open-source Wishbone-based offering provided by Gisselquist [6]. A general set of cores can connect to anything, regardless of if it has a Wishbone bus.

  By offering a set of cores that provide a general connection to user-provided logic, they can connect to anything, regardless of if it has a Wishbone bus. This design choice also prevents the user logic from interfering with the debugging tools. In the case of the Gisselquist debugger, the user-provided logic could very easily create a deadlock condition on the bus, preventing any data from being transferred back to the host machine, and disabling the debugger.

  Additionally, connecting to user logic through cores provides a very natural point to perform clock domain crossing (CDC). Cores execute reads and writes on the bus clock, but connect to user logic on its native clock domain. The means by which this is accomplished depends on the core. Cores utilizing block

memory will use a dual-port, dual-clock Block RAM to perform CDC, but those without it use a two flip-flop synchronizer.

- *Cores do not need to communicate amongst themselves.* This frees them from managing their own data transfers, and allows the bus to lack provisions for handling them. Bus transactions are initiated by the receive bridge, travel through the daisy-chained cores, and exit the transmit bridge. Cores have no knowledge that other cores exist, and only concern themselves with memory operations on their own address space.

Combined, these two assumptions allow for a bus design that is stateless, which simplifies design *considerably*. Any cores attached to the bus are only concerned with handling the present transaction, which allows Manta's bus to omit features found in brand name busses, such as:

- Notion of controllers or peripherals.
- Ready flags. The concept of backpressure does not exist in this bus design.
- Separate read and write channels. Even though the bus contains `wdata` and `rdata`, data flows in a single direction across the chain.
- Transaction privileges, burst modes, or strobe signals, secondary tag busses, or any other metadata.

3.2.3 Routing

In addition to its simplicity and extensibility, this bus design was chosen for its flexibility during routing. Routing is the last step in building HDL source code to an uploadable bitstream, and consists of routing the connections between primitives on the FPGA fabric. During this process, the EDA tools continuously evaluate timing constraints to ensure signals have fully propagated before the next clock edge, preventing metastability.

For some designs, satisfying these constraints is very difficult. Large designs that use significant resources spread their logic across the FPGA fabric, as do designs that incorporate multiple clock domains. This presents a large distance over which the signal must propagate within a single clock cycle. Designs using fast clocks reduce the amount of time available for signals to propagate, decreasing the distance they can travel on the FPGA.

One of Manta's core design goals is to place as little cognitive load on the user as possible. For users designing routing-constrained logic, this means using Manta should not force any reconsideration of on-chip routing, or restructuring of onboard clock domains. The internal bus provided by Manta accomplishes this by allowing registers to be added in series with the bus connection between each module. Adding registers eases the burden on the routing engine, as bus signals need only to reach the next register in a single clock cycle, not the next core. An arbitrary number of registers can be included, allowing the internal bus to reach any location on the chip. This is illustrated in Figure 3-6.

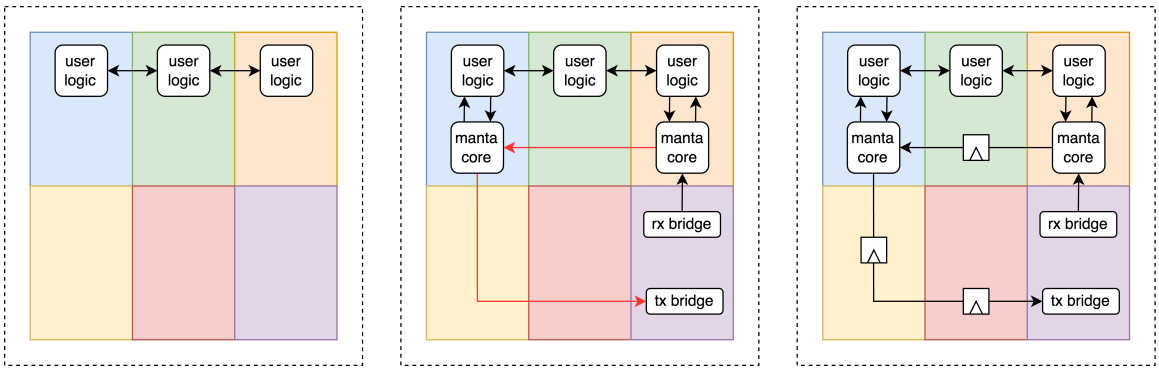

Figure 3-6: Bird's-eye view of the logic placed on an FPGA fabric, where each colored square represents a clock domain. On the left is a user design spanning multiple clock domains, to which Manta is added in the center figure. This creates long paths for the bus to travel, highlighted in red. Adding registers allows the design to satisfy timing constraints, and is shown on the right.

This does add a few clock cycles of latency between adjacent cores, however this does not affect the system's operation. Cores do not perceive this latency as they do not communicate amongst themselves, so this latency is only perceptible by the host machine, which receives responses to its request slightly later. For typical FPGA

designs in the hundreds of megahertz, this equates to a delay of a few tens of nanoseconds. This is far less than the rate at which the host's operating system flushes the read buffer of whatever interface it uses to communicate with the FPGA, and is therefore imperceptible to the host machine.

Lastly, it is worth noting that this daisy-chained, memory-mapped bus architecture takes strong inspiration from JTAG, the debugging interface of choice for the other kind of programmable logic - microcontrollers. Since the 1990s, JTAG has been commonly used to test microcontrollers during manufacturing, and debug them during development. This requires interacting with multiple sections of silicon scattered across the chip, which poses a similar routing problem. JTAG solves this by also using a daisy-chained bus, which can be routed over an arbitrary length on the chip. Despite its extremely widespread usage and routing flexibility, JTAG itself was not used as it not common for FPGAs to expose their JTAG controller to user logic. The behavior of these controllers is also very device-dependent, so foregoing JTAG allows for easier platform-agnosticism.

3.3 UART Interface

3.3.1 Description

The UART interface allows for arbitrary bytes to be sent to and from the FPGA. However, Manta's internal bus uses 16-bit address and data words, meaning multiple bytes must be used to communicate a bus transaction. This is done by adopting a standardized message format, which varies depending on if the message specifies a read or a write, or if the message is a request from the host or a response from the FPGA. These four formats are shown in Figure 3-7.

Each of these messages is a string of ASCII characters, consisting of a preamble, optional address and data fields, and an End of Line (EOL). The preamble consists of the letter `M`, and the address field encodes the address of the bus transaction as hexadecimal digits. This uses characters representing numbers 0-9 and letters A-F,

Figure 3-7: Format of read and write requests and responses

Sequence Number	UART Activity	Operation
1	Host → FPGA: `M1234(CR)(LF)`	-
2	FPGA → Host: `M5678(CR)(LF)`	Read `0x5678` from `0x1234`
3	Host → FPGA: `MF00DBEEF(CR)(LF)`	Write `0xBEEF` to `0xF00D`
4	Host → FPGA: `MF00D(CR)(LF)`	-
5	FPGA → Host: `MBEEF(CR)(LF)`	Read `0xBEEF` from `0xF00D`
6	Host → FPGA: `M12340000(CR)(LF)`	Write `0x0000` to `0x1234`

Table 3.1: Example UART traffic for memory reads and writes.

which requires four characters to represent the 16-bit address word. If the transaction is a write request, then it will contain a data field after the address field, which is represented in exactly the same manner. Both request types will conclude with an End of Line, which consists of the two ASCII characters indicating a Carriage Return (CR) and a Line Feed (LF).

These requests are sent by the host machine to the FPGA, which reads them from the `rx` line on the interface transceiver. This is handled by the receive bridge, which parses incoming messages, and generates bus transactions from them. Once

this transaction runs through every core in the chain, it arrives at the transmit bridge, which may send a response back to the host over the `tx` line.

If the request specified a read operation, then a response will be produced. These responses have the same structure as the read request itself, albeit with the data read from memory substituted in place of the address. This results in a message of the same length, just with the address swapped for data. If the request specified a write operation, then no response will be sent back to the host. This stems from the design of Manta's internal bus - the bus doesn't contain any metadata about a write operation, so there is no new information to provide the host. This differs from other data transfer models that provide metadata for write operations. For instance, a POSIX `write` syscall returns the number of bytes written, and a POST request made over the Hypertext Transfer Protocol (HTTP) returns a status code indicating if the write was successful. Manta provides no metadata of this sort. If a host wishes to know if a write was successful, it must read from the address after writing to it, and verify that the data returned is as expected.

3.3.2 Justification

This message format was designed primarily for human readability. Typically users do not need to manually inspect UART traffic, as it is handled automatically by the Python API. However, situations can be encountered where debugging the tool itself is necessary, at which point the UART traffic may need to observed. One of Manta's core design goals is to provide a simple user interface, and in these situations, the serial port itself becomes the user interface. As a result, the messages are designed to be as simple and readable as possible.

Consequently, it was chosen to represent numbers as hexadecimal digits so that they could be easily interpreted in a terminal emulator. If the address and data fields were encoded as raw bytes, they would contain a significant amount of missing data when viewed. This is because most terminal emulators parse the bytes as ASCII, which contains many "non-printable" characters that have no textual representation. These characters are not shown in a terminal, meaning the traffic would be impossible

to read - an undesirable behavior when debugging an interface.

Although not strictly necessary, the EOL is also included for readability. When viewed in a terminal emulator, it renders a newline, placing each bus transaction on its own line. This makes UART traffic far easier to follow. The EOL consists of both the Carriage Return and Line Feed characters, which is a choice made for compatibility. Windows needs both CR and LF to specify a newline, while POSIX-based operating systems only require the LF. Including both ensures traffic can be easily inspected on hosts following either convention.

Lastly, no hardware or software flow control is used, as none is needed. The FPGA is always ready to accept incoming bytes because responses are never longer than requests. Namely, a seven-byte read request will generate a seven-byte read response, and a 11-byte write request will generate no response. This means the FPGA will never need to stall an incoming request while it transmits a response. As a result, the FPGA is able to process a series of incoming requests with no gaps between them, and is always ready to accept an incoming request. This removes the need for flow control, as the host never needs to stop sending requests while it waits for the FPGA to process them. This reduces the complexity of both the UART interface as well as the data bus, which has no concept of backpressure and does not have any ready/valid signalling.

3.4 Ethernet Interface

3.4.1 Description

For situations where the onboard UART is not available, Manta provides a 100Mbps Ethernet link for communicating between the host machine and target FPGA. This link implements a L2 MAC on the FPGA, designed to be directly connected to a host machine on a dedicated network adapter. The MAC is controlled by a bridge interface, which performs the exact same function as it does on the UART interface. Incoming packets are parsed into bus transactions, placed on the bus, and any response data is

encapsulated into another packet sent to the host.

This is done by interacting with an Ethernet PHY, an onboard transceiver IC that converts between the FPGA's logic-level signaling and the voltages on the cable's twisted pairs. The communication between the Ethernet PHY and the FPGA is done over an interface that's dependent on the speed of the PHY. The 10/100 Mbps interface used on the Nexys A7-100T uses the RMII as defined in IEEE 802.3u. RMII is the second-oldest member in the Media Independent Interface family, with newer revisions of 802.3 supporting faster interfaces.

Manta's bus clock must be equivalent to the PHY's reference clock if Ethernet is to be used - in the case of the 100Mbps RMII PHY on the Nexys A7 used in 6.205, this is 50MHz. This doesn't pose a problem for user logic, which is connected through Manta's cores that perform CDC internally. It does mean that a reference clock for the PHY has to be synthesized outside of Manta itself, and the means by which this is done varies by FPGA vendor and toolchain.

This MAC allows for the usage of packets with the structure shown in Figure 3-8. The bus transaction being communicated is placed at the beginning of the packet's payload field, which IEEE 802.3 allows to vary in length from 46 to 1500 bytes. The 46-byte lower limit requires 41 bytes of zero padding to be added to the five bytes used to specify a bus transaction, and only one bus transactions is specified in each Ethernet frame. This abundance of unused space results in all packets being the same length, whether the packet contains a read request, write request, or read response. Packets containing write requests elicit no response from the FPGA, just as write requests delivered over UART produce no response. The justification for this behavior is shared between the Ethernet and UART interfaces, and is provided in Section 3.3.2.

These packets are addressed directly to the host's MAC address, which is obtained during code autogeneration. These packets also use a fixed Ethertype of `0x88B5`, which is specially reserved for "public use and for prototype and vendor-specific protocol development" in IEEE 802.1. This was done to create an Ethernet II frame instead of a legacy 802.3 frame, without having to implement a higher level protocol

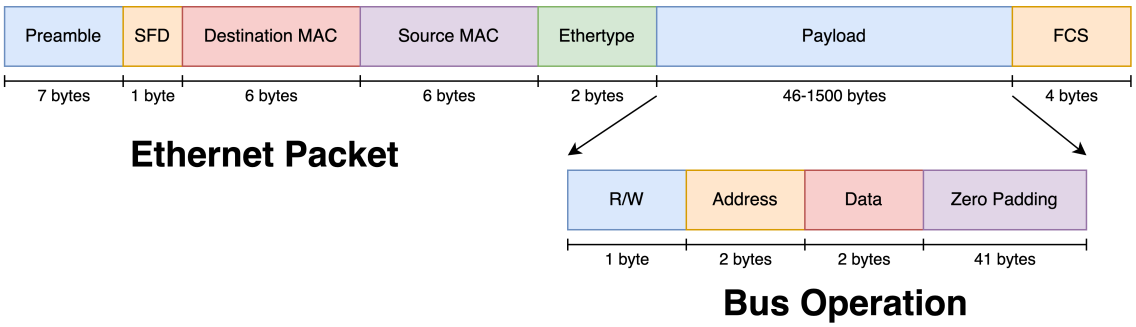

Figure 3-8: Structure of the Ethernet packets exchanged between the host and FPGA.

like TCP or UDP to safely use a fixed Ethertype. This allows the MAC to use modern Ethernet II frames safely, but save FPGA resources.

3.4.2 Justification

In addition to not being necessary for compatibility with modern networks, implementing higher-level protocols such as UDP or TCP was avoided. This would permit debugging any FPGA on the same network as the host, but this would not be particularly useful as a local machine would still be required for uploading bitstreams, which happens frequently during debugging. Instead, it is recommended that remote debugging be performed by connecting a host machine to a remote FPGA over a local link, and then logging in into the remote host. This paradigm is popular amongst vendor tools, which can expose debug servers over the network for remote debugging.

Lastly, the design and source code of the MAC's receiving side is taken from the Ethernet lab developed for 6.205 by Jay Lang. This was done for convenience and maintainability, as the course staff using Manta will already be familiar with this MAC implementation. The MAC's transmit side was completed by the author of this text, and reused some portions of the receiving side. Portions written by Lang are distributed under a 3-Clause BSD license, which is included along with his copyright in the publicly-distributed source code. This BSD license permits redistribution under alternative licenses, so while it may contain components with other licenses, Manta on the whole is released under Version 3 of the GNU General Public License (GPLv3).

3.5 Block Memory Core

3.5.1 Description

Block memory, also referred to as block RAM (BRAM), is a staple of FPGA designs. It consists of dedicated blocks of memory spaced throughout the FPGA die, and is very commonly used in hardware designs due to its configurability, simplicity, and bandwidth. Although each block memory primitive is made of fixed-function silicon, EDA tools allow them to be mapped to logical memories of arbitrary width and depth, combining and masking off primitives when necessary. These are exposed to the user's logic over *ports*, which contain four signals for reading and writing to the BRAM. These signals specify the address, input data, output data, and the desired operation (read/write) to the core. Most BRAM primitives include two ports, each of which may live on a separate clock domain, making them useful for clock domain crossing in addition to data storage. Each port can handle a memory operation on every clock edge, which is practically the maximum memory bandwidth possible in any digital system.

Central to Manta's design objectives is the ability to debug user logic in an intuitive and familiar manner. Practically, this means being able to interact with bits on the FPGA in whatever method they're presented. Block memory is one such method, and their pervasive use is acknowledged by the inclusion of a Block Memory Core in Manta. This core takes a standard dual-port, dual-clock BRAM and connects one port to Manta's internal bus, and gives the other port to the user. This means that both the host machine and the user's logic have access to the BRAM, allowing large amounts of data to be shared between both devices.

This is accomplished by architecting the Block Memory Core as shown in Figure 3-9. Internally, the Block Memory Core consists of multiple BRAMs connected in parallel. This is done to maintain the ability to create block memory of arbitrary width and depth. Manta's internal bus uses 16-bit data words, so if a user wishes to create a BRAM of width N where N is larger than 16 bits, then multiple addresses in Manta's memory are required to contain the data at a single BRAM address.

These multiple addresses are created by creating many smaller block memories, each of which stores a 16-bit slice of the N-bit wide data. As a result, $ceil(\frac{N}{16})$ smaller BRAMs are needed to present a BRAM of width N to the user. One set of ports on these smaller BRAMs are concatenated together, which presents a N bit wide BRAM to the user. The other set of ports are individually connected to Manta's internal bus.

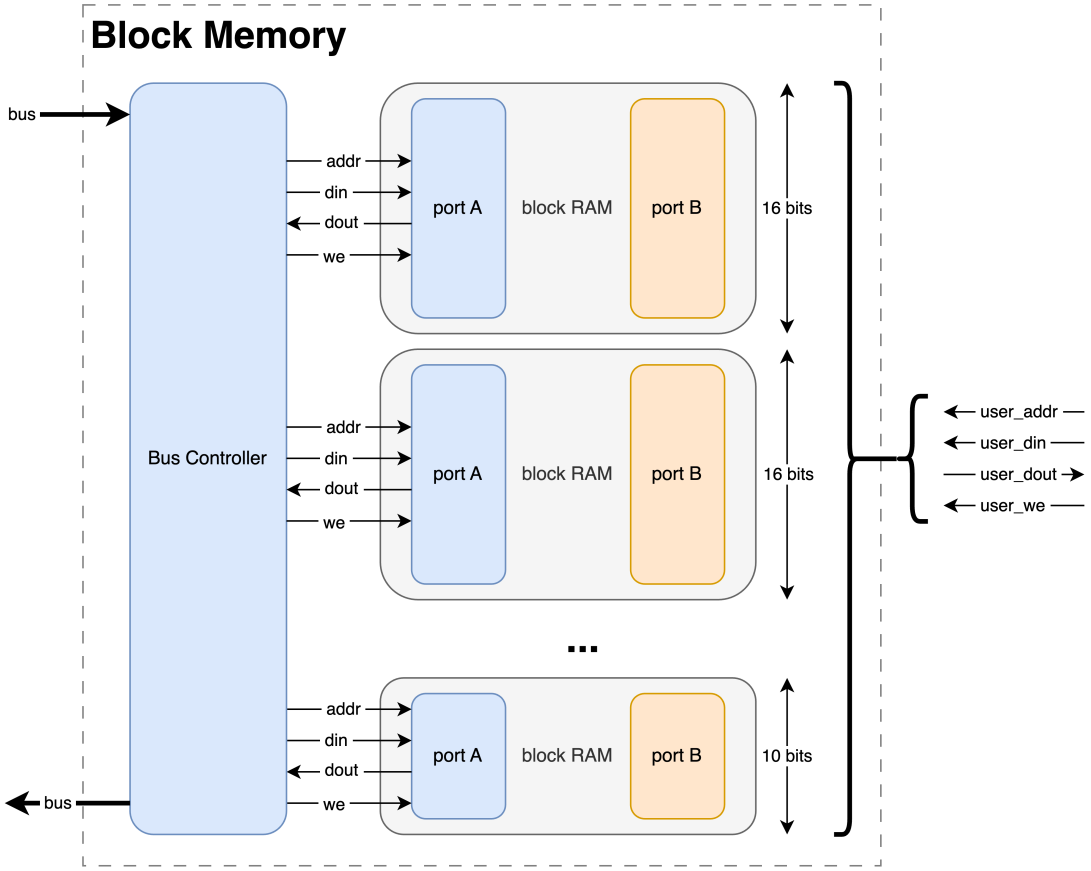

Figure 3-9: Block diagram of the Block Memory Core. Blocks in blue are clocked on the bus clock, and blocks in orange are clocked on the user clock.

3.5.2 Justification

Although the arrangement of the constituent BRAMs is relatively simple, it is rather inefficient in terms of address space. For instance, a Block Memory Core configured to be 19 bits wide and 128 addresses deep would be made of two 16 bit wide BRAMs,

containing 128 addresses each. This would occupy 256 addresses, each of which contain a 16-bit data word. This means that $256 \times 16 = 4096$ bits can be accessed, but the user will only store $256 \times 19 = 2432$ bits in the memory. If the memory was perfectly packed, then only $2432 \div 16 = 152$ addresses would be required. This means that only 59% of the addressable memory contains useful bits.

Address space comes at a premium on a 16-bit address bus. Numerous methods were considered for fully-packing data into each memory address, but none were simple enough to warrant implementation. It is believed that the true solution to this problem is changing the address width on Manta's internal bus. This is discussed as future work in Section 5.1.4.

However, it should be noted that this inefficiency only applies to the address space, not the BRAMs themselves. During synthesis, EDA tools will find unconnected registers and prune the logic driving them, including those within BRAMs. In the example described above, two 16-bit wide BRAMs implement a block memory core that is presented to the user as 19-bits wide. As designed, Manta will place the first 16 bits of the 19-bit data word into the first BRAM, and the remaining 3 bits in the second BRAM. This leaves the remaining 13 bits unused. Normally, this would cause 16 bit's worth of resources to be utilized on the FPGA, but EDA tools will optimize these out during synthesis. As a result, no unnecessary BRAM is claimed on the FGPA.

It is also worth mentioning that this architecture raises an issue of synchronicity. For BRAMs with widths larger than 16 bits, updating the data at one (user-side, not bus-side) address requires multiple bus transactions. During this time, the data at a given user-side address will be a mixture of its initial value, and the incoming value from the bus. Depending on the application, this can be problematic. If the user's logic assumes that the entire contents of the BRAM are valid at all times, then garbled data can propagate into downstream logic.

In situations like this, it is typical to include two BRAMs, such that one may be written to while the other is read from. Which memory bank to use for each purpose is typically communicated with a doorbell, such that the two may be exchanged once

the entirety of the new data arrives. This doorbell can be implemented as an input or output probe on an IO core, which is described in Section 3.6.

Lastly, the Block Memory Core uses inferred BRAM templates. These are snippets of human-readable Verilog that define logic that *behaves like* a BRAM, such that it is *implemented as* a BRAM by the EDA tools during build. This allows for greater portability between chip families and vendors, each of which design the block memory primitives on their chips differently. By using an inferred BRAM template, Manta does not need to account for these differences, and thus the resulting complexity of offloaded to the EDA tools.[18]

3.6 IO Core

3.6.1 Description

Registers are a fundamental building block of digital hardware. Registers store values as they move throughout the FPGA, and are operated on by the logic placed onboard the chip. Interfacing with this logic in an intuitive manner is Manta's primary design objective, and as a result it includes an Input/Output (IO) core to directly measure and control arbitrary signals on the FPGA. This is done by routing them to registers, which are then exposed to the host over Manta's internal bus.

This is done with the architecture shown in Figure 3-10. A series of connections are made to the user's logic. These are called *probes*, and each may be either an input or an output. If the probe is an input, then its value is taken from the user's logic, and stored in a register that may be read by the host machine. If the probe is an output, then its value is provided to the user's logic from a register written to by the host. The widths of these probes is arbitrary, and is set by the user at compile-time.

However, the connection between these probes and the user's logic is not direct. The state of each probe is buffered, and the buffers are updated when a *strobe* register within the IO core is set by the host machine. During this update, new values for output probes are provided to user logic, and new values for input probes are read

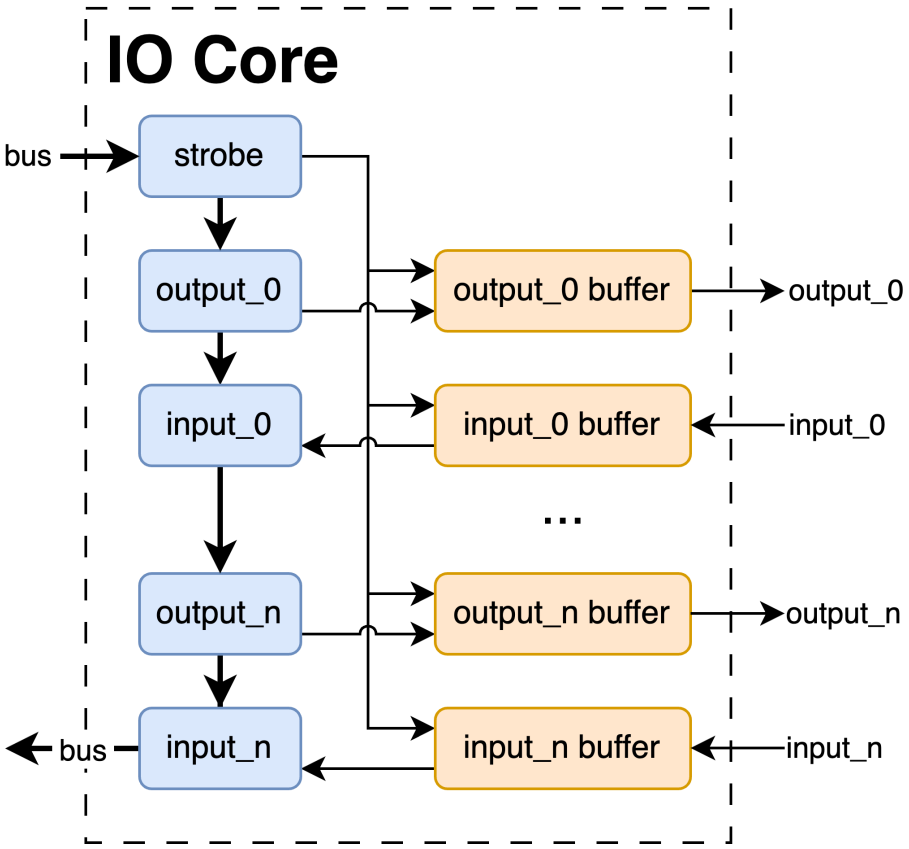

Figure 3-10: Block diagram of the IO core. Blocks in blue are clocked on the bus clock, and blocks in orange are clocked on the user clock.

from user logic.

This is done to mitigate the possibility of an inconsistent system state. Although users may configure registers of arbitrary width, Manta's internal bus uses 16-bit data words, meaning operations on probes larger than 16 bits require multiple bus transactions. These transactions occur over some number of clock cycles, with an arbitrary amount of time between each.

This can easily cause data corruption if the signals were unbuffered. For instance, a read operation on an input probe would read 16 bits at a time, but the probe's value may change in the time that passes between transactions. This would cause the host to read a value for which each 16 bit chunk corresponds to a different moment in time. Taken together, these chunks may represent a value that the input probe never had. Similar corruption would occur when writing to an unbuffered output

probe. The value of the output probe would take multiple intermediate values as each 16-bit section is written by the host. During this time the value of the output probe is not equal to either the incoming value from the host, or the value the host had previously written to it. The user logic connected to the output probe has no idea of this, and will dutifully use whatever value it is provided. This can very easily induce undesired behavior in the user's logic, as it is being provided inputs that the user did not specify.

Buffering the probes mitigates these issues, but slightly modifies the way the host machine uses the core. When the host wishes to read from an input probe, it will set and then clear the strobe register, which pulls the current value of the probe into the buffer. The host then reads from buffer, which is guaranteed to not change as it is being read from. Writing to an output probe is done in much the same way. The host writes a new value to the buffer, which is flushed out to the user's logic when the strobe register is set and cleared. This updates every bit in the output probe all at once, guaranteeing the user logic does not observe any intermediate values.

These buffers also provide a convenient location to perform clock domain crossing. Each buffer is essentially a two flip-flop synchronizer, which allows the IO core to interact with user logic on a different clock than Manta's internal bus.

3.6.2 Justification

The functional simplicity of the core left little choice in its architecture. However, a minor variation of this architecture omitting the strobe register was considered. In this design, when a host wanted to read or write to probes wider than 16 bits, it would still access the constituent memory addresses over multiple transactions, but ordered the transactions by increasing memory address. This meant that the lowest memory location occupied by a probe would be accessed first, and the highest memory location would be accessed last. This would allow the IO core to know when a probe was going to be read from or written to, and automatically flush the buffers. In the case of an input port, this would occur when the lowest memory location was accessed, and in the case of an output port, this would occur when the highest memory location was

accessed.

This approach offered the advantage of speed at the expense of complexity. By inferring when the buffers should be flushed, setting and clearing the strobe register would not be necessary. However, this increase in speed would be moot, as the IO core is not intended to interface with timing-critical logic. The speed at which the IO core updates signals on the FPGA is entirely dependent on the speed of the interface between the host machine and FPGA, whose timing is not guaranteed. As a result, increasing the speed of something asynchronous with user logic is not particularly useful, and does not justify the additional complexity.

3.7 Logic Analyzer Core

3.7.1 Description

Central to Manta's design is the ability to debug logic in a manner intuitive and familiar to 6.205 students. As such, Manta includes a logic analyzer tool that allows them to inspect their logic through a waveform display, similar to how it might be inspected through simulation. A typical workflow for using the core consists of the following:

- The user describes the signals they would like to probe in the configuration file. The user provides a list of probe names and widths, which are needed to generate suitable Verilog.
- The user describes the *trigger conditions* that must be met inside the FPGA fabric for a capture to begin. Triggers are defined as simple logical operations on probes, for instance checking if a probe named `foo` is equal to the number 3, or if a probe named `bar` has just transitioned from high to low. The user also specifies the number of samples to be captured, referred to as the *sample depth* of the core.
- Once fully configured, a Manta module is generated and flashed to the target

FPGA with the process described in 3.1.1.

- Once flashed, the user initiates the ILA from the host machine. This causes the Logic Analyzer Core to start sampling its inputs, waiting for the trigger condition to be met.
- Once met, the core begins saving the values of the probes to an internal block RAM called the *sample memory*. This occurs every clock cycle until a number of samples equal to the sample depth has been captured, and the sample memory is full.
- Once complete, the host machine reads out the sample memory and stores it internally. This is then exported as a VCD file for use in a waveform viewer like GTKWave.

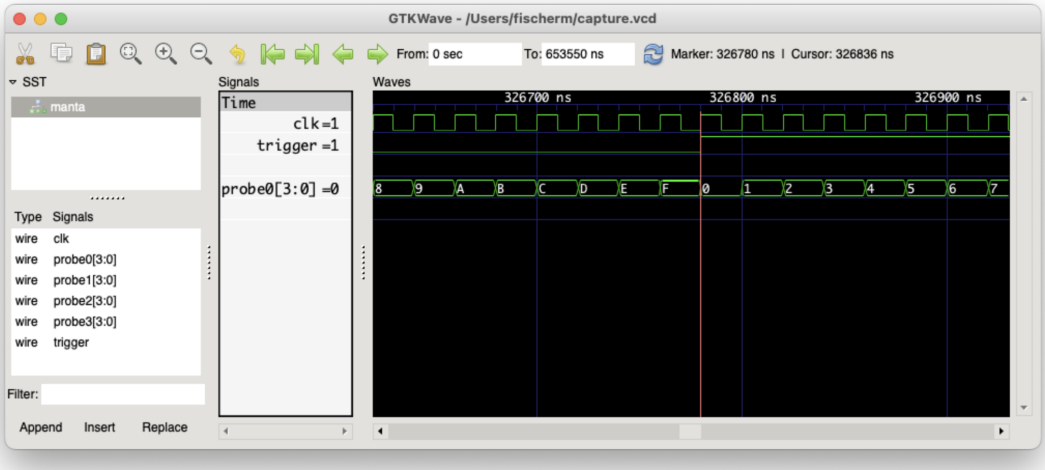

Figure 3-11: A logic analyzer capture displayed in GTKWave.

This workflow is very similar to the behavior of the Xilinx ILA or a benchtop logic analyzer. This is intentional. FPGA engineers are familiar with on-chip logic analyzers, and electrical engineers are familiar with external logic analyzers. Very little is intended to be different, although a few extra features deserve mention:

3.7.2 Features

Trigger Modes

The behavior described in 3.7.1 is referred to as single-shot trigger mode. This means that once the trigger condition is met, data is captured on every clock cycle in a continuous single shot. This is useful and the preferred behavior for most cases, but Manta also supports *Incremental* and *Immediate* trigger modes.

In Incremental mode, samples are only recorded to sample memory *when* the trigger condition is met, not *once* it is met. This allows slower-moving behavior to be captured. For instance, digital audio signals on a FPGA commonly use a 44.1kHz sampling frequency, but are routed through FPGA fabric clocked at hundreds of megahertz. As a result, many thousands of clock cycles may go by before a new audio sample is processed by the FPGA - filling the sample memory of a traditional logic analyzer with redundant data in the meantime. Placing Manta's Logic Analyzer into incremental mode solves this, as audio samples will only be saved to the sample memory when they change, assuming the trigger is configured correctly. In this case, the amount of memory required on the FPGA to capture a fixed number of audio samples is reduced by a thousandfold.

In Immediate mode, the trigger condition is ignored. The core begins filling the sample memory as soon as it is enabled, stopping only once the sample memory is filled. This allows the user to inspect the current state of their probes without a trigger condition. This is especially useful for investigating cases where a trigger condition is never being met, such as latchup or deadlock conditions. This mode is also useful for obtaining a random snapshot of the FGPA's state. The core is enabled by an interface (UART, Ethernet) that is slow relative to the clock speed of the FPGA fabric, meaning that the capture occurs at an effectively random time. Successive captures of this nature can be used to determine the "average" state of onboard logic - what information is "usually" on a bus, or what state a module is "typically" in.

Configurable Trigger Location

In the scenario described in 3.7.1, the sample memory is written to as soon as the trigger condition is met - and not before. This only records the probe values after the trigger, but knowing the state of the FPGA immediately before is also rather useful. To do this, the core can be configured to buffer the last few clock cycles before the trigger condition. During this time the sample memory is used as a FIFO, and once the trigger condition occurs, samples are acquired until the sample memory is filled. The number of cycles to record ahead of the trigger is called the *trigger position*. By default, most logic analyzers place the trigger condition in the middle of the acquisition such that there is equal amounts of data from before and after the trigger condition. To feel as intuitive and familiar as possible, Manta defaults to the same. However, this can be changed by writing to a register in the logic analyzer core.

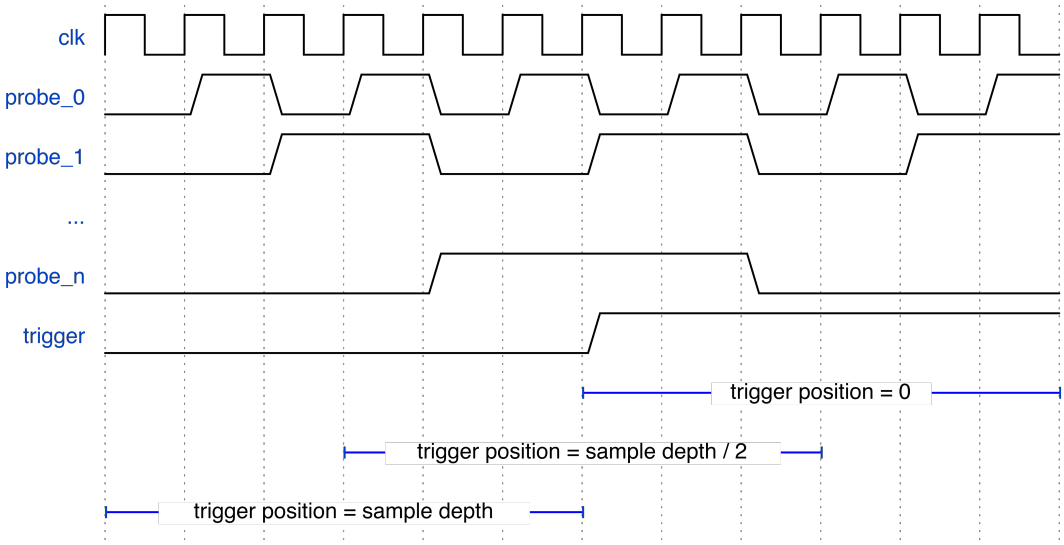

Figure 3-12: Regions captured by the Logic Analyzer Core as trigger position is varied.

Simulator Playback

Manta also allows data captured from the Logic Analyzer core to be "played back" in simulation. Any obtained capture data can be exported as a `.mem` file, which can

be used in most simulators via the `readmemh` and `readmemb` Verilog functions. Manta autogenerates a convenient Verilog wrapper for this, allowing users to simulate logic with signals directly measured from the real world. This is useful for verifying that a testbench is providing the proper inputs to logic under test. This is useful for a few scenarios:

- *Input Verification.* This targets the common student experience in 6.205 of designs working in simulation, but failing in hardware. In the absence of any build errors, this usually means that the inputs being applied to the logic in simulation don't accurately represent those being applied to the logic in the real world. [1] Playing signals back in simulation allows for easy comparison between simulated and measured input, and the state of the logic downstream.

- *Sparse Sampling.* When users are debugging, their fundamental concern is the state of their logic. Normally this is obtained by sampling every net of interest with a logic analyzer probe, but for designs with a large amount of internal state sampling many signals requires significant block memory and lots of time to set up. If the design has fewer inputs than state variables, it requires fewer resources to sample the states and simulate the logic than to directly sample the state. For instance, debugging a misbehaving branch predictor in a CPU can be done by recording its address and data busses, playing them back in simulation, and inspecting the branch predictor there. This frees the user from having to sample the entire pattern history table, which would consume significant block memory.

Reprogrammable Triggers

Manta's triggers are reprogrammable, such that rebuilding source code is not necessary to change the trigger condition. Each of the logic analyzer's input probes has a

[1] Sometimes the toolchain will step in and modify the logic specified by the user. For example, if a net is driven by two nets at the same time, Vivado will connect the net to ground, and raise a critical warning. In this case, a valid bitstream is still generated, but it doesn't configure the FGPA in a way that will match simulation.

trigger assigned to it, which continuously evaluates some combinational function on the input. This logic can be programmed to check for rising edges, falling edges or any change at all. It can also be programmed to check the result of a logical operation (such as $>$, $\leq$, $=$, $\neq$, etc.) against an *argument*. The operation and argument for each probe's trigger are set with a pair of registers in Manta's memory.

The output of each of the individual triggers is then combined to trigger the logic analyzer core as a whole. These are combined with a N-input logic gate (either AND or OR) specified by the user through another register in memory. As a result the entire trigger configuration is specified by the state of Manta's memory, and changes to the configuration require resetting registers, not resynthesizing bitstreams.

However, this greatly restricts the trigger conditions users can specify. To mitigate this, Manta provides an option for an external trigger that allows for more complex triggers. When enabled, Manta adds an input port to the `manta` Verilog module, and triggers off its value, rather than the internal comparators. This allows users to provide their own Verilog to produce the desired trigger condition.

3.7.3 Architecture

The Logic Analyzer Core's implementation on the FPGA consists of three primary components:

- The *Finite State Machine (FSM)*, which controls the operation of the core. The FSM's operation is driven by its associated registers, which are placed in a separate module. This permits simple CDC between the bus and user clock domains.

- The *Trigger Block*, which generates the core's trigger condition. The trigger block contains a trigger for each input probe, and the registers necessary to configure them. It also contains the N-logic gate (either AND or OR) that generates the core's trigger from the individual probe triggers. CDC is performed in exactly the same manner as the FSM. If an external trigger is specified, the

trigger block is omitted from the Logic Analyzer Core, and the external trigger is routed to the FSM's `trig` input.

- The *Sample Memory*, which stores the states of the probes during a capture. This is implemented as a dual-port, dual-clock block memory, with the bus on one port and the probes on the other. The probe-connected port only writes to the memory, with the address and enable pins managed by the FSM. CDC is performed in the block RAM primitive itself.

3.7.4 Justification

Special attention was paid to clock domain crossing while designing this core. Manta hosts multiple cores by daisy-chaining them across a common bus, which can be on a different clock than the user logic. This requires proper clock-domain crossing, which can be confusing to implement if not explicitly specified in the architecture. To ease this, the Logic Analyzer Core groups all configuration registers into modules that run on the bus clock. Separating them into a separate module allows for the easy inclusion of a two flip-flop synchronizer between the registers and the logic they control. This separation also allows for code reuse, as the configuration registers for the FSM and trigger block share the same source as the IO core described in (3.6).

Care was also taken in the design of the reprogrammable logic for specifying trigger conditions. While restrictive, the comparator-based design offers reasonable flexibility while consuming few resources on the FPGA. Logic that supports on-the-fly reconfiguration for more complicated triggers could be designed, but would consume significant resources on the FPGA.

For instance, at one point it was considered to specify trigger conditions with a small scripting language, which would compile to bytecode for a miniature CPU inside the trigger block. This would allow for extreme generality, but would require significant time to learn the language, and FPGA resources to run the bytecode. Very rarely does 6.205 encounter situations requiring trigger conditions that would require a triggering scheme this elaborate, and those that do usually indicate poor systems

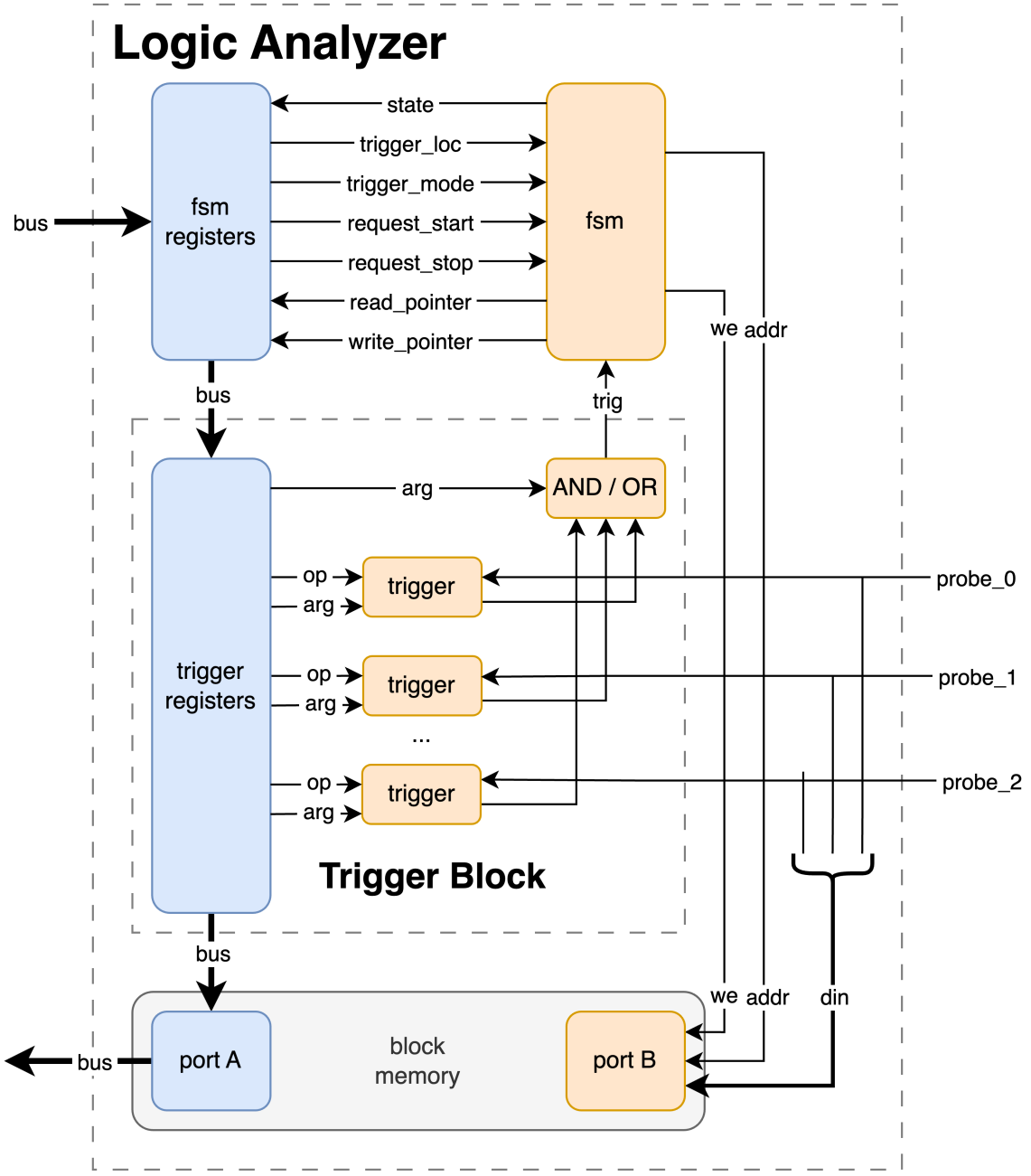

Figure 3-13: Block diagram of the Logic Analyzer Core. Blocks in blue are clocked on the bus clock, and blocks in orange are clocked on the user clock.

design on behalf of the user. As a result, use cases requiring complexity beyond that provided by the comparator-based approach are deemed far too rare to design for, and are not worth the added complexity, resource utilization, and time.

Simplicity aside, comparator-based approaches are familiar to most users. The

Xilinx ILA specifies its trigger condition with comparators, as do most benchtop logic analyzers. Keeping with convention prevents users from being distracted by extra features, and allows them to focus on debugging their logic.

Chapter 4

Evaluation

4.1 Bandwidth

When using Manta, the user will often need to access a large portions of Manta's internal memory. For example, running a Logic Analyzer core requires the host to read the entire sample memory contained in the core. This action is blocking, and the user must wait for data to be read from the entire core before they may continue debugging. As a result, the speed at which data can be read and written heavily influences the user's productivity and experience. Manta's internal bus can read or write 16 bits of data to memory every clock cycle, which, on the native 100MHz clock of the Nexys A7 used in 6.205, is 1.6 Gbps. Although the bus supports bandwidths this high, memory operations do not take place at this speed as they are bottlenecked by the interface used to connect the host machine to the FPGA. These interfaces allow the host machine to issue transactions (memory reads and writes) on Manta's internal bus, over UART or Ethernet.

The investigation of these limitations is the subject of this chapter.

4.1.1 UART Interface

Most FPGA development boards expose a UART interface over a USB port, which is typically also used for programming the device. This is typically done with a dedicated USB-UART converter chip, such as those from Future Technologies Devices International (FTDI). These chips are extremely common in embedded electronics, and the FT2232 family is particularly popular with FPGAs. This chip exposes two data channels to the host machine over a single USB port, allowing for one channel to be used to program the FPGA, and the other to provide a UART interface to it. This allows one USB cable to provide programming, communication, and (for smaller FPGAs) power. The latest generation of the FT2232 is the FT2232H, and is the variant found on most Digilent FPGA development boards, including the Nexys A7 used in 6.205. The chip is also used on the iCEstick development board, which is popular within the open-source community for its compatibility with open-source EDA tools.

Although its internal bus can reach gigabit speeds, Manta's memory bandwidth is limited by that of the converter. The pervasive FT2232H advertises a maximum baudrate of 12 Mbps, however the variety of operating systems and device drivers practically limits this to 3 Mbps. Depending on operating system, the driver used for the FT2232 can be sourced from the OS vendor, FTDI, the open-source community, `libusb`-based userspace drivers, or even Vivado. As a result, 3 Mbps represents a reasonable maximum speed on most host machines and FPGA development boards. As such, it is used for the following analysis.[1]

Importantly, 3 Mbps refers to the speed of bits on the `tx` and `rx` lines, not the speed at which bits in Manta's memory are accessed. Despite bits being placed on the interface at 3Mbps, the effective memory bandwidth is far lower. For instance, a single read requires sending a 7 byte message, where each byte occupies 10 bits on the wire due to the start and stop bit. This causes 70 bits to be placed on the wire to read 16 bits from memory, and equates to an efficiency of 22%. Writes are slightly less

[1] It is believed this legacy behavior from the previous generation of the FT2232H (the FT2232D) which supported a maximum baudrate of 3 Mbps.

efficient, as writing 16 bits to memory requires sending an 11 bytes message, placing 110 bits on the wire. This still only accesses 16 bits of Manta's memory, netting an efficiency of 14.5%.

Due to this overhead, the 3Mbps bandwidth offered by the FT2232H is reduced to a maximum speed of 436.3kbps while writing to memory, and 685.7kbps while reading from it. To confirm these speeds were being obtained with the operating system, device driver, and serial library (pySerial) being used, a test was constructed to measure the effective bandwidth of the UART interface. This test involved sending transactions in bursts of varying lengths, and measuring the time between the arrival of the first and last transaction in the burst. As this procedure tests the software stack, the transfer time can not be accurately measured from the host machine. Instead, it was measured directly from the FPGA, by placing a Manta core on the FPGA along with a counter to measure elapsed time. The results of this test are presented in Figure 4-1. Although the data presented here was recorded on a Linux host, similar performance is seen on Windows and macOS devices.

Two notable behaviors are seen in the results of the test. First, the time needed to execute a transfer is relatively constant for small transfer sizes. This occurs around a transfer size of 8kbits, below which reads take ≈50ms and writes take ≈30ms, far slower than the theoretical speed. This is likely due to the many sets of buffers between user space and the device itself, each of which with its own flow control and interrupts. [22] It is suspected that some buffer is serviced more often once transfer sizes exceed 8kbits, but transfers larger than this converge to a fixed speed.

Second, the write performance converges to its theoretical value, but the read performance does not. Rather, the read performance converges to a fixed speed nearly half as fast as the theoretical value. This is because pySerial, in its current usage, is single-threaded. To read from a memory address, the host sends a message to the FPGA requesting the read, to which the FPGA issues a response with the requested data. Currently, a single thread on the host sends all requests, and then receives all responses. This causes an approximate halving of the effective bandwidth, as the latency between buffers must be incurred twice. This could be solved by spawning a

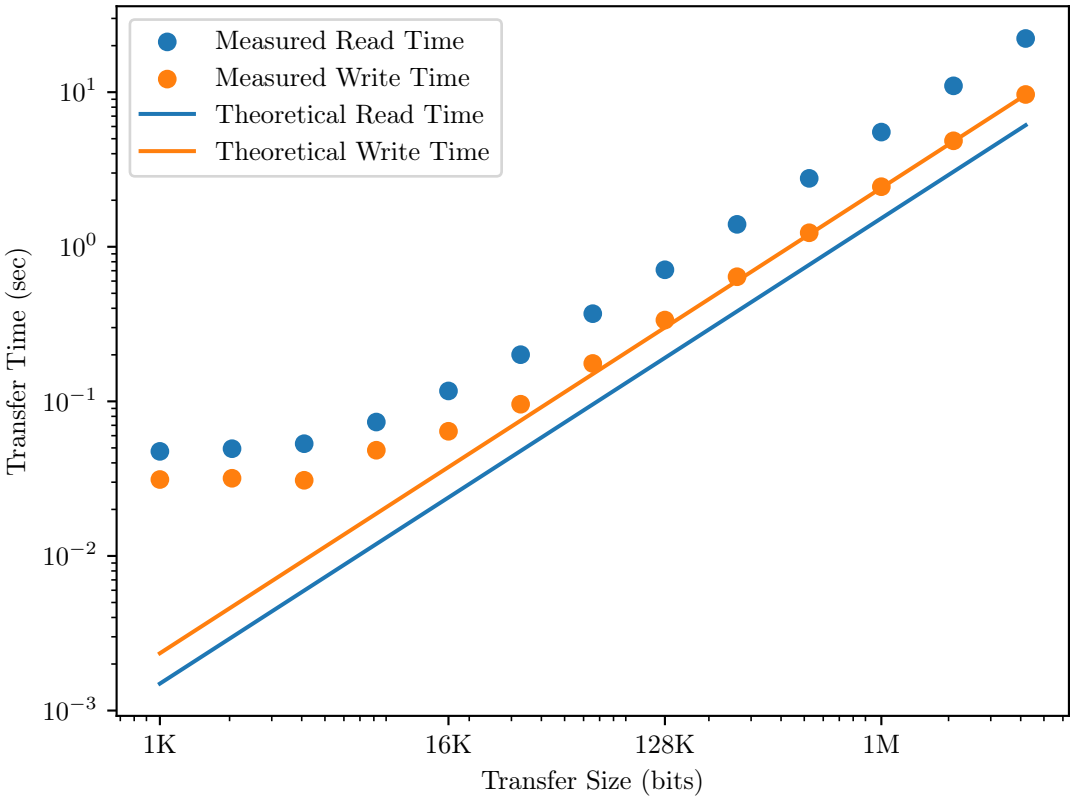

Figure 4-1: Memory bandwidth between the host machine and FPGA over a UART interface. Each point represents the fastest transfer out of 5 identical runs.

separate thread to read responses while the main thread sends requests.

Despite the inefficiencies, it is worth contextualizing these bandwidths. The XC7A100T present on the Nexys A7 used in 6.205 contains 4,860kbits of block ram. [14] With the speeds measured in Figure 4-1, writing to all of the block memory on the chip would take 11.4 seconds, and reading the entire BRAM would take 7.25 seconds. Rarely would a logic analyzer core need this much memory. The largest configuration tested in **??** uses 16 32kbit BRAM primitives, which takes 2.7 seconds to read, and 1.2 seconds to write. This is considered fast enough to provide a debugging experience on par with the Xilinx ILA.

4.2 Resource Utilization and Build Time

4.2.1 Logic Analyzer Core

The performance of the Logic Analyzer core was evaluated by measuring its resource utilization on the FPGA, and build time in Vivado. This was expected to vary with sample depths and probe sizes, so three test configurations were chosen with parameters representative of use cases in 6.205. These configurations consisted of a *Wide* core, which recorded 1024 samples of 4 probes, each 16 bits in width. This was chosen to demonstrate the core's performance with many input signals, sampled for a relatively short time. This test case is complemented by the *Deep* configuration, which samples a few signals over a long time, recording 32678 samples of 4 probes, each 4 bits in width. Finally, a *Nominal* configuration was chosen as an intermediary between the two, recording 4096 samples of 4 probes, each 8 bits in width. As 6.205 students are likely to use Manta over UART than Ethernet, the UART interface was selected for each core. For comparison, Xilinx ILAs with equivalent configurations were also tested.

The cores were implemented on the Nexys A7, and built by Vivado 2022.2 running on an Ubuntu 22.04 server equipped with a Ryzen 2700X, 16GB of RAM, and 512GB of SSD storage connected over NVMe. To prevent unconnected logic from being optimized out of the design, each core's probes were connected to a simple counter. This ensured that the inputs were driven by onboard logic, with as little added resource consumption as possible.

It is worth mentioning that Vivado supports building source code in either *project* or *non-project* mode. The choice between the two does not affect the resources used by the design, but builds in non-project mode are often significantly faster. Manta's Logic Analyzer and the Xilinx ILA may be built in either mode, but the ILA is typically built, configured, and run in project mode. However, 6.205 typically builds student projects in non-project mode. To provide a fair comparison, the ILA is built in project mode only, and Manta is built in both. This allows both typical workflows to be properly represented, but isolates any performance improvements due to building

in non-project mode. This allows for a fair evaluation of build times between the ILA and Logic Analyzer Core, shown in Figure 4.1.

Configuration	Core	Build Time	LUT	FF	BRAM
Wide	Xilinx ILA (project mode)	2m 50s	1283	2230	2
1024 Samples	Manta (project mode)	1m 57s	532	610	2
4 Probes	Manta (non-project mode)	1m 35s	532	610	2
16 bits each	**Improvement**	**44%**	**58%**	**73%**	**0%**
Nominal	Xilinx ILA (project mode)	2m 47s	1206	2093	4
4096 Samples	Manta (project mode)	1m 57s	372	476	4
4 Probes	Manta (non-project mode)	1m 35s	372	476	4
8 bits each	**Improvement**	**43%**	**69%**	**77%**	**0%**
Deep	Xilinx ILA (project mode)	2m 53s	1238	2054	16
32678 Samples	Manta (project mode)	1m 58s	402	427	16
4 Probes	Manta (non-project mode)	1m 37s	402	427	16
4 bits each	**Improvement**	**44%**	**68%**	**79%**	**0%**

Table 4.1: Performance comparison between the Xilinx ILA and Manta's Logic Analyzer Core, in terms of build time and FPGA resource utilization. Each of the build times shown represents the fastest of five runs.

In every case tested, Manta's Logic Analyzer core builds faster and uses fewer resources than the Xilinx ILA. Manta used between 58-81% fewer LUTs, and reduced flip-flop usage by up to 90%. In all cases, Manta used the same number of BRAM primitives as the ILA, and when built in non-project mode, took 44% less time. This represents a sizable increase in performance relative to the ILA, and meets the design objective presented in Section 1.2 of complementing vendor tools.

Additionally, the performance of the Logic Analyzer Core was measured relative to itself. This was done with a parameter sweep over the core's configuration variables: sample depth, number of probes, and probe widths. Three values were chosen for each, and for simplicity, all probes were set to the same width. The build times and resource utilization of these configurations are shown in Table 4.2, 4.3, and 4.4.

		Number of Probes (Sample Depth = 1024)		
		4	8	16
Width	4	Build Time: 1m 29s LUT: 298 FF: 397 BRAM: 0.5	Build Time: 1m 33s LUT: 386 FF: 486 BRAM: 1	Build Time: 1m 34s LUT: 532 FF: 610 BRAM: 2
	8	Build Time: 1m 39s LUT: 400 FF: 488 BRAM: 1	Build Time: 1m 43s LUT: 560 FF: 634 BRAM: 2	Build Time: 1m 42s LUT: 842 FF: 926 BRAM: 4
	16	Build Time: 1m 48s LUT: 560 FF: 670 BRAM: 2	Build Time: 1m 42s LUT: 901 FF: 966 BRAM: 4	Build Time: 1m 54s LUT: 1436 FF: 1558 BRAM: 8

Table 4.2: Resource consumption and build time of the Logic Analyzer Core with a sample depth of 1024. Each of the build times shown represents the fastest of three runs.

		Number of Probes (Sample Depth = 2048)		
		4	8	16
Width	4	Build Time: 1m 31s LUT: 301 FF: 400 BRAM: 1	Build Time: 1m 36s LUT: 388 FF: 472 BRAM: 2	Build Time: 1m 35s LUT: 540 FF: 616 BRAM: 4
	8	Build Time: 1m 36s LUT: 409 FF: 492 BRAM: 2	Build Time: 1m 44s LUT: 560 FF: 640 BRAM: 4	Build Time: 1m 41s LUT: 845 FF: 936 BRAM: 8
	16	Build Time: 1m 49s LUT: 567 FF: 676 BRAM: 4	Build Time: 1m 43s LUT: 902 FF: 976 BRAM: 8	Build Time: 1m 56s LUT: 1443 FF: 1577 BRAM: 16

Table 4.3: Resource consumption and build time of the Logic Analyzer Core with a sample depth of 2048. Each of the build times shown represents the fastest of three runs.

		Number of Probes (Sample Depth = 4096)		
		4	8	16
Width	4	Build Time: 1m 31s LUT: 288 FF: 403 BRAM: 2	Build Time: 1m 35s LUT: 372 FF: 476 BRAM: 4	Build Time: 1m 36s LUT: 531 FF: 622 BRAM: 8
	8	Build Time: 1m 38s LUT: 401 FF: 496 BRAM: 4	Build Time: 1m 41s LUT: 552 FF: 646 BRAM: 8	Build Time: 1m 42s LUT: 838 FF: 947 BRAM: 16
	16	Build Time: 1m 49s LUT: 554 FF: 682 BRAM: 8	Build Time: 1m 44s LUT: 890 FF: 987 BRAM: 16	Build Time: 1m 56s LUT: 1425 FF: 1601 BRAM: 32

Table 4.4: Resource consumption and build time of the Logic Analyzer Core with a sample depth of 4096. Each of the build times shown represents the fastest of three runs.

Notably, build time appears to scale with the width of the probes, and is relatively invariant to their number or sample depth. This behavior is also seen for similar tests performed on the Block Memory Core, suggesting that this behavior could be the result of the block memory used in the core. The mechanism that drives this behavior is unknown, and would require further analysis.

Further, BRAM utilization is invariant with the organization of the BRAM, and scales only with the number of bits used. This can be seen by the configurations represented on the upper diagonal of Tables 4.2, 4.3, and 4.4, which use the same amount of BRAM for a given sample depth, regardless of if the core has (4) 16-bit, (8) 8-bit, or (16) 4-bit probes. This is likely the result of optimizations done during the build, which shows that the BRAMs are being correctly inferred by Vivado.

4.2.2 IO Core

Tests similar to those run on Logic Analyzer core (4.2.1) were also performed on the IO core. Resource utilization was also expected to vary with the size and number of probes, so three test configurations mirroring those performed on the Logic Analyzer core were chosen. These also represent typical workloads seen in 6.205, and consist of a *Thin*, *Nominal*, and *Wide* configuration. Each configuration contains four input and output probes, but vary the width of the probes between 4, 8, and 16 bits respectively.

The performance of the IO core is compared to an equivalent Xilinx VIO core, just as the Logic Analyzer was compared to the ILA in Section 4.2.1. The input and output probes were connected to each other to prevent unconnected logic from being optimized out of the design. This adds no additional logic to the design, and therefore does not influence resource utilization on the FPGA. The designs were built on the same machine described in Section 4.2.1, which was done in both project and non-project mode for the same reasons described there. The results of the test are presented in Table 4.5.

Configuration	Core	Build Time	LUT	FF	BRAM
Wide	Xilinx VIO (project mode)	2m 07s	776	146	0
4 Input Probes	Manta (project mode)	1m 41s	145	195	0
4 Output Probes	Manta (non-project mode)	1m 21s	145	195	0
16 bits each	**Improvement**	**37%**	**81%**	**86%**	**0%**
Nominal	Xilinx VIO (project mode)	2m 05s	666	1227	0
4 Input Probes	Manta (project mode)	1m 41s	125	139	0
4 Output Probes	Manta (non-project mode)	1m 20s	125	139	0
8 bits each	**Improvement**	**35%**	**81%**	**89%**	**0%**
Thin	Xilinx VIO (project mode)	2m 05s	612	1110	0
4 Input Probes	Manta (project mode)	1m 41s	117	111	0
4 Output Probes	Manta (non-project mode)	1m 20s	117	111	0
4 bits each	**Improvement**	**36%**	**81%**	**90%**	**0%**

Table 4.5: Performance comparison between the Xilinx VIO and Manta's IO Core, in terms of build time and FPGA resource utilization. Each of the build times shown represents the fastest of five runs.

These observed performance is similar to that of the Logic Analyzer core. In

every case tested, Manta's IO core builds faster and uses fewer resources than an equivalent Xilinx VIO core. Manta builds around 36% faster, uses 81% fewer LUTs and between 86-90% fewer flip-flops, all while sporting no increase in BRAM usage. This is a significant performance improvement, and demonstrates that Manta meets the performance objectives outlined in Section 1.2.

Although they may be promising, care should be taken to not interpret these results beyond the test cases presented. The ILA and VIO cores include a piece of logic Xilinx refers to as a Debug Hub, through which all debug cores communicate. This includes the ILA and VIO in addition to other utilities. As a result, the results presented in Figure 4.1 may reflect the design of the Debug Hub, and not the ILA or VIO cores. The design of the Debug Hub is not public and its source files are encrypted, so broad conclusions are not possible. However, it is clear that for typical 6.205 use cases, Manta builds faster and uses fewer resources on the FPGA, meeting the design goals presented in Section 1.2. [20]

Tests of the IO Core against itself were also performed. This was done with a parameter sweep over two variables: the number of probes on the core, and their width. For simplicity, each probe was assigned the same width, and each configuration was built in non-project mode. The resulting build times and resource utilization is shown in Table 4.6.

The results show a consistent build time across all runs, regardless of number of width of the probes. It is believed this represents some kind of lower bound in the build process, where the additional logic specified by larger designs does not contribute substantially to the observed performance. Further tests with more, larger probes are needed to determine this. Despite this, the parameters chosen represent typical cases encountered in 6.205, and it may be concluded that build time is roughly invariant for designs of this size.

		Number of Probes		
		4	8	16
Width	4	Build Time: 1m 20s LUT: 117 FF: 111 BRAM: 0	Build Time: 1m 21s LUT: 125 FF: 139 BRAM: 0	Build Time: 1m 21s LUT: 145 FF: 195 BRAM: 0
	8	Build Time: 1m 21s LUT: 127 FF: 127 BRAM: 0	Build Time: 1m 21s LUT: 144 FF: 171 BRAM: 0	Build Time: 1m 22s LUT: 180 FF: 259 BRAM: 0
	16	Build Time: 1m 21s LUT: 139 FF: 159 BRAM: 0	Build Time: 1m 22s LUT: 163 FF: 235 BRAM: 0	Build Time: 1m 21s LUT: 215 FF: 387 BRAM: 0

Table 4.6: Resource consumption and build time of the IO Core. Each of the build times shown represents the fastest of three runs.

The results also show a pattern in the usage of LUTs and Flip-Flops. Configurations presented on the ascending diagonal of Table 4.6 have similar LUT and Flip-Flop usage, regardless of if they are configured as (4) 16-bit probes, (8) 8-bit probes, or (16) 4-bit probes. This scaling is approximate, and additional testing is necessary to determine the mechanism which presents this behavior.

Lastly, no BRAM primitives onboard the FPGA are utilized in any case. The IO core contains its entire state in registers, and therefore no BRAMs are instantiated anywhere in the design.

4.2.3 Block Memory

Although the Block Memory Core has no equivalent Xilinx product, its resource utilization can still be measured relative to itself. Similarly to the previous comparisons, the resource utilization of the core is dependent on parameters chosen at compile-time, in this case the width and depth of the memory. As a result, the utilization of the core was measured through a parameter sweep, where four values were chosen for the memory width and depth, producing 16 unique combinations. These were built with Vivado in non-project mode, where a simple counter drove the block memory's user port, preventing any unused logic from being optimized out of the design. The results of the test are presented in Table 4.7.

		Depth			
		256	512	1024	2048
Width	8	BT: 1m 22s LUT: 148 FF: 181 BRAM: 0.5	BT: 1m 22s LUT: 153 FF: 191 BRAM: 0.5	BT: 1m 22s LUT: 152 FF: 193 BRAM: 0.5	BT: 1m 23s LUT: 156 FF: 195 BRAM: 1
	32	BT: 1m 23s LUT: 156 FF: 238 BRAM: 1	BT: 1m 23s LUT: 156 FF: 241 BRAM: 1	BT: 1m 23s LUT: 156 FF: 244 BRAM: 1	BT: 1m 23s LUT: 169 FF: 247 BRAM: 2
	128	BT: 1m 26s LUT: 204 FF: 484 BRAM: 4	BT: 1m 25s LUT: 204 FF: 493 BRAM: 4	BT: 1m 26s LUT: 204 FF: 502 BRAM: 4	BT: 1m 26s LUT: 207 FF: 511 BRAM: 8
	512	BT: 1m 58s LUT: 345 FF: 1469 BRAM: 16	BT: 1m 58s LUT: 352 FF: 1502 BRAM: 16	BT: 2m 0s LUT: 357 FF: 1535 BRAM: 16	BT: 1m 57s LUT: 357 FF: 1568 BRAM: 32

Table 4.7: Resource consumption and build time of the Block Memory Core. Build times are abbreviated by `BT`, each value shown represents the fastest of three runs.

Notably, the build time and usage of LUTs and Flip-Flops vary only with memory width. The mechanism behind this behavior is not currently well understood, but it is suspected this is due in some part to the series of small, 16-bit BRAMs instantiated in the Block Memory Core. As the width of the memory is increased, more BRAMs

are instantiated inside the core, presenting a greater load on Vivado's block memory optimizer. This combines multiple block memories into a single BRAM primitive, and it is believed this optimization increases the build time. However, this is merely speculation, and further profiling will be required to obtain conclusive results.

Chapter 5

Conclusions

This thesis presents Manta, a tool for debugging programmable logic on FPGAs. Manta is a lightweight, modular, platform-independent, and intuitive tool that serves to complement vendor-provided debugging tools in educational, professional, and hobbyist settings. This is done with a series of cores on a daisy-chained bus, which allow users to debug their designs with logic analyzers, block memory, and direct access to signals on the FPGA. The behavior of these cores can be easily extended through the provided Python API, allowing for easy development of custom applications.

During development Manta was beta tested by several 6.205 alumni, whose initial impressions of the tool were invaluable in guiding its development. Many thanks are due to Brady Sullivan, Dev Chheda, Ivy Liu, Jan Park, and Jordan Wilke for the wisdom shared and bugs discovered.

Although the work presented here represents a complete system that meets its design goals, much further work is available that would enhance the reliability, utility, and accessibility of the tool.

5.1 Future Work

5.1.1 Formal Verification

A chain is only as strong as its weakest link, and Manta's daisy-chained internal bus is no exception. If any module mishandles a message and produces incorrect data, all modules downstream are affected, which effectively disables the tool. This is most likely to occur in the receive or transmit bridges, which are the most complex modules in the chain and thus provide the most space to harbor bugs.

Formally verifying the behavior of these modules would improve Manta's reliability. Formal verification allows for a mathematical proof the hardware's behavior, ensuring that no invalid messages are placed on Manta's internal bus, and no invalid messages are sent back to the host. This can be extended to the rest of the logic generated by Manta. Although the receive and transmit bridges likely harbor the most bugs, formally verifying each core's handling of bus transactions would also be extremely useful. This would create a bus that is guaranteed to produce correctly-formatted responses to correctly-formatted requests. This would mean bugs could only exist in the cores' handling of user logic, which again could be exhumed with formal verification.

However, the universality of these guarantees comes at the cost of development time. Properly verifying hardware is a nuanced and time-intensive process, and as a result has not yet been performed on Manta's hardware. However, modern open-source tools such as SymbiYosys are very accessible, and can easily be used by future developers of the tools presented in this work.

5.1.2 Additional Link Layers

To gain relevance in industrial settings, Manta absolutely needs to support PCI Express (PCIe) and 10 Gigabit Ethernet (10GbE). 100Mbps Ethernet and 3Mbps serial will likely be more than sufficient for educational settings, but most FPGAs used at professional scale are accelerator cards that connect over PCIe, or are connected to

a very high-speed network in some fashion. Both PCIe and 10GbE are packet-based protocols, the latter being a later revision to the MAC presented in this text. As a result both protocols have physical, data link, and transport (or transaction) layers. Typically the physical layer is usually handled by interface transceivers external to the FPGA, and the data link layer is simple enough to be implemented in hardware. However, communication at the transaction layer is very stateful, and is usually handled by a soft microprocessor. This is subtly acknowledged by the work presented here - Manta's Ethernet stack only supports a L2 MAC as higher-level protocols are incredibly difficult to implement in hardware.

This can be avoided with a handful of assumptions and careful systems design [23], but the added complexity conflicts with Manta's core design goal of simplicity. Tastefully implementing these higher layers to balance simplicity and usability is the subject of much further work.

5.1.3 Name-brand bus endpoints

Although using a "name brand" bus such as AXI, Avalon, or Wishbone for Manta's internal bus was foregone for the reasons specified in Section 3.2.2, being able to interface with these busses is still useful. A Manta core exposing a controller or peripheral on a "brand-name" bus would allow users to quickly test IP, allowing them to confidently use logic sourced from commercial vendors or open-source repositories like OpenCores. This would give digital designers a shortcut when bringing up logic for the first time, increasing productivity and saving time.

This feature would also be useful to users developing their own bus-attached IP. Bus traffic could be federated by a bus controller core, while also being sniffed by a Logic Analyzer Core. This would make Manta able to debug at the individual signal level with the Logic Analyzer core, the transaction level with the bus controller core, and at the application level with its Python API. Being able to debug across this many levels of abstraction concurrently is not a common feature of many FPGA tools. Xilinx provide some tools that allow a host machine to control an onboard bus, but these only target AXI controllers. [16]

5.1.4 Data Bus Improvements

Although Manta's internal bus was designed for simplicity, it could be made simpler. Manta includes separate channels for read and write data, but only one is ever in use at a time. To save resources on the FPGA, these could be combined into a single data channel, where read and write operations are differentiated with the `rw` signal, as is done currently. This means Manta would place data read from memory on `data` during a read, and write the contents of `data` to memory during a write. This would significantly lighten the resource utilization of the bus, as 16 bits are saved on every connection between a pair of devices. For a daisy-chained bus topology with N nodes, this is $16 \times (N - 1)$ bits saved.

Additionally, Manta's 16-bit wide address bus presents it with some limitations. Manta's Block Memory and Logic Analyzer Cores place a significant amount of memory on the bus, which rapidly consumes address space. This is exacerbated by memories wider than 16 bits, whose inefficient packing can be rather wasteful with address space, as described in Section 3.5.2. Even if the memory was packed perfectly, only $16 \times 2^{16} \approx 1Mbit$ worth of memory could be addressed by the bus, which is less than a fourth of the block memory available on the Nexys A7 used in 6.205. Clearly, more bits are needed

This raises the question of how large the address bus should be. Clearly 16 bits is insufficient, and an address width of 32 bits would be the next most obvious choice. However this is overkill for smaller designs, and would increase Manta's resource utilization and build time.

An alternative would be to set the width of the bus when Verilog is generated. Manta's Python API could size the address bus to be just wide enough to support the cores in the current configuration, and no wider. This would use no more FPGA resources than absolutely necessary, but at the expense of greater complexity during HDL generation. The current structure of the Python API would easily allow this, but a significant amount of code would need to be updated, making this optimization time-intensive to implement.

5.1.5 Waveform Processing Tools

Traditional benchtop logic analyzers typically include protocol decoders, which decode digital protocols such as I2C, SPI, UART, CAN, and Ethernet from signals connected to the logic analyzer. The resulting bus traffic is usually presented as another trace on the waveform display, and can also be used in the analyzer's trigger flow. This allows for captures to begin on conditions such as a write to a specific address over I2C, the start of a CAN frame, or a particular byte being sent over UART. This is remarkably useful, as it allows for quick verification that the bus is behaving as expected.

Manta may benefit from features similar to these. However, some flexibility is offered in their implementation as Manta consists of both a hardware component on an FPGA, and a software component on a host machine. The protocol decoder could be implemented on the FPGA, processing signals in real-time and optionally using them as triggers in the Logic Analyzer Core. This would be the most similar in function to traditional logic analyzers, but would require more resources on the FPGA.

Alternatively, this analysis could be done on the host machine, after the data has been captured. This would save the resource utilization on the FPGA, but sacrifice the ability to decode in real-time. This would also allow users to perform completely arbitrary analysis, as operations performed on the captured data would not be limited to what Manta could place on the FPGA. Some external software tools are available for this, such as the very modern Waveform Analysis Language [21].

Depending on circumstance either approach may be the most appropriate, but both would be a welcome addition to Manta's feature set.

5.1.6 Migrating to Verilator

As part of its automated testing, Manta runs a series of tests on the Verilog templates used to generate its hardware. This is done in Icarus Verilog, which functions by compiling a set of source files into an executable, which is then run in a custom environment called vvp. This environment is tailor-made for Icarus, and does not

support the addition of user code beyond the Verilog source.

However, this is not the case for other open-source simulators, such as Verilator. It works by transpiling Verilog source to a C++ or SystemC model, which the user includes in their own source code. Typically the user's source is little more than a wrapper to set up and run the test, but this workflow allows for user applications that directly access the logic defined in the Verilog.

One such application is a virtual serial port, which provides a file from which bytes may be read and written to - just like a physical serial port's device driver does. Operations on this file can be passed along to the device model generated by Verilator, meaning communications sent over the virtual serial port appear identical to those sent over a physical serial port to real hardware. This method was pioneered by Dan Gisselquist, who has used it to great success [8].

Using a virtual serial port would allow Manta to be tested completely end-to-end in simulation. Currently, Manta only tests individual components of its hardware in simulations isolated from each other. Using Verilator for simulation would allow a complete core to be tested in tandem with its Python API, allowing for a complete integration test to be done entirely in simulation.

This would be a most welcome improvement, as presently Manta can only be tested end-to-end in hardware. Trying to perform tests that accomodated this produced no shortage of DevOps challenges, as automated tests required access to both a physical FPGA as well as commercially-licensed, platform-constrained build tools. Integrating this in GitHub Actions was remarkably difficult, and verifying equivalent behavior of the tests across Windows, macOS, and Linux hosts was impossible. Using virtual serial ports would remove this dependency on physical hardware, meaning the entire system can be integration tested across multiple operating systems in parallel with lightweight, open source tools.

5.1.7 FuseSoC Integration

FuseSoC is a open-source package manager and build system for hardware designs. Here, "packages" are modules written in HDL, developed by the community and

cataloged by FuseSoC. This allows users to include and manage external code in a manner similar to pip, cargo, or npm, allowing easy reuse and distribution of code across hardware projects.

This has greatly matured the state of open source hardware. Previously, most community-developed cores were small project-specific designs with very similar purposes, but little portability. Providing a unified frontend allowed the community to focus its effort, deduplicating effort and building a mature catalog of portable hardware designs.

It is believed that adding Manta to FuseSoC's catalog direct some of this attention towards the tool. FuseSoC has its own set of features for automatically generating these cores, which it also specifies with a YAML configuration file. Little more than a wrapper script would be necessary to properly integrate with FuseSoC. This would allow for even smoother usage of Manta, while also expanding its user base and soliciting developer attention.

Appendix A

Online Documentation

This thesis devotes heavy attention to Manta's design and the reasoning behind it, as well as its performance. However, a description of the practical nuts-and-bolts of working with the tool is notably absent. This is intentional, and is done to keep this thesis relevant and readable. As the tool continues to evolve, the commands run in a user's shell and functions imported from the API are guaranteed to change, but the overarching design of the tool will likely remain relatively constant. Describing the design in proper detail with full context is difficult to do in online documentation, just as it is difficult to describe methods of an API in a thesis. As a result, these works are separate.

The online documentation may be accessed at `https://fischermoseley.github.io/manta/`. If this domain ever becomes unavailable, users may build the website locally by cloning Manta's source code from `https://github.com/fischermoseley/manta`, and running `make serve_docs` at the command line after installing Material for MkDocs with `pip install mkdocs-material`. If that is not possible, then the site Markdown source may be read directly from the `doc/` folder in Manta's GitHub repository.

Bibliography

[1] Intel Corporation. Adding simulator-aware signal tap nodes. https://www.intel.com/content/www/us/en/docs/programmable/683819/22-4/adding-simulator-aware-signal-tap-nodes.html, December 2022. Accessed: 2023-05-03.

[2] Intel Corporation. Intel quartus prime pro edition user guide: Debug tools - signal tap trigger flow description language. https://www.intel.com/content/www/us/en/docs/programmable/683819/22-4/trigger-flow-description-language.html, December 2022. Accessed: 2023-05-03.

[3] Intel Corporation. Sequential triggering. https://www.intel.com/content/www/us/en/docs/programmable/683819/22-4/sequential-triggering.html, December 2022. Accessed: 2023-05-03.

[4] Intel Corporation. Quartus prime 21.1. https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime/resource.html, April 2023. Accessed: 2023-05-02.

[5] Laurentiu-Cristian Duca. openverifla. https://github.com/laurentiuduca/openverifla/tree/master, October 2022. Accessed: 2023-05-04.

[6] Dan Gisselquist. Finishing off the debugging bus: Building a software interface. https://zipcpu.com/blog/2017/06/29/sw-dbg-interface.html, June 2017. Accessed: 2023-05-04.

[7] Dan Gisselquist. Getting started with the wishbone scope. http://zipcpu.com/blog/2017/07/08/getting-started-with-wbscope.html, July 2017. Accessed: 2023-05-04.

[8] Dan Gisselquist. Taking a new look at verilator. https://zipcpu.com/blog/2017/06/21/looking-at-verilator.html, June 2017. Accessed: 2023-05-16.

[9] Dan Gisselquist. wbscope. https://github.com/ZipCPU/wbscope, January 2021. Accessed: 2023-05-04.

[10] Dan Gisselquist. dbgbus. https://github.com/ZipCPU/dbgbus, February 2023. Accessed: 2023-05-04.

[11] Kevin Hubbard. sump2 – 96 msps logic analyzer for $22. https://blackmesalabs.wordpress.com/2016/10/24/sump2-96-msps-logic-analyzer-for-22/, October 2016. Accessed: 2023-05-04.

[12] Opal Kelly Incorporated. Frontpanel user's manual. https://assets00.opalkelly.com/library/FrontPanel-UM.pdf, March 2015. Accessed: 2023-05-05.

[13] Xilinx Incorporated. Virtual input/output v3.0 product guide. https://docs.xilinx.com/v/u/en-US/pg159-vio, April 2018. Accessed: 2023-05-05.

[14] Xilinx Incorporated. 7 series fpgas data sheet: Overview. https://docs.xilinx.com/v/u/en-US/ds180_7Series_Overview, September 2020. Accessed: 2023-05-16.

[15] Xilinx Incorporated. Chipscopy. https://xilinx.github.io/chipscopy/2021.1/index.html, November 2021. Accessed 2023-05-05.

[16] Xilinx Incorporated. Jtag to axi master v1.2. https://docs.xilinx.com/v/u/en-US/pg174-jtag-axi, February 2021. Accessed: 2023-05-16.

[17] Xilinx Incorporated. Vivado 2022.2. https://www.xilinx.com/support/download.html, October 2022. Accessed: 2023-05-01.

[18] Xilinx Incorporated. Vivado design suite 7 series fpga and zynq-7000 soc libraries guide. https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries/Introduction, October 2022. Accessed: 2023-05-06.

[19] Xilinx Incorporated. Chipscopy. https://github.com/Xilinx/chipscopy, January 2023. Accessed: 2023-05-04.

[20] Xilinx Incorporated. Vivado design suite user guide. https://docs.xilinx.com/v/u/en-US/ds180_7Series_Overview, May 2023. Accessed: 2023-05-16.

[21] Lucas Klemmer and Daniel Große. WAL: a novel waveform analysis language for advanced design understanding and debugging, 2022.

[22] David S. Lawyer. The linux documentation project: Serial howto. https://tldp.org/HOWTO/html_single/Serial-HOWTO/, February 2011. Accessed: 2023-05-16.

[23] Fischer Moseley and Jay Lang. 6.111 final project - network attached laser projector. https://fischermoseley.com/laser_projector.pdf, December 2020. Accessed: 2023-05-16.

[24] Massachusetts Institute of Technology. Course catalogue of the massachusetts institute of technology 1968 - 1969. http://hdl.handle.net/1721.3/98443, July 1968. Accessed: 2023-05-04.

[25] ultraembedded. openlogicbit. https://github.com/ultraembedded/openlogicbit, June 2021. Accessed: 2023-05-04.