UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

How to build an FPGA based PCIe fuzzer

Wie man einen FPGA basierten PCIe Fuzzer baut

Bachelorarbeit

im Rahmen des Studiengangs
Robotik und Autonome Systeme
der Universität zu Lübeck

vorgelegt von
Leon Christopher Dietrich

ausgegeben von
Prof. Dr. Thomas Eisenbarth

betreut von
Thore Tiemann

Lübeck, den 24. November 2021

Abstract

Modern high performance electronic devices often rely on PCIe to perform inter-chip communications in a fast manner. Due to increasing demand for new features with the goals such as improving the throughput even further, reducing power consumption or adding new administrative and virtualization capabilities the already complex and organically grown PCIe protocol is becoming even more complex. Furthermore the once internal protocol is exposed to outside connections for quite some time now and reprogrammable devices are exposed to both external connections as well as the PCIe network for an equally long period of time.

Having a PCIe fuzzer capable of sending arbitrary TLP messages might prove to be a valuable tool for analysing the exposed interfaces and thus hardening the protocol. In this thesis we are going to describe our path towards implementing one using an Intel Agilex F-Series Field Programmable Gate Array development kit with an PCIe x16 interface capable of being modified to send arbitrary TLP messages.

Zusammenfassung

Moderne Hochleistungselektronik setzt häufig auf PCIe um eine schelle Kommunikation zwischen einzelnen Computerchips zu gewährleisten. Aufgrund der steigenden Anforderungen an die Geschwindigkeit, Virtualsierungsfunktionsdichte und Energiespareigenschaften, wird das, ohnehin schon sehr komplexe und organisch gewachsene, PCIe Protokoll zusehends komplizierter. Erschwerend kommt hinzu, dass dieses Protokoll, welches historisch betrachtet, nicht dazu gedacht war, mit der Außenwelt in Kontakt zu stehen, seit einiger Zeit auch für externe Verbindungen genutzt wird. Zu guter Letzt sind außerdem rekonfigurierbare Komponenten, welche an das PCIe Netz eines Gerätes angeschlossen sind, keine Seltenheit mehr.
Die Fähigkeit, mithilfe eines FPGA basierten PCIe Fuzzers, diese offenen Schnittstellen untersuchen zu können, könnte sich in der Zukunft als ein Wertvolles Werkzeug erweisen, um das PCIe-Protokoll und die verbundenen Geräte ab zu sichern. In dieser Arbeit beschreiben wir, wie wir einen solchen Fuzzer auf Basis eines Intel Agilex F-Series FPGA Entwicklungsbausatzes gebaut haben, welcher in der Lage ist, mithilfe der PCIe x16 Schnittstelle beliebige modifizierte TLP Datagramme zu versenden.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 24. November 2021

Preface

Acknowledgements

First of all I would like to thank Thore Tiemann for all of his help, guiding me towards completing this thesis. Also, I would like to thank Ingrid Schumacher and Anja Rabich for proof-reading my thesis draft. Furthermore I would like to thank my girlfriend for being patient with me during the last weeks of this work. Finally I really need to thank the other students in the students lab for not killing me whenever the FPGA decided to go full haywire with its cooling fans.

To the reader of this thesis

Dear reader of this thesis, I would like to thank you for taking your time. Assuming that you are not engineering electronic circuits or writing software on a daily basis I decided to include little info boxes explaining things in order to assist you. If you however are familiar with the described terms you are welcome to skip those as they will not contain anything beyond explaining these terms.
Furthermore I have appended an quite extensive glossary at the end of this thesis. All technical abbreviations should be explained there for reference purposes.

Contents

1 Introduction

Prior to understanding how we build a fuzzer we may discuss why we did it.

1.1 What is this PCIe thing all the cool kids are talking about?

First of all we will gain some insight on what PCIe is and why it is an interesting thing to look at from a security stand point.

1.1.1 What is PCIe and where does it come from?

PCIe (which is short for "Peripheral Component Interconnect - Express") is a switched network connecting various components on and between printed circuit boards (PCB) enabling them to communicate with each other. The main advantages of using PCIe instead of any are interconnect, are that PCIe is extremely wide spread which results in great compatibility of components and the high data transfer rates it achieves while providing comparatively stable signaling.
Modern PCIe does not have a lot in common with its first predecessor presented by Intel in 1991. Most notably it is not a BUS anymore[1]. Instead it consists of multiple serial connections (referred to as "lanes") between individual components (a single root port, multiple switches and multiple end points and nowadays even firewalls). All of these components route packages to their requested destination but they behave differently doing so:

- An **End Point** is a member of a PCIe network whose job is to provide an actual device (for example a NIC) to the PCIe network (and thus to the computer we use). Almost all PCIe end points provide some kind of MMU in order to provide access to memory mapped I/O (see info box 1.1.1).
- The **Root Port** is a special kind of end point. It is the one that enumerates all other devices on the PCIe bus, configures them and has the final word on power management. Usually a CPU acts as the root port.
- A **Switch** is a device providing only minimal configuration registers. Its purpose is limited to forwarding packages received from other network members to their destination members. Some switches also perform certain packet processing techniques.

[1]Although it is still referred to as a "bus"

- A **Firewall** is a device that can be configured to only pass through packets if certain criteria are matched. Usually these criteria include TLP type, source and destination address. Some also perform advanced packet inspection and processing but those are quite rare.

Like most (if not all) modern electronic communication protocols, the PCIe protocol can be separated into layers each fulfilling a different task. The first layer (Layer 1) is referred to as the physical layer. Its job is providing a synchronous serial lane between two devices. This layer does not know about other PCIe lanes let alone other devices than the two it provides direct connectivity for. The second layer is referred to as the data link layer. Its job is to bind the individual physical lanes (usually $x1$, $x4$, $x8$, $x16$ or $x32$) into a logical link and perform basic error detection and correction tasks. It also makes sure that a packet that got lost between the two devices will be resend if the package is still within the buffer of the sending device. However this layer does not guarantee that a packet will not get lost or altered (due to transmission error or maliciously). The third layer is called the transaction layer. Its purpose is to send messages between devices with multiple hops. This layer is also supposed to ensure that broken packages will be resent and all packages arrive. There are multiple types of transaction layer packets with different purposes which are listed below (see Table 1.1) together with their type flags as listed within the PCIe spec[PCI06]. A more detailed description of a TLP[2] can be found at subsection 2.5.3,where we explain how to generate those. The fourth (and possibly further) layer(s) are application specific.

Info 1.1.1

Memory Mapped I/O is a methology where control registers (and even entire device memories) are mapped into the virtual address space of other devices. Using this method to configure and control devices is much faster and flexible than other methods (such as special processor instructions) used in the past since implementing a driver for a certain feature does not require interacting with special hardware registers. Instead a process can simply map the corresponding memory into its own address space and write to device registers like own variables[a]. This works both for kernel processes as well as userland processes as long as systems like the Linux `/dev/mem` wrapper exist.

[a]It is also common to declare static volatile variables for particular device registers

Besides the above mentioned primary types defined by the `Type` field, there exist further types (like power management or proprietary vendor specific ones), which are defined by the `Fmt` field and some type values are reserved for future use. There are also some "forbidden" combinations of `Type` and `Fmt` that should not be used, which are designed for error detection. Although these should not exist in real communication some vendors decided to use them for internal features.

[2]Transaction Layer Packet – Layer 3 of PCI. Basically all acronyms are explained when they are first used but can also be looked in the glossary.

Table 1.1: Most important TLP packet types

`Fmt`[7:5], `Type`[4:0] value	Name	Description
`"00B 0000B"h`	Memory read	Read the memory content at the specified address
`"01B 00000"h`	Memory write	Write to the specified memory location
`"000 00010"h`	I/O read	*Deprecated:* Read an I/O value
`"010 00010"h`	I/O write	*Deprecated:* Write to an I/O register
`"000 0010B"h`	Configuration read	Read a value from the destination devices PCIe configuration registers
`"010 0010B"h`	Configuration write	Write a value to a PCIe configuration register of the destination device
`"0BB 10BBB"h`	Message	Vendor specific PCIe messages. They are often used to implement interrupts

Hence they are usually not filtered as some devices would break without them and there are collisions in the meaning between different vendors. Last but not least there are combinations of `Type` and `Fmt` which are only mentioned as deprecated in the specification and some that are described as "confidential". — In short: It is a mess.

Info 1.1.2

Register notation: Specifying a name followed by $[A : B]$ means that the bits corresponding to said name are the bits A down to B. For Example "`Fmt`[7:5], `Type`[4:0]" means that the bits 7 to 5 of the byte correspond to `Fmt` and the bits 4 to 0 correspond to `Type`. Prior one saw the usage of bit strings (for example `"000 00010"h` or `"01B XZ010"h`). These are simply a list of all bits right next to each other. The first bit is the most significant (in this case 7) one and the last bit the least (in this case 0). A `0` refers to the logical '0' (respectively 'LOW'), `1` means a logical '1' or 'HIGH', `B` means that depending on the case the value might be '0' or '1', `Z` means high-impedance (I will get to that later) and last but not least `X` means "we do not care". The optimizer may choose either `'1'` or `'0'` – which ever way proves to be beneficial – in that last case. There are other notation symbols that are less common and will be introduced once they become necessary.
The `h` at the end of that string indicates that we are dealing with 9-way logic. If we are only dealing with binary logic, and want to emphasize this, we may write a `b` instead.
One thing to look out for: hardware people as well as some computer scientists start counting at 0. This means that the first bit is referred to as bit 0 and the last will be, for example, 31 if the word is 32 bit long in total.

1.1.2 Why is PCIe security important all the sudden?

So there are some chips in the computer that talk to each other — What is the deal? Until the early 21st century one could say that what happens on your motherboard stays on your motherboard. This is not true anymore. It has been shown that remotely compromised hardware may cause harm to your sensitive data[DPVL10][DPM11] and any device connected to the PCIe network may steal secrets from memory[MRG$^+$19]. It does not help that exposing PCIe over thunderbolt or USB4 is becoming mandatory[USB21]. Furthermore one can rent FPGAs, GPUs and other accelerators in the cloud. Assuming that one would be able to somehow send arbitrary messages to other devices from such accelerators one could theoretically compromise other devices and in turn steal or alter other parties (secret) data.
Last but not least there is a huge trend in the industry[VD19] to connect more and more devices with each other without having a CPU in the middle. This approach has the advantage of having less bottlenecks but enforces each device to care for its own security (and sometimes safety). As security considerations tend to become more complex from generation to generation and every device needs to implement the entire security stack perfectly, it is becoming more and more likely that a link in the chain breaks.
Unfortunately the efforts of PCIe hardware and software vendors do not seam to reflect these requirements well. Since in the future we can expect DMA (over PCIe) to be the default way to go with our increasing bandwidth demand, we should make sure we at least know what is going on.

1.1.3 Wait. They access your memory now and (almost) never tested it for bugs?

Of course we are not the first looking into PCIe security matters. There are plenty of people looking into operating system security in regards of PCIe. The most relevant are described below.

Bypassing IOMMU Protection against I/O Attacks This paper [MANK16] shows that an attacker having access to the PCI(e) bus using either their own hardware (or design on an FPGA) or compromising other hardware within the system may gain access to the systems main memory due to a lack of proper IOMMU (see info box 1.1.3) usage and misconfiguration. They show that the security model highly depends on correct usage of MMUs and show that (at least at the time of their writing) the Linux kernel failed to properly utilize the IOMMU.

Info 1.1.3

The **IOMMU**s are a special kind of MMU found on many `x86` based computers. Its purpose is to translate between a CPUs main memory and a virtual address space of a device attached to the PCIe network.

Thunderclap While the Thunderclap paper[MRG+19] was not the first to reveal massive issues with DMA security measures, they show that IOMMU protection on Intel based hardware can be bypassed due to bad implementation of this feature in most common operating systems (here: Microsoft Windows, Linux, macOS and FreeBSD). They focused their work on PCIe enabled external peripherals (they assumed this to be the only practical way to send untrustworthy PCI messages) but any device connected to the PCI(e) bus is capable of such actions. While some of the vulnerabilities they have found, are allegedly fixed by the operating system vendors, some of them are not. The authors also note that IOMMU protection is disabled by default in most operating systems due to performance issues[3]. Last but not least they show that complete protection from such attacks is impossible with the current architecture as the 4K page overlaps are too widespread across RAM for most devices and driver implementations.

Info 1.1.4

Thunderbolt is a proprietary external interconnect developed by Intel, becoming mandatory with USB 4.0, that routes direct PCIe lanes outside the computer and hence requires additional security measures. Such security measures include proper configuration of all MMUs a device has as well as performing checks that should make sure that only "trustworthy" devices acquire DMA access. Due to the ongoing demand in high speed peripherals and Thunderbolts high bandwith characteristics, Thunderbolt is seeing a widespread adoption on desktop computers, laptops, phones and other consumer devices.
Further information can be obtained from the corresponding specifications for USB 4.0 [USB21] which is the first public specification of thunderbolt as its previous versions were proprietary.

Thunderspy The thunderspy paper[Ruy20] is yet another example of Thunderbolt devices (for example PCIe/NVMe thumb drives) performing DMA attacks on victim computers. In this paper it is being made clear that existing IOMMU protection is flawed to say the least and that Thunderbolt firmware verification is really easy to bypass. They also show further flaws in the IOMMU design by Intel. In addition they published tools on their website that promise to alleviate the dangers of having non disabled Thunderbolt ports combined with flawed operating systems by correctly configuring the devices MMUs on boot.

NetTLP The NetTLP[KNMS20] project provides a way to send TLP (Transaction Layer Packet, Layer 3 of PCIe) packets directly over TCP/IP. Unfortunately the FPGA they have selected

[3]Even though modern MMUs are build for performance they still imply a performance penalty compared to just raw pointers. Manufactures go to great effort, to fix these performance issues while trying to maintain a reasonable level of security.

performs extensive data integrity checks in hardware prior to sending the TLP packets making sure one can only tamper with the TLP payload rendering their project a PCIe Layer 4 to Ethernet adapter. This was used to debug and test operating system driver implementations.

1.2 What is Fuzzing and why do we do it?

First of all we will look into what fuzzing is. We will see why we are doing it and what kind of fuzzers there are. Last but not least we will look into what we are dealing with when we are approaching PCIe fuzzing.

1.2.1 What is Fuzzing?

Info 1.2.5

A **bug** is an issue with a computer system (within its hardware or software) commonly caused by flawed design. Some of those bugs are of minor importance while others may severely impact the health of the system or lead to data corruption or leakage. Further information on the classification of certain bugs as well as a funny anecdote on where the therm bug originated can be reviewed in the work "Etymology of the computer bug". [Sha87]

Fuzzing is a technique where one inserts malformed data into a system and records abnormal behaviour. One then analyses what part of the data caused the issue. A data set entry is called a fuzz case and an issue that is triggered by one or multiple fuzz cases is called a bug (see info box 1.2.5 for further information).
As one is searching for bugs using flawed data one does not know exactly what such data looks like in advance. Due to this limitation one has to try numerous combinations of such data in order to find working (e.g. breaking) cases. Hence one measures the quality of a fuzzer by the effectiveness of the fuzz cases it produces, the variation of bugs it finds (different fuzz cases might trigger the same bug) but also the speed at which the fuzzer is able to test the system (measured in fuzz cases per second or FCPS).[LPJ^{+}18]

1.2.2 Why is this a great thing?

While there are other techniques for finding bugs in foreign designs, a fuzzer is a good approach as it is quite easy to begin with and usually results in quite deep analyses. Other popular methods are:

- **Static analysis** is generally speaking trying to find issues by looking at the implementation of a system. When done by human experts this technique proves to be rather effective but does not scale well. There are also pieces of software to statically analyse other software and while using such tools proves to scale well for larger software these tools tend to be quite error prone.

- **Dynamic analysis** is done by defining a set of forbidden states a system may never enter. When the system under test reaches one of these states a human has to research how the system entered the state and fix the issue. While this approach works well for systems where it is quite easy to observe the state the system is in, it is also quite labour intensive to just debug a single fault.

- While the two above-mentioned alternatives to fuzzing are used wildly, **Symbolic execution** is mainly of academic interest due to its tendency towards exponential complexity. In theory, one models the entire system as a set of symbolic equations and defines forbidden states. One then solves the equation system towards the forbidden states and obtains the bugs of said system. Unfortunately this has only ever been done for very small software programs as every new path within the system at least doubles the amount of possible states.

Due to the obvious disadvantages of the above-mentioned alternatives (most notably the common need for perfect observability of the system), fuzzing can be considered a good approach for testing hardware.[LZZ18]

1.2.3 What kind of fuzzers are there?

There are multiple approaches to fuzzing. All have in common that they try to produce test cases that discover as many distinguishable bugs as possible, but differ on how they try to reach this goal. The most important distinction lies in the dependence on having access to the inner workings of the design under test (later called DUT).
If the fuzzer has complete knowledge of the inner workings and states of a system one calls the fuzzer a **white-box fuzzer**. If the fuzzer cannot access the source code (in our case, the hardware schematics in some form) of a system but is capable of tracing the state of execution (software: for example by performing taint analysis[GS17], in our case for example by probing state machine counters) one calls such a fuzzer a **grey-box fuzzer**. If a particular fuzzer does not have access to both information, it is known as a **black box fuzzer**.[MHH+19]
One classifies fuzzers also by their usage (and acquisition) of feedback from the system under test. A fuzzer which is not using feedback is considered to be a dumb fuzzer. One that is using feedback such as areal coverage of the fuzz cases is considered to be a smart fuzzer. Usually smart fuzzers are way slower but produce slightly better fuzz cases and thus are considered to be more efficient.[VKC17][LPJ+18]
Furthermore one can classify fuzzers based on the way they generate their fuzz cases. A **mutation based fuzzer** does not know how the input needs to be formed in order to achieve good coverage.

Instead it "mutates" a given set of input data and inserts the result. A **generation based fuzzer** on the other hand knows how input needs to be crafted in order to reach deep paths within the system under test and generates fuzz cases using that knowledge. A generation based fuzzer often uses an input grammar to craft the input data.
Last but not least one can further classify fuzzers having knowledge about their DUT (white- and grey box fuzzers) based on their approach of coverage. A **directed fuzzer** tries to reach a specified target that might lay deep inside the system. A **coverage-based fuzzer** tries to maximize its total coverage of the system.

1.2.4 The caveats of fuzzing hardware instead of software

According to multiple papers dealing with fuzzing[LZZ18][LPJ+18] [GWY+15][MHH+19] it is always a nice thing to know how input affects the system one wants to test. Unfortunately such information is hard to obtain when dealing with hardware. The following possibilities exist (in ascending order of difficulty):

1. Using the response channels of hardware: more complex circuits tend to have signals that tell the user if a certain action succeeded. One can monitor these channels for unexpected behaviour.

2. Measuring the response time of a circuit. If a circuit does a particular action one expects the circuit to perform it in more or less constant time. If there is a large deviation of time one can assume that the circuit either reached a bad state and had to reset itself or did something that it was not supposed to do. Either way it notifies a human to take a closer look on what was going on.

3. Measuring side-effects. When we are talking with a NIC we do not expect other devices on the same BUS to answer or even perform undesired action. The same goes for other chips on the PCIe network.

4. Measuring side-channels for coverage: Electronic circuits consume power when operating and while doing so they emit heat and a magnetic field. One can measure these three parameters and deduce the area of effect (coverage) on the chip.

1.2.5 Other (more classical) approaches to testing hardware

Fuzzing hardware is a quite new approach. The default methods of testing hardware usually involve elaborated testing of smaller modules using simulation and formal verification both manually and by utilizing automated tools. Integration tests are usually limited to answering the question "Does it work?" by either manual testing or running suits of predefined tests automatically. The most

common way of doing so is writing an input file (or a firmware) that is supposed to trigger every hardware feature at least once and if the end of said file is reached without any detected errors the hardware is considered to be working.
The upside of doing so is that it is fast. Having a short time to market[4] is crucial for companies delivering hardware. The downside is that it is quite common that one encounters major issues with hardware that is already on the market. Perhaps the most common examples of such issues are the pentium bugs[Pra95] and the recent plenitude of side channel attacks and exploited speculative execution bugs.

1.3 What is an FPGA

In order to fulfil the requirements of fuzzing hardware we need a device that is capable of injecting arbitrary data on the PCIe network.

1.3.1 Why can't we simply use some microcontroller and call it a day?

Microcontrollers are little processors that you can program using your favourite programming language. They are easy to use, real flexible and cheap. They usually are the first choice for building things that interact with hardware. Unfortunately we can not use one for our purposes.
So we want to fuzz some hardware (PCIe components in our case). This means that we can not use microcontrollers because we need to have access to a layer that is as low as possible in the protocol stack. Although there are some microcontrollers that are actually capable of communicating using the PCIe protocol, they enforce a great deal of error detection and correction and offer a very generalized interface for programming. Usually this is a good thing but not for fuzzing since that involves sending corrupted data.
Fortunately some FPGAs allow us to do so.

1.3.2 What exactly is an FPGA then?

The acronym FPGA is short for Field Programmable Gate Array. The name says it all. It is an array of gates that do not form a hardware circuit on their own. The logic gates need to be connected first in order to obtain something useful and this can be done "in the field" (outside the chip factory). The process of connecting those logic gates (or in reality look up tables, but more on this later) is called configuring or programming.
The benefits of having an FPGA instead of having a fixed chip or classic Gate Array or Complex Programmable Logic Device (CPLD) are numerous. Here are the points that are relevant to us:

[4]Time required to finish a product and starting to sell it

- We can actually afford one. It is way cheaper to buy an FPGA kit than getting someone to produce your own chip design.
- Contrary to CPLDs one can configure an FPGA as often as one wants to as long as your design does not damage the hardware. One does not have to buy a new one if the design is not working or needs to be changed for some reason.

1.3.3 The gotchas of working with a big FPGA

Besides consisting of countless logic building[5] blocks and their interconnects (which combined is called the fabric) modern FPGAs also feature so called hard IP (where IP generally means Intellectual Property which is some hardware that you did not design by yourself but use in your design).
Usually such hardware is necessary since the inner workings of a particular FPGA remain a well kept industry secret[6] and one cannot implement own hardware in a serious manner without them. Popular examples for such elementary building block IP are PLLs for dealing with clock boundaries (see info box 1.3.6), BRAMs for having on chip storage or DSPs for performing complex arithmetic operations such as (floating point-) multiplications or FFTs in a space and time efficient manner.

Info 1.3.6

Clock Boundaries are areas in your hardware where circuit A and circuit B need to exchange data with each other but both circuits use different clocks and thus are not synchronized. These clocks may run on different frequencies and may have a non-zero phase shift, may have different duty cycles or different setup and hold timings. Based on the knowledge of the nature of the data to be transmitted and knowledge of the clocks in use, one needs to design special behaviour on these boundaries in order for the data transfer to not loose or damage any data. Such transmission hardware can be arbitrarily complex.

The downside of such plenitude of functionality is complexity. Complexity of hardware is the root cause of long synthesis times (even our little fuzzer design takes well over an hour to synthesize even on a very powerful computer[7]) and a lot of issues to think about. This makes it practically impossible to use the popular programmers approach of rapid prototyping so one needs to adopt a new work algorithm:

1. **Think** how you like your hardware to function

[5]The most common logic devices are: Lookup Tables (LUT), Arithmetic Logic Units (ALU) and Muxes
[6]There are significant attempts from the open source community in providing FOSS toolchains for dealing with FPGA. [SHW$^+$19] While writing those tools, they also document the inner workings of popular models.
[7]Recommended system specs for building the project are more than 32 GB of memory, at least 500 GB of free disk space and as many fast CPU cores you can get.

2. **Design** your hardware module

3. **Simulate** your module and (if possible) use a formal verifier to proof that your hardware works (at least in theory)

4. **Fix** your hardware if it does not work in simulation (most of the time one needs to deal with bad timings)

5. **Pray** that your module will work in its final place.

Unfortunately a simulation is only as good as the model it is based on and if one forgot to add certain constraints it is still possible that hardware works perfectly fine in simulation but fails in reality. Part of this is due to the fact that while working with small FPGA one does not have to think about clock and reset trees. When working with big ones one has to make sure the signal arrives at the right time and think about clock distribution.

1.4 What others already have done

Of course we are not the first ones looking into this matter. Let us gain some insight into other contributions in this field.

1.4.1 Google PCIe Fuzzing

In 2017 researchers at Google already had a look at IOMMU security[Han17]. They built a system that searched the entire address space of a server to check if their MMUs within their cloud services were correctly configured. Unfortunately there are no further information on this matter. All we could find was a press notice and no paper or extended results.

1.4.2 The Pulsar paper

The Pulsar paper[GWY+15] references a collection of smart algorithms for black box fuzzing of proprietary network protocols. They used records of unknown clients and servers in order to create fake clients and fuzz the server without knowing implementation details of either of them. This revealed certain starting points for designing our payload generator.
Unfortunately hardware usually does not return such comprehensive error messages and thus we can not use their techniques directly but they are an inspiration for sure.

1.4.3 The Agamotto fuzzer and related work

Agamotto[SHK+20] is a VM-Based PCIe fuzzer. It fuzzes TLP payloads from a VM in order to test devices the VM has access to with the intention to harden kernel based virtualization. It is also

used the other way around to test device drivers within the kernel by emulating these devices for the VM under test. Their fuzzer is based on AFL[8]. Unfortunately it is not possible to generate arbitrary TLP messages or send messages to a real device. Due to this limitation we consider this to be a layer 4 fuzzer.
Based on the Agamotto fuzzer or using a similar approach there are some other projects dedicated to hardening kernel drivers and hypervisors by fuzzing their input buffers using various payloads. The only one of those we found even considering PCIe to be an issue is the hyper cube paper [SAA$^+$20]. Some smaller projects within the Linux community did not produce papers though.

1.4.4 PCIe Man-In-The-Middle

The "Toward a Hardware Man-in-the-Middle Attack on PCIe Bus" paper[KLR$^+$20] describes the approach of using an FPGA to log data transactions occurring on a PCIe link. They want to use this technology to obtain information required to unlock a smartphone in the setting of forensic analyses. They demonstrated this capability on a simulation platform and would like to enhance their setup to perform replay attacks in order to trick a smartphone into allowing endless pin entries without locking the storage. They are currently using an Intel Stratix 10 FPGA to read the data but want to switch to an Intel Arria 10 in the future due to its advanced PCIe capabilities.

1.5 What is the goal of this work?

Just prior to getting into on how to build a fuzzer we may recapitulate on what we are trying to achieve. PCIe is one of – if not the – most important high speed interconnect in todays computers, ranging from smartphones over laptops to servers and supercomputers. Devices connected over PCIe are usually granted read and write access to various memory locations of other connected devices. Furthermore this interconnect provides numerous other features that are fairly new and thus untested as well as ones that are mostly forgotten while still being present and providing powerful access to hardware functionalities. While there are significant barriers towards efficiently testing PCIe it is still surprising how little research there seams to be.
Having a tool capable of efficiently fuzzing devices connected to this network could help closing the gap between the lack of research and the potential security implications of using PCIe.
We would like to engineer an FPGA design capable of sending raw unfiltered Transaction Layer Packets. While doing so we solve the issues of using the given hardware outside of its intended purpose. Furthermore we discuss a way to judge the outcome of fuzz cases without being capable of directly observing the state of the target hardware. Last but not least we demonstrate the workings of the fuzzer.

[8] `https://github.com/google/AFL`

2 Building the Fuzzer

Let us start with the actual development of the fuzzer.

2.1 Preparations

Prior to getting started we need to decide on what we are targeting and how we would like to do so.

2.1.1 Reading through the PCIe specs and select desired targets

First of all we read the PCIe specifications for revision two in order to get an idea of what needs to be done and what are worthy targets for a fuzzer. The reason that we started with revision two was due to it being easier to obtain. Later we also took a look at revision 4.

We quickly realized that Layer one and two of the PCIe protocol are not interesting in terms of fuzzing them. Layer one, while being complicated in terms of physical properties does not come with anything to fuzz at all and layer two only features a state machine which either works or does not. Those two layers do not seem to be that complicated at all and manufacturers probably get them right. Layer 4 of the PCIe protocol, while also being really interesting in terms of attack surface, is out of scope of this work due to the fact that testing this layer from software is supposed to be quite simple. All one has to do is to write a driver that sends the data one likes and record the behaviour[9].

Layer three turned out to be a completely different story. It is filled with all kinds of configuration registers, especially with revision 4, that are hard to reach, if not impossible, from an operating systems perspective. These registers also represent a deep intervention into the inner workings of a device. If for example one could trick the configuration of SR-IOV into switching virtualization domains an attacker VM would have access to all the data a victim VM loaded on a rented accelerator in a cloud environment.

2.1.2 Reading through manufacturers documentation to find suitable hardware

So we found our target layer which proved to contain a lot of interesting targets that are complicated to implement and hard to test. The only downside is that they are actually hard to reach. One needs to send specially crafted TLP to reach them and user applications (both hardware and software) are

[9]As it turns out it is still complicated enough to write a separate paper about how you did it in addition to the paper about the results you got.

not supposed to directly invoke these features. Users should only configure their own hardware the way these features are supposed to function and let the hardware deal with the rest.

As a consequence we are looking for FPGAs capable of transmitting raw and unmodified Transaction Layer Packets from user configuration space without filtering those out that are considered to be malformed. Unfortunately this does not seem to be a common use case. Some server grade FPGA support our use case, although not directly.

So we decided to acquire an Intel Agilex F-Series dev kit for PCIe [10] and contacted a merchant. This FPGA is much larger in terms of fabric size than we would ever need but it features a P-Tile capable of performing the plain packet bypass we need.

About one month later we have received the FPGA and began building a test bench for working with it.

2.1.3 Designing an architecture

When beginning a bigger project it may be wise to first think about what one wants to accomplish. A fuzzer needs a way to send its test cases to the DUT. It also needs some form of control system that is capable of deciding what to send next and one that is capable of processing the results. Furthermore, we need modules that are capable of maintaining a steady connection to the connected PCIe devices.

Our goal is to use ready made IP where possible and to engineer the remaining modules ourselves.

The Diagram 2.1 gives an overview of our SoC design. Of course the real architecture contains further components, such as necessary memories, protocol converters, timers, PLLs, etc. that are not drawn here for the sake of simplicity.

On the right side we have drawn the components essential for maintaining a PCIe connection as well as actually sending data through the PCIe interface. Besides a Phy for that interface, which is the P-Tile, we also need modules for managing the connection state as well as modules designed to react to power management hints in time. In the centre of Figure 2.1 we see a big module named 'Fuzzing Controller'. Its purpose is to instruct the package generation module as well as having an overall eye on the process of fuzzing. The development of this module is described in Section 2.6.

Having a piece of hardware that can fuzz the PCIe network is no good without actually being able to extract the results from it as well as controlling it. This is where the left side of the above mentioned diagram comes into play. Hence we need some kind of software driven processor that we can use to do so.

[10] `https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-agf-fpga.html`

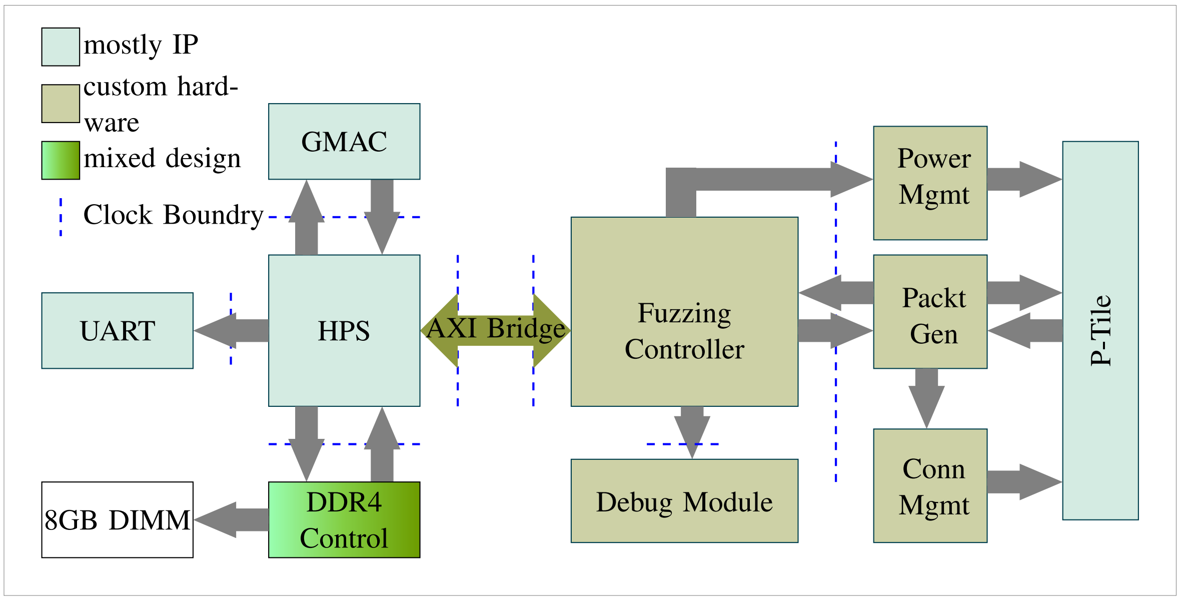


Figure 2.1: Simplified Architecture of our Fuzzer

Info 2.1.1

A **Finite State Machine** (or short FSM) is a piece of hardware that resides within one of a collection of predefined states. On every rising edge clock cycle the FSM may perform a set of predefined actions, like outputting or transmitting something. On every falling edge clock cycle it transitions to a new state while the new state might be the old one.
A FSM closely matches the concept of a DFA with output as known from theoretical computer science. The reason a lot of hardware is implemented in some form of FSM is due to its simplicity and robustness.

As a consequence, we decided to utilize the HPS as our control instance for setting fuzzing patterns, starting, pausing, resetting the Fuzzer as well as conditioning the results. Our development kit features a gigabit Ethernet port that one can connect to the SoC we designed. This enables us to use features such as SSH to control our experiments and transmit data. For convenience and debugging matters we also decided to add a simple UART connection as even the simplest bootloader prompt is capable of dealing with it. The entire development process of the HPS is described in Section 2.3.
Furthermore we decided to implement the packet generation and connection state management as Finite State Machines (see info box 2.1.1) in hardware. Self-evidently we implemented all other required pieces for communicating over PCIe in hardware as well. Intel provides us with an IP block for decoding layer 2 of PCIe called the P-Tile though. We can interface with it in order to

Table 2.1: Hardware required to perform operations

Purpose	OS	CPU	RAM equipped (used)	Required Storage
Synthesis	Linux	16 Threads @ 4.5GHz	64GB (about 40GB)	256GB
Programming	Windows	6 Threads @ 4.1GHz	8GB (> 8GB)	200GB
HPS Image	Linux	32 Threads @ 3.5GHz	128GB (about 20GB)	up to 400GB

send the TLPs we need. This is described in detail in section 2.5.

Last but not least we still have two separate chunks of hardware that we still need to interconnect. This is where the bold arrow in the middle of the block diagram comes into play. As a matter of fact it is a quite complex module itself, connecting the AXI interface of the HPS to the fuzzing control FSM. Its development is described in Section 2.4.6.

2.2 Beginning the Odyssey

Designing an entire SoC from the ground up is quite an endeavour. At first one has to lay down the foundations though.

2.2.1 Setup the build environment

Last but not least we installed Intel Quartus (a piece of software that allows one to synthesize hardware designs and program the FPGA) which turned out to be surprisingly complicated due to installers that refused to install device drivers on Microsoft Windows (due to long expired certificates) and requiring long outdated operating systems when working on Linux.

Generally Quartus works way better when running under Linux as there seem to be quite a few hard coded paths within software that indicate it was written with a UNIX-like operating system in mind. At least the Linux version does not complain about not finding the *`/tmp`* directory.

So after having downloaded multiple hundred GB of software, edited countless environment variables and a lot of manual OS patching we are ready to go. One also needs to install the development tools for the HPS but despite the fact that these tools are distributed all over the place they were surprisingly easy to install as they did not require fixing first.

Finally we ended up with a test bench computer running Windows for programming matters (the JTAG driver simply refused to be loaded with any recent Linux kernel) and two powerful Linux workstations for designing the hardware, synthesis and compiling a firmware for the HPS. Hardware combinations that worked for us are described in Table 2.1.

2.2.2 Configuring a first Hello World bitstream

Whenever one get new hardware one would like to perform initial tests in order to figure things out. In particular we tried to achieve the following two things:

- Get some debug lights to blink – This is a common "Hello World" example when working with FPGAs as it shows one how the clocks work and how one configures the FPGA.
- Let the FPGA show up as an unknown device within the host operating system – This shows us that the PCIe layer one decoder (also known as Phy) is working.

While doing so we learned a couple of lessons. First of all we need to tell Quartus to preserve unused transceiver channels or they will degrade when being unused within an active design (see info box 2.2.2). So we had to enable *`PRESERVE_UNUSED_XCVR_CHANNEL`* within our qsf file[11] and route an active clock to the unused transceiver channels (in this case the E-Tile channels to stimulate them). The second thing we observed was the fact that we were not able to configure the FPGA. It took us (and the Intel support we contacted) quite a while to figure out that the engineering sample we have got did not come with a pre-burned SDM image and we had to flash one ourself. Furthermore we had some DIP switch configured incorrectly which was quickly fixed.
After having sorted out these issues it was just a matter of debugging the P-Tile settings (as it turns out it is quite easy to provoke a Windows blue screen when one does now yet know how to properly utilize PCIe enumeration), programming the FPGA using JTAG and watching LEDs blink.

Info 2.2.2

High speed tranceivers degrade over time if they encounter a static signal due to burn out of silicon based power amplifiers. Sending a high frequency signal over a relatively long copper wire requires a large amount of energy[a]. In order to achieve such high insertion power rates one has to use said amplifiers and if they encounter a static *`HIGH`* they may degrade resulting in poorer signal quality which in turn results in more packet errors resulting in more packets that must be resent which results in lower total bandwidth. As most high speed signals are implemented using differential signalling one has to toggle these amplifiers.

[a]This is the main reason why tera bit speed network switches may require multiple kilo watts of energy for even modest port counts.

2.2.3 Writing the structure of the actual circuit

When dealing with projects in Quartus one should be aware of four different concepts. First of all (like many other EDA suites) Quartus expects one to maintain a top level module. This module must not be created using QSYS (see info box 2.3.3). Instead one needs to write this module using

[11]A .qsf file is the file where Quartus stores environment variables (a. k. a. project settings) for the synthesis.

a text based hardware description language like System Verilog or VHDL which may lead to quite large files to maintain (especially if one has to feature a lot of LVDS signals like PCIe is. This module instantiates all submodules and its outside connections are mapped to the FPGA pins.

Simply having connections within the top module does not really connect them. How should the software know which human readable name should correspond to which pin of the chip? This is where the second concept about the constraints file comes into place. The current active constraints file (`*.qsf` file defined within the `.qpf`, Quartus project file) tells Quartus, among many other things, which pin correspons to which port of the top level module. We also need to define the voltage levels, LVDS pairs, maximum current loads, signal characteristics etc. in this file. It is of utmost importance that one uses the correct values within this file because in the best case the design does not work and one is wasting time looking for the error but in the worst case one is producing magic blue smoke.

At next one needs to setup the timing constraints file (`<project name>.sdc`). This file tells Quartus which signals toggle at which rate using what duty cycle. It is not necessary to tell Quartus about every signal (remember, there are thousands of them) as it can deduce them by following the clock tree but one needs to specify each input clock as well as how the clock tree should be used. One also may define false paths within this file (see info box 2.5.11).

The last thing one should know is that one has to use the Reset-Release IP when dealing with Agilex FPGAs. This IP only has one single output and one should not release the reset of our circuit as long as this signal is `HIGH` as it means that the configuration of the FPGA is still not done.

After having dealt with all of those issues it is a simple matter of implementing everything.

2.3 Getting the HPS to work

We need some sort of control instance for the fuzzer in order to process the data. Similar to many other FPGAs our Agilex chip features a hard processor we can use. It is called Hard Processor System (HPS). Getting it to work is our next objective.

2.3.1 Designing the hardware we need

The HPS itself only features four 64 bit Arm cores and some very basic peripherals. In order to form a working SoC we need to surround it with further required peripherals, design interrupt channels and build a physical memory layout. Fortunately we can use QSYS for this matter.

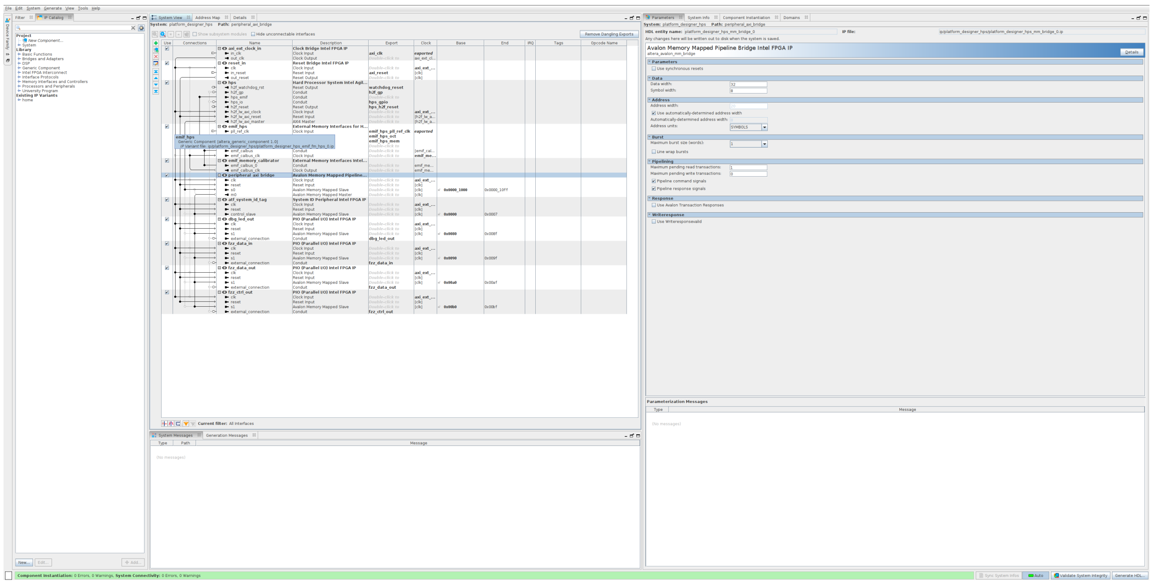


Figure 2.2: A screenshot of the QSYS setup of the HPS SoC

Info 2.3.3

QSYS is a part of Quartus allowing us to compose systems using a graphical interface. Instead of having to manually connect every signal of an interface like we had to do for the PCIe interface in our top level module we can simply connect parts together using connection lanes on the left side of Figure 2.2. In the middle we see important system parameters like interrupt routing, address mappings, clock- and reset trees as well as custom instructions.

Listing 2.1: Quartus latency error message

```
Error(11928): 'enet_intn~pad' with I/O standard 1.8 V, was constrained to be
    within bank 'HPS'
Info(11929): '1.8V' is a valid VCCIO value
```

The main thing we have to do is provide clock bridges and instantiate the HPS. A HPS is hard IP so one can only instantiate as many as there are instances of it on the FPGA. In this case exactly one. Regarding parameters this block is actually quite simple. All there is, are the bridge settings for various interconnects, reset settings and peripheral routing options. While the former two do not require much explanation the last one does.
When dealing with the HPS we want to connect various peripherals to it (in our case a mico SD card for storage (MMC), a gigabit Ethernet interface, UART, and an I²C controller for management).

When doing so one needs to configure the peripheral routing of the HPS to perform the correct routing in addition to setting up the pin constraints. Furthermore, and this is really crucial, leaving the input latency configurations on `auto` for certain peripherals within the IP parameter editor will cause the fitting step to fail – with a very meaningless error message, see Listing 2.1. It took quite some time to figure out that this was the cause.
As a consequence we need to research the data sheet of every peripheral controller we like to use and calculate, using the schematics, the input chain delay for those components.
When having configured the external peripherals it is about time to configure the internal peripherals of our SoC. By now all we have are four ARM cores, with some controllers connected to the AMBA BUS (see info box 2.3.4). Next we need to setup the timers and clocks that are required for the HPS to function. Furthermore we need to route the interrupt signals using QSYS. They need to be routed to both the HPS and to an interrupt latency controller. Connecting an interrupt lane to at least one watchdog instance is not a bad idea either. We also still need to connect our own hardware modules to it and do so by enabling the HPS to FPGA AXI Lite bridge and performing the address and clock translation. In order to finish what we were doing we need to export the remaining AXI bus to our top level module and instantiate our bridge to our fuzzing controller (more on that in subsection 2.4.6).

Info 2.3.4

AMBA, which is short for Advanced Microcontroller Bus Architecture, is a platform specification for integrating ARM based SoC. Its main purpose is to interconnect ARM processing cores with each other and their MMUs. It also defines AXI as an memory mapping edge connector to external network-on-chip devices.

2.3.2 The art of having more than 256KB memory

Up until now, we have underestimated the fact that we do not have memory for our processing cores yet. The Agilex HPS has 256KB of on chip memory. That is approximately enough storage to run a bootloader and some bare metal applications. We like the convenience of an operating system though. While we could technically run a FreeRTOS within this space we would like to run Linux and thus need to utilize one of the four DDR4 DIMM slots on our dev kit.
When settings up the HPS one may notice that the cache coherence unit is connected to multiple muxes. They switch the onboard memory from being the primary memory source to being utilized for different purposes. As the SDM bootstraps the HPS using this memory as the primary memory and copies the FSBL to it we need to make sure that our FSBL payload switches the used memory properly. More on this in subsection 2.3.3.

After bootstrapping we enable the 32KB instruction and data caches each ARM core has as their L1 caches, use the 1MB of cache connected directly to the four ARM cores as their shared L2 cache and use the 256KB of on board memory as our IO cache. Last but not least we configure our main memory be provided by our DDR memory controller.
While almost all of the above is a matter of correctly configuring the CCU and the MMU, the usage of our primary memory involves further work. The HPS MMU is connected to a specific region of the FPGA containing only basic lookup tables and a lot of configurable muxes as well as outside connections to the DIMM slots. We have to use this section in order to perform the following tasks[12]:

1. Provide a DDR4 controller that controls the memory stick and serves it to the HPS.
2. Provide a DDR4 initialization module that initializes the memory, queries its timing and configures the controller with the values it obtained.

While one can utilize Intellectual Property provided by Intel one would be overwhelmed by the number of highly technical parameters. We suggest reading an introduction[13] to DDR4 on the internet in order to better understand the meanings of these parameters. After having done so we used Intel IP modules where appropriate but were forced to write the memory hand-off ourself as this part is application and design dependant.

2.3.3 How to build an embedded operating system

Once we have our SoC we need software to run on it. Let us start with the FSBL as it is the first piece of software that is running on the HPS after the SDM copied it to the onboard memory and set the PC to the start address of it.

The Bootloader One has to build it using the μBoot[14] source code for Agilex devices. Besides that, one has to compile the Arm Trusted Firmware (see info box 2.3.5) for Agilex and assemble the memory setup code generated by QSYS (which is just a collection of moving magic numbers into certain memory mapped registers causing the CCU and MMU of the SoC to switch to external memory) when generating the HPS system. Finally one has to link[15] them all together and keep the resulting binary file. We need this `.hex` to supplement the bitstream with it as the programmer will pass it to the SDM during configuration of the FPGA (as we already discussed).

[12]One may also implement other kind of memory here, for example sharing the memory with the application implemented on the FPGA or using the FPGA BRAM as HPS memory

[13]For example `https://www.systemverilog.io/ddr4-basics` as a starting point and `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/agilex/mnl-1100.pdf` to find suitable data sheets for parameter lookup.

[14]`https://github.com/altera-opensource/u-boot-socfpga`

[15]In computer science the process of merging multiple fragments of software in binary form into a complete software is referred to as linking.

Prior to compiling the ATF we need to either generate a signature of our second stage bootloader and pass it to the ATF configuration or disable the signature validation of the ATF. As there is no documentation on how to disable the signature validation of the second stage boot loader (SSBL, we always got a crashing firmware while trying to do so) we simply pass the signature to it. Conveniently the default configuration of `make socfpga_agilex_atf_defconfig` generates a suitable PEM file[16] for us yet oddly enough does not automatically build an ATF compatible SSBL. At this point one starts to notice that automating the build script would be a good idea. Have a look at Listing 4.1 in the appendix to review ours.

Info 2.3.5

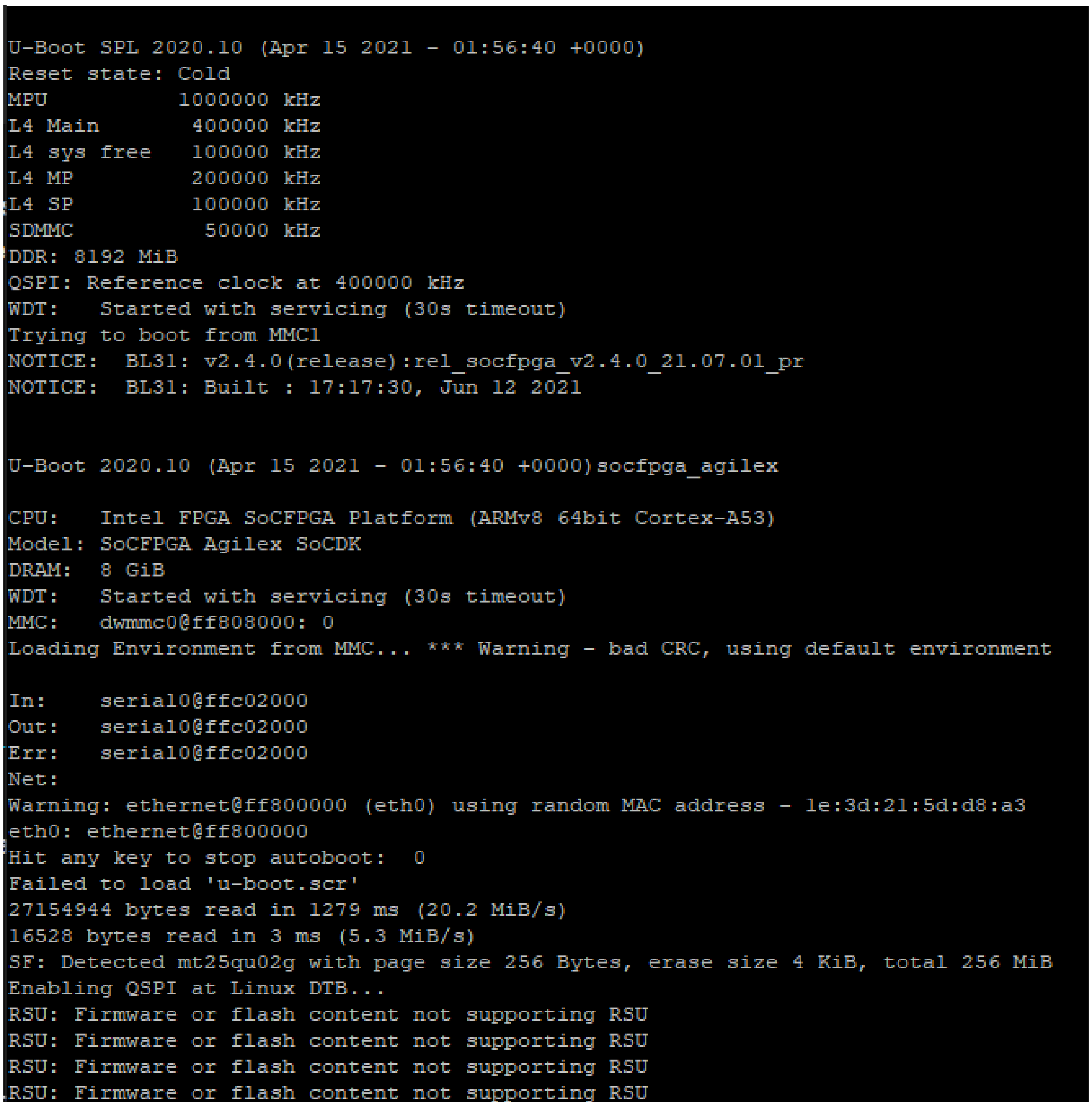
The **ATF** is a piece of software that is responsible of setting up Arm Trustzone (a special execution mode that promises advanced execution security for accessing data) and validating software signatures of software to be run (in this case the second stage bootloader SSBL μBoot). Besides that, the ATF version for Agilex devices communicates with the SDM which seams to be mandatory.

At the end of the script we see a command to build an SD card image. After having built our entire bootloader we can use that command (without specifying a Linux Image, device tree binary (DTB) and userland (have a look at info box 2.3.6) – we have none of those yet) to try it out. We may now generate our bitstream as well as our image and burn it to an empty SD card.

Now it is time for a first test. After having debugged any issues with the hardware side, one is greeted by UBoot with the initialization logs presented in Figure 2.3.

After having switched to the new memory mode the SSBLs signature is being validated and if it passes it will be executed. Once the bootloader finished initializing the rest of the attached peripherals, it greets us with a warning message that no `u-boot.scr` file was found (we expected that as we did not specify one yet) and begins a countdown. After firmly pressing backspace we can verify that UART works (we are getting a console after all), Ethernet works (we can ping things and got an IP address from the DHCP server) and mass storage works (we can review the content of our (yet pretty empty) SD card). A complete list of all commands our UBoot image features can be reviewed in Table 1 in the appendix.

[16] A PEM file — which stands for Privacy Enhanced Mail due to historical reasons — is a file containing a cryptographic certificate. In this case it is responsible for signing code.



```
U-Boot SPL 2020.10 (Apr 15 2021 - 01:56:40 +0000)
Reset state: Cold
MPU          1000000 kHz
L4 Main       400000 kHz
L4 sys free   100000 kHz
L4 MP         200000 kHz
L4 SP         100000 kHz
SDMMC          50000 kHz
DDR: 8192 MiB
QSPI: Reference clock at 400000 kHz
WDT:   Started with servicing (30s timeout)
Trying to boot from MMC1
NOTICE:  BL31: v2.4.0(release):rel_socfpga_v2.4.0_21.07.01_pr
NOTICE:  BL31: Built : 17:17:30, Jun 12 2021


U-Boot 2020.10 (Apr 15 2021 - 01:56:40 +0000)socfpga_agilex

CPU:   Intel FPGA SoCFPGA Platform (ARMv8 64bit Cortex-A53)
Model: SoCFPGA Agilex SoCDK
DRAM:  8 GiB
WDT:   Started with servicing (30s timeout)
MMC:   dwmmc0@ff808000: 0
Loading Environment from MMC... *** Warning - bad CRC, using default environment

In:    serial0@ffc02000
Out:   serial0@ffc02000
Err:   serial0@ffc02000
Net:
Warning: ethernet@ff800000 (eth0) using random MAC address - le:3d:21:5d:d8:a3
eth0: ethernet@ff800000
Hit any key to stop autoboot:  0
Failed to load 'u-boot.scr'
27154944 bytes read in 1279 ms (20.2 MiB/s)
16528 bytes read in 3 ms (5.3 MiB/s)
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total 256 MiB
Enabling QSPI at Linux DTB...
RSU: Firmware or flash content not supporting RSU
RSU: Firmware or flash content not supporting RSU
RSU: Firmware or flash content not supporting RSU
RSU: Firmware or flash content not supporting RSU
```

Figure 2.3: A screenshot from the UART output of the HPS SoC right after reset

Info 2.3.6

The **userland** is the part of an operating system that provides the default executable programs (for example text editing, settings management, a shell, etc.) as well as the default file system layout to the user. Some people also refer all non-kernel processes as part of the userland even though they are only existing in memory as opposed to programs on hard storage. In addition to the userland an operating system also provides a kernel.

Device Trees Now prior to explaining how to compile Linux and building a user space we shall have a look at device trees. When dealing with all the memory mapped I/O (see info box 1.1.1) the question arises how does the kernel know where hardware is located and what it is? The short answer: It does not as long as we do not tell it.

While certain hardware structures are always available on certain architectures and provide a bus enumeration capability (for example the internal UART interface every Windows compatible computer still has or PCIe) the majority of hardware is not. At the beginning of embedded devices there was a special Linux port for every device there was. These sub-architectures contained information on how the hardware needs to be used. Since the explosion of availability of such devices this approach does not scale any more hence the device tree concept was introduced.[GH06]

Instead of having a special Linux kernel for every device[17] we now have a special kernel suitable for every mayor CPU architecture and specify the layout of our device within a device tree, even including information such as resets, clocks and memory layout. The bootloader loads the compiled version of the device tree from a `.dtb` file into memory and passes the DTB address in a register to the application that is supposed to be booted (usually Linux). Since the advent of embedded devices capable of supporting dynamic memories (like out HPS) the bootloader also patches the device tree in memory, to fill in a list of all available memory and to pass the kernel boot parameters, prior to jumping to the start address of the kernel. A compatible kernel then boots in the most basic operation mode the hardware supports[18] and parses the device tree in order to initialize the hardware and retrieve the boot parameters.

With the advent of SoC we usually have architectures around the CPU that have a lot in common. Due to this SoC manufacturers usually provide a template for their hardware that includes all the devices the SoC features having only mandatory devices (like for example memory and clock controllers) enabled. Such templates are packed within `*.dtsi` files where the i stands for include. One includes the correct file then and enables the hardware one wishes to use inside the platforms `.dts`[19] file. Of course one may include several files (for example one for the SoC one uses and one for the peripherals one connected to the SoC).

As we are building our own SoC we have to provide such a file for ourselves as well (see Listing 4.2). Furthermore we wrote several device tree definitions (for debugging reasons) of which Listing 4.3 provides the final one.

Building a Linux Distro But first we need to compile a kernel. All we have to do is patch certain parameters, calling the configuration system (`make socfpga_defconfig`) and building the

[17]Special device drivers may still be included as one only needs to ship the drivers one really needs but they are maintained separately.

[18]Usually the bootloader already initializes certain hardware itself. UBoot nowadays also uses the very same DTB file to do so.

[19]dts stands for *device tree source*

kernel (`make Image dtbs`)[20]. It helps to enable a parallel build by specifying `-j <number of cores>` as building the kernel would take ages using just a single thread. After everything compiled we have a file called `Image`. Congratulations, we have got ourself a kernel.
Last but not least we have to build our userland. We use a software called bitbake[21] to not have to build one completely by hand. Furthermore we need meta layers from the yocto project[22] as well as layers provided by Intel covering the HPS[23] specific part as well as our own building our software. As each meta layer is supposed to be only specifying how a particular layer of the OS is supposed to look like[24] all we need to do in theory is write our own meta layer for our specific drivers and software and call it a day. We choose the following layers for our image:

- `meta-core` — A dependency for other layers
- `meta-mentor-common` — Another dependency
- `meta-intel-fpga` — Hardware support for our SoC
- `meta-oe` — The basic userland
- `meta-mel` — Debugging and development Tools
- `meta-networking` — A network stack, including TLS libraries
- `meta-filesystems` — Extended file system support
- `meta-python` — Our favourite programming language
- `meta-fuzzer` — Our own kernel modules and software

In reality the layers provided by Intel are deeply broken and need to be heavily patched prior to being usable (at least for our purpose). As broken embedded designs seem to be normal the yocto project provides a tool (have a look at bitbakes `recipetool`) to fix these layers without touching them too much and we do not need to use `sed`[25] that much and after patching and configuring the layers we can finally press the start button and grab a cup of tea.
It is a good investment of time to optimize the setup though as it is quite common that a yocto build process takes up to a week. After optimizing it, it is possible that it will only take a couple of hours on decent hardware. We should also keep in mind that it is wise to include developer tools with our

[20]dtbs stands for *device tree binaries*, this target compiles all applicable .dts files to the corresponding .dtb one.
[21]`https://github.com/openembedded/bitbake`
[22]`https://www.yoctoproject.org/`
[23]`https://git.yoctoproject.org/git/meta-intel-fpga`
[24]This way one can for example exchange the machine dependent part when switching to a new hardware version and can keep using the upper layers of the firmware
[25]sed is a streamlining editor found on most Linux or UNIX based operating Systems

distro as modifying software in place is much faster than having to rebuild the entire image every time one wants to change something.
The complete build script can be reviewed in Listing 4.1 in the appendix.

2.3.4 Digression: Hacking the Linux Kernel

Info 2.3.7

The **differences between kernel and user space** lay in the way security measures are being enforced to the running code.
While being in user space a process cannot access resources that it does not own and is guaranteed that all resources that its own are not altered from the outside (except when the process explicitly allows it). Memory protection and exceptions are enabled and guard your code. Furthermore code running in user space does not have to perform cooperative multitasking. When the process is taking too long all of its memory (registers, PC, heap and stack) are being saved, the process is being rescheduled and its memory restored later when the process is running again.
The complete opposite applies to code running in kernel space. While this is very inconvenient it is still mandatory as the kernel code needs to interact with the hardware on that level.

We need to understand a hand full of things prior to reading the next chapters as kernel code requires special treatment and thus certain things might seem strange.
The process of writing kernel code is actually quite well documented within the Linux documentation[26]. First of all we need to acknowledge the fact that we are not the only code running on the CPU. Other code wants to run too. While the Linux kernel provides a scheduler for kernel code it is not available in every scenario and we need to make sure that our code is not blocking when it is not available — otherwise we would freeze the entire computer. It is also best practice to leave such an area as fast as possible[27] due to obvious reasons and not call functions that may block carelessly.
Second although most of kernel code we are going to see is run in "user context" this has nothing to do with user space (see info box 2.3.7) so our code needs to align to that. In particular it is wise to not perform any long computations within kernel space since we have to schedule manually and are bound to certain limitations regarding memory. Furthermore it is of utmost importance to not use any floating point or SIMD code while in kernel space without explicitly handling register configurations. The latter is hard to get right so we simply delegate all data processing to user space.

[26] `https://www.kernel.org/doc/html/latest/#introduction-to-kernel-development`
[27] For example schedule a soft IRQ or tasklet when serving a hard IRQ and exit. Then use the soft IRQ to handle the rest.

Last but not least kernel space functions, while working similar most of the time, behave differently than their user space equivalents and sometimes in an unexpected manner. Most notably memory allocation works entirely differently and calling the same function from different contexts may not always yield a valid value. One has to check from which context one may call which function and should always check in which context the code is being executed at the moment.
These are the most fundamental differences one should always keep in mind as loosing one precious data would be a bad thing. We will mention further differences as needed

2.3.5 Writing drivers for our hardware

As we have our basic image and an idea of how one might write kernel code, it is about time we write some drivers. While understanding all the interfaces the Linux kernel provides might be a life time job, understanding the most important parts is relatively straight forward. For the vast majority of drivers one has to understand the following three concepts:

- **Devices** are a structure that represent physical devices. They are used to distinguish the actual device requiring attention. For example there might be multiple thumb drives connected to a computer while only having one driver for all of them. The driver then uses the information of the current device data to identify where to store your cat pictures.
- **Busses** organize the structure of device instances. Usually they correspond to real hardware structures like the PCIe network and feature functions to query devices. While a device itself is only a data structure a bus contains code that manages the devices registered with it. If for example an IRQ (interrupt request) from the PCIe stack happens the corresponding ISR (interrupt service routine) schedules the bus responsible. In this case it would be the one for the PCIe subsystem. The scheduled function should do something with it, which it turn might, for example (if a new device was connected), identify the driver responsible for said device and calls its probe function.
- **Frameworks** are collections of functions that allow similar drivers to share common functionality. For example a GPU driver may register with the DRM framework[28] and all it has to do is translate the drawing requests coming from the framework to its GPU drawing commands and not implement all the other functionality required for providing a graphics environment. As a result an application drawing things in userspace only has to know how to use the DRM framework and not every interface from every graphics driver out there.

The entry code of every late driver[29] is the *`module_init`* function of the driver regardless whether or not the module is compiled into the kernel and thus loaded right at the end of the boot

[28] DRM – which stands for Direct Rendering Manager – Is an abstraction layer between graphic card drivers and kernel modules that want to draw something.
[29] A late driver is a driver that is being loaded after start_kernel finished initialization.

process or being loaded at runtime using `modprobe`. The purpose of this method is to check if the module can be loaded and if so, initialize the module. In our case that would be registering with a bus in order to listen for our device type (a.k.a. the compatibility string from our device tree) and registering with a framework in order to communicate with userland. Regarding the bus system to use we have chosen the platform bus system as it is a good choice for memory mapped hardware that is directly connected to the CPUs via a memory mapped path and fixed[30]. We have chosen the misc devfs framework for our driver as it is causing our fuzzer to show up as `/dev/fuzzer` and is quite convenient to use without providing any special functionality (There is no hardware category "hardware fuzzer" in Linux).
After all our driver is quite simple. We write our init function which registers our driver with the bus and framework. We also need two functions that handle the case of the device being removed as well as a function that handles the unloading of the module. Next we write our probe function that is called for every fuzzer device the kernel finds (which should be not more than one but we make it capable of handling multiple instances – we may not need it but it is considered good practice). When this function gets called it extracts the memory location of the fuzzer and maps it into its address space. Furthermore we write functions that get called when a process opens the device file, reads from it, writes to it and closes it.
Having written those functions we than add some fundamental management logic and leave all the real processing to the userland process which we will notify about every new fault coming in. The complete driver source code can be reviewed at Listing 4.4 in the appendix.

2.3.6 Writing software to control the fuzzer

As we now have a device driver and can export our fuzzer to `/dev/fuzzer`, we need some userland software to handle it. For the sake of simplicity we have decided to write this software in Python. All it really does for now is, starting the fuzzer and logging the recorded exceptions. To do so it first opens the device file and configures and starts the fuzzer by writing the configuration commands as well as the magic word `start` to it. After doing so it reads all events that our driver sends us from said file. When we receive an event we check that it is actually a new one (as fuzzing might reveal a lot of faults that are actually the same) and if so format it to JSON and store it on a remote computer using the network.

2.4 Communicating with the fuzzer

We have the HPS now to control the fuzzer by setting the registers provided by the fuzzing control module. But how do we send the data to those registers and read the results back?

[30]As in not hot pluggable

2.4.1 How does one establish communication between the HPS and the fuzzer

First of all we should have a look how the HPS is capable of communicating with the configurable part of the FPGA. There are three AXI (have a look at chapter 2.4.6 for further information) bridges between the HPS and the FPGA and some conduit signals connected to the interrupt ports of the ARM cores.

The first bridge on the chip is called FPGA to HPS bridge. It is a 512 bit wide interconnect enabling the hardware application to access the HPS internal memory. A popular example on how to use this in a typical situation would be by sending the FPGA a memory address and letting the FPGA fetch data from the HPS without having the ARM cores of the HPS doing all the work of copying the data.

Another popular example would be implementing a core that is capable of reading and writing the HPS memory for debugging purposes. We did this only to be disappointed by the fact that we were only capable of reading data outside the external registers.

The second bridge is called HPS to FPGA bridge and lets the HPS access memory or registers provided by the FPGA. We will not use this one and use the last one instead.

The third bridge is called lightweight HPS to FPGA bridge and provides a 32 bit wide AXI Lite interconnect serving the same purpose as the second bridge but is more user friendly at the downside of providing less bandwidth. As our limiting factor is either the actual processing rate of the HPS or the test case generation rate anyway we can live with this penalty and take the much improved usability of AXI Lite compared to AXI.

We are utilizing this bridge to write to control registers of our fuzzer control unit which are mapped to memory as well as reading out the FIFO implemented as a ring buffer (see info box 2.4.8) used to store detected faults. Our underflow strategy is best described as pausing the kernel driver in case of eminent underflow. Our overflow strategy is based on the idea of slowing down the test case generation in case of the buffer reaching its limits.

While the overflow strategy results in a slow down in case of many faults being detected it is preferable over dropping them which would cause a loss of faults in such a case.

Info 2.4.8

A **ring buffer** is a buffer consisting of shared memory and two pointer registers. The first register is incremented every time data is being inserted into the buffer. The second register is incremented every time data is being read from the memory. Every now and then these registers overflow causing them to start at address 0 again, thus the name. In order to prevent data loss at the event of one side being faster than the other, the module has to stall under the condition of both registers containing the same content if one would increment.

2.4.2 What is this AXI (lite) thing?

As we were speaking of AXI (lite) we should clarify certain things a bit. First of all AXI is one of multiple[31] on chip interconnects available for designing hardware. Usually one cannot simply use I³C (or I²C for that matter) or other external interconnects to form a NoC[32] as those usually require bidirectional data transfers in order to save wires on PCBs. Having bidirectional or high-impedance I/O requires special I/O buffers that are (usually) only available at the boundaries of the FPGA connecting to outer pins. Furthermore if one is using purpose built interconnects one can utilize optimizations for the particular domain, respectively platform one is working with. As with DMA on chip interconnects are usually the same bit width as the memory controller ports they are interfacing with and optimized to be capable of delivering the data at the speed of the MMUs they are connected to.

As the wire count usually is of no concern on modern FPGA (or other chips) fabrics, one is using specialized interconnects on them. Furthermore the HPS is build using ARM cores and thus required to use AXI so we will have to use AXI as well.

Basically it allows us to read and write memory exposed by the modules we are speaking with. When dealing with multiple modules we need to make sure that each module has a different address space as there is no chip addressing using AXI. Besides that we can connect as many children[33] to the bus as we have space available on our FPGA.

The main difference between the two versions of AXI is that the lightweight version neither supports priority queues nor burst transmissions. Due to these limitations AXI Lite proves to be much simpler to implement. There are further differences like not supporting dynamic bit width changes as well but these smaller ones would not be that hard to implement.

2.4.3 Why do we need to write our own AXI core?

Intel provides a hand full of IP cores one can utilize within their own design. The majority of these cores either provides basic building blocks and thus do not require complex interconnects or use the proprietary Avalon bus developed by Altera (now owned by Intel). While not being particularly fast the Avalon bus is quite simple to use. We have to use the AXI protocol though as it is provided by the HPS as priorly stated.

When using Intel IP within QSYS (have a look at subsection 2.3.1) we can utilize tools to automatically bridge between the Avalon bus required by most IP and the AXI bus provided by the HPS. Doing so allows us to mix and match these components. However we cannot instantiate these

[31]Other popular examples include wishbone, Intels proprietary Avalon bus, STBus or even EIB (Element Interconnect Bus, not to be confused with European Installation Bus)

[32]Network on Chip – A fancy way of saying that one is connecting multiple IP cores on a single package

[33]The AXI specification calls a 'child' device slave and a 'controller' master. Due to ongoing debates about these terms we chose to not use those.

bridges individually and cannot use such a bridge for components not provided by Intel (including our own).
Unfortunately the components provided by Intel do not reflect our use case well. In fact they do not reflect most use cases but the most basic ones and encourage one to build an AXI child endpoint per purpose. The IP provided by Intel designed to interface an application running on the FPGA from the HPS is limited to a parallel IO module, an SPI module and a vector interrupt controller. While we intermediately used a parallel IO core for interfacing, non of these are well suited for our application.
This yields two options:

1. Build a bridge controller to some other interconnect
2. Build an AXI child controller ourself.

As building a bridge controller would include building an AXI child anyway we decided to build a AXI controller.

2.4.4 How does AXI work in detail?

Let us have look into the basic concepts of AXI Lite prior to describing the bridge core. It consists of three signalling groups, each carrying multiple data streams:

- Global Control Signals
- Write Control Signals
  - Address Handling
  - Data Handling
  - Control
- Read Control Signals
  - Address Handling
  - Data Return

Next we are going to have a look at them in detail.

Global Control Signals This group is quite lucid. It consists out of *A_CLK* and *A_RESETN*. As these names imply they handle the clock of the bus, which needs to guarantee a 50 percent duty cycle and an active low reset.

Write Control Signals This group can be subdivided into address handling, data handling and control (also called "Write Return" within the specifications). These channels are explained below.

Write Address handling First of all we have the `AW_VALID` and `AW_READY` signals. With `AW_READY` being `'1'` the child signals that it is ready to receive the next address. With `AW_VALID` the parent signals that the Address written to `AW_ADDR` is in a valid state. `AW_ADDR` itself contains the address the parent would like to write next.
The `AW_PROT` signal contains the write protection type and defines if one is accessing data or instructions and at which security level, encoded as a three bit twos complement ring number (see info box 2.4.9). It also states how one is accessing the mapped memory.
Furthermore there are `AW_ID` to identify different transmissions over the same channel as well as `AW_BURST`, `AW_LEN` and `AW_SIZE` to control the type, number (length) and size of messages within a burst. Last but not least there are `AW_LOCK` to mark a transaction as atomic, `AW_CACHE` to control the caching behaviour, `AW_QOS` to control the priority of the transmission and `AW_USER` to handle any user-defined protocol extensions on the write address side.

Write Data handling Again we have `W_VALID` and `W_READY` signalling valid data on the write channel by the parent and readiness to receive this data on the child side. We also have the `W_DATA` signals containing the actual data to write as well as the `W_STRB` signal to mask which bits defined in `W_DATA` are actually valid and supposed to be written. As it turns out there are quite a lot of write requests marking not a single bit as valid within `W_STRB`, but this behaviour is for some reason not considered to be an issue by Intel.
The `W_LAST` signal will be `HIGH` when the last transmission of the same burst is reached. Last but not least in this category the `W_USER` signal is provided to allow user defined protocol extensions and in this case is actually used by the HPS. Up to this date we could not identify what this signal actually does in this case.

Write return control channel In this channel we have the `B_VALID` signal notifying our child module about burst acknowledgement initializations and `B_READY` to mark the child ready for receiving these control sequences. Furthermore we have the `B_RESP` and `B_ID` signals to notify the child module about the state of a particular burst transmission. Of course there is the `B_USER` signal set dedicated to protocol extensions. This time it is pointing towards the next parent module.

Read channel The read channel is working quite similar to the write channel except for the fact that the directions are reversed and some minor differences we will look into now.
The address channel actually works exactly the same except that all signals are prefixed with `AR` instead of `AW`. The write data channel does not have a corresponding channel within this group.
The read return channel features two extra signals. First of all we have the `R_DATA` signal containing the data that was read. Secondly we have the `R_LAST` signal designed to mark the last

transmission of a read burst.
The *R_USER* signals are also facing towards the parent module.

Info 2.4.9

Rings define different layers of security within a computer. As a general rule of thumb: If the current ring number is higher one generally has less access rights. The ARM architecture however defines a higher number as more privileged. As hardware modules (or devices) are commonly mixed and matched this can lead to confusion and security issues.
Ring 0 is generally considered to be the access level an operating systems kernel has to the hardware. Userland processes usually [a] have access to ring three and above. Sometimes there are access privileges with negative numbers reserved to controllers handling deep functionality of the platform, such as the Intel Management Engine or other embedded controllers.

[a]The are exceptions when userland processes are partially mapped into kernel space for performance reasons.

2.4.5 Digression: Formal Verification of hardware designs

As hardware designs tend to be quite complex with a lot of things going on at the same time influencing each other it is quite easy to loose track of what is going on and hard to find the real cause of bugs (see info box 1.2.5). In order to find problems within a complete design one usually instantiates modules that perform debugging output (like for example signal tap modules). However as hardware tends to be quite complex, one can only focus on smaller parts of the design. If the cause of this problem is not within the module one is looking right at, it is usually difficult to find the issue.
Classic approaches to address this problem include simulating the module and comparing the results to predefined golden outputs similar to basic unit testing of software or simply looking at the wave forms with a human eye. The idea behind this is that if every single module is working correctly, everything should be fine. This method has two downsides though.
First, it is quite easy to miss an important test case or simply overlook them when analysing the graphs. Furthermore these hardware simulations are quite calculation intensive and may really take a long time to perform.
Fortunately thanks to the liberation of hardware development by the open source community there is finally an answer to this. It is called formal verification. While this approach never reached popularity within software development due to its complexity it is easier to do with hardware due to the way most designs look and is also faster to compute than simulating.
The basic idea is that instead of testing the module with a large set of example input, asserting the correct output and hoping that our tests actually cover all relevant conditions we define a set

of properties that our module must fulfil under a set of defined conditions. The verification software then parses the hardware description and generates a model of temporal logic. Finally the defined assumptions are solved using the model and the verification process succeeds if there is no contradiction or fails if there is one.

In case there are contradictions the software then generates an example of input signals that would cause such a contradiction. Hence we do not think that our module is working correctly, we let the computer prove it.

Of course there are downsides with this approach as well. First of all one has to really think about the constraints one needs to define. Secondly the tools, while being open source, are not yet well (or in the case of Intel Quartus not at all) supported by commercial EDA tools and thus usually do not include clock boundary information or information about oddities of the target platform. It is still possible to use them but some of their results might be better taken with a grain of salt.

One major upside though is that the open source community generally provides quite a lot of matured constraint files for common issues. This means that we can download a verification file for our AXI lite controller and test our design with it.

2.4.6 The actual core we wrote

Since we have a basic understanding of the way AXI lite works it is now time to have a look how one might realize such a core. We choose to build an asymmetric AXI child as we particularly do not need to read and write to the same registers.

Listing 2.2: AXI core registers

```vhdl
type register_bank is array (natural range <>) of std_logic_vector;
signal regs : register_bank(0 to amount_of_32_bit_control_registers - 1)(31 downto 0);

signal scheduled_data_to_write_bank : register_bank(0 to amount_of_32_bit_control_registers - 1)(31 downto 0);
signal scheduled_data_to_write_word : std_logic_vector(31 downto 0);
signal scheduled_data_to_write_strobe_mode : std_logic_vector(3 downto 0);
signal write_register_address : natural range 0 to amount_of_32_bit_control_registers - 1;

signal scheduled_read_addr : natural range 0 to (2 ** output_memory_address_width) - 1;
signal write_ready : std_logic;
signal read_ready : std_logic;
signal next_read_valid : std_logic;
signal awready : std_logic := '0';
```

First of all we need to declare the registers we are going to use. As we can see in Listing 2.2 there actually are not that many. Of course we need to store the control registers we want to write to. As we need to synchronize transactions we also need to buffer the data which is why we have the series of scheduling registers for both channels. Contrary to `migen`[34] we can not use automatic signal scheduling and thus have to also manually register our control signals in order to meet the signal travelling characteristics of the other AXI end points.

[34] `https://m-labs.hk/migen/manual/`

First of all we implement the write channel behaviour. Besides some combinatorical logic regarding synchronization and control this essentially boils down to

Listing 2.3: AXI write channel

```
if write_ready = '1' then
  regs(write_register_address) <= scheduled_data_to_write_bank(write_register_address);
end if;
```

and the reset logic. The read channel is as simple. All we have to do is to implement the control logic and forward the translated read address to our output memory and its data output port to the `R_DATA` port whenever we are allowed to use it.

Furthermore we need to take care of the acknowledgements and validity signals. The idea here is that we set a `VALID` flag whenever we were done initializing a burst and had a successful transaction. The acknowledgements are quite simple. Whenever we are in a clock cycle that is supposed to acknowledge something, we simply check if the channel is ready to transmit and in a valid state as well as a transaction was requested. The idea behind this is that if we are supposed to do something we assume that we did it as we do not enforce security policies within our own design — nor do something that might abort the transaction.

Last but not least we handle the control signals which follow some synchronous combinatoric logic which is also very intuitive. The remaining lines within the file are used up by boiler plate code being necessary due to low level HDLs like `VHDL` tending to be very infantile.

2.5 The P-Tile site of things

In this section we are going to have a look on how to use the heart of the fuzzer. The P-Tile.

2.5.1 What is the P-Tile and why do we need it?

As we are building a PCIe fuzzer it is mandatory to communicate with PCIe devices. As this protocol relies on very high speed transmissions one can not simply use arbitrary GPIO pins of the FPGA to build the differential signalling transceivers. This is where the P-Tile comes into play.

This piece of hardware provides transceivers to communicate with a connected PCIe network and may either act as an endpoint or root port for a channel, depending on the desired configuration. Each channel controls four PCIe lanes and one can combine these channels as ones design requires it. Not all combinations are possible though: One might configure four independent ones with four lanes each, two channels with eight lanes each or a single channel controlling all 16 lanes. If this is not enough one might use PCIe bifurcation to further multiply the number of combinations.

Each channel can either be configured to be an all-in-one memory mapped I/O adapter serving memory channels for each SR-IOV node, providing reconfigurable Avalon interfaces for accessing devices when in endpoint mode or can be used as upstream or downstream ports (see info box 2.5.10) when configured in TLP-bypass mode.

Info 2.5.10

Up- and downstream ports in PCIe describe the direction of data travelling across a PCIe network. A port connected towards the root port is called an upstream port. A port connected towards end points is called a downstream port.
In relation to the P-Tile, this notation was slightly abandoned. While it is still important to distinguish between ports facing a root complex or endpoints, these terms are also used to describe the capability of a port of initializing transactions. Intel provides a chart stating where which notation is used within their documentation. Unfortunately different versions of the documentation contradict with each other so we have to do some investigations from time to time to figure out what is meant in certain situations.

The IP block itself is divided into a hard IP portion and a portion that is implemented as soft logic. The hard IP deals with enumeration, link management and some basic configuration registers every PCIe capable device needs to provide in order to function. The soft logic part deals with everything else.
Between those two parts is a bridge consisting of a bus carrying the content of the TLPs and an Avalon bus for controlling the hard IP part. When a channel is configured in TLP bypass mode the soft logic equivalent is simply left out and this bridge is exposed. This is the place that we tap into with our fuzzer.

2.5.2 How to use the P-Tile IP

In order to use the P-Tile we need to take care of a few things. First of all we need to design an initialization module that configures the P-Tile in a correct way and handles events at run time. Furthermore we need to provide power management and above all we need to design our packet generator. But first things first.
As a starting point we take care of the control interface. We are provided with an Avalon-MM bus called `hip_reconfig` for every channel (here called port) we are using. Our goal is to setup the channel in a way enabling us to send arbitrary packets. The idea behind this is quite simple. We need to populate a hand full of registers with their associated configuration data and do so by implementing a sequential control module that iterates over a read only memory containing the data to be configured after reset. Its content can be reviewed in Table 2 in the appendix. The main difficulty resides in the fact that Intel only provides a long list of register definitions but no documentation on how to use these registers or which ones of them are actually relevant.

After having implemented this hardware one may notice, that our synthesis tool complains about a couple of timing constraint violations. When dealing with the P-Tiles interfaces one has to deal with multiple clock boundaries. While one of them definitely was a false path (see info box 2.5.11) the others were not. A property of the PCIe protocol is the fact that it may operate on a variable control sequence frequency between 125 MHz and 500 MHz. The solution to this non trivial timing issue is to operate all related circuitry on the clock provided by the P-Tile and cross the domains with skid buffers were necessary.

Info 2.5.11

A **false path** is a theoretical path between two registers that may never be reached. Sometimes the timing tools find these and if they are considered to be the longest path a signal may travel they will be marked as the critical path. If the duration of the signal on such a false path exceeds the theoretical time limit between clock cycles the timing tool will throw an error. In such a case one needs to mark this path as a false path in order for the timing tool to continue. A common source for false paths are clock boundaries (see info box 1.3.6) and Phase Locked Loop (PLL) feedbacks but one needs to make sure that the suspected false path is indeed unreachable.

Next we need to take care of the power management functions. This piece is one of the most important parts of a PCIe design as it has direct impact on the behaviour of other devices[35]. The basics are quite simple. We need to wake up a device that we would like to talk to if it is sleeping and put it back to sleep if we are finished talking to it and it was sleeping prior to us waking it up. Self-evidently we must not do this if a different device started talking to said device while we were and instead hand over this burdon to the new device. Furthermore we need to handle the power management notifications other devices toss towards us – but we are lazy on that side and do not implement any power states other than full power on and »we are about to loose power due to shutdown«. Last but not least we need to handle the events that occur when other parties veto our actions.

Table 2.2 describes the most important power states. While there are other ones, we really need to take care of those. Especially classes 2 and 3 can be divided into further states. One needs to know that even though the PCIe power states sound a lot like the ACPI ones, and often work in a similar way they are not one to one related and should not be confused with each other.

Most of the time our own device will reside in `L0` (function space `S0`) which means that our device will be enabled completely. When we are interacting with other devices we use the temporary `D`

[35]In fact we already crashed our test computer a couple of times due to improper power management behaviour. In theory the root complex should have the final word on power management – if your operating system implements it correctly. In practise at least Windows seams to be less talented on that side.

Table 2.2: Important power management states

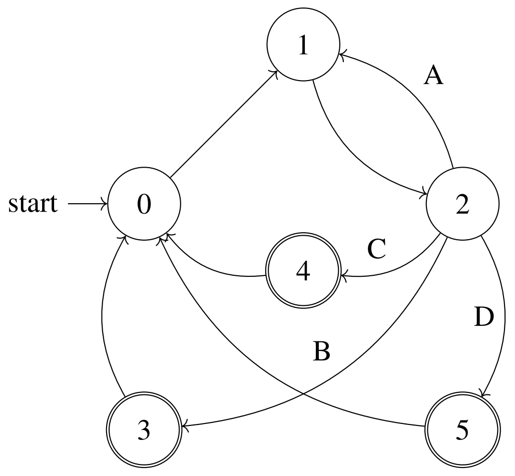
Sequence	State	State Code	D-State	D-State Code
–	`IDLE`	`"000"h`	`INVALID`	`"0000"h`
Completely turned on	`L0`	`"001"h`	`D0`	`"0001"h`
Preserve Power	`L1`	`"010"h`	`D1`	`"0010"h`
Selective Suspend	`L2`	`"011"h`	`D2`	`"0100"h`
Suspend	`L3`	`"100"h`	`D3`	`"1000"h`

states but ask the device about its preferred state prior to setting it.
One can really fill entire books about power management but these basics suffice to build basic devices. As a last step we need to implement the actual TLP generator which we will do next.

2.5.3 Digression: How do TLP packets work?

In order to build said generator we first need to have a look what to build. As with every recent protocol layer they can be splitted into a prefix, header, a payload part and an optional digest field. Each of these fields has a variable length (a multiple of 32 bit) but the maximum total length of a TLP is defined by the hardware capabilities of the decoder. A common value now a days would be 512 Bytes although older hardware may only process up to 256 Bytes [LDL14] and devices of the latest generation will happily support more than 4096 Bytes[Law14]. It is up to the sending device to figure this out and thus many devices implement protocol extensions to do so[36].
We distinguish between TLP with prefixes and ones without as they are optional. The next four or eight bytes after the last prefix (or the first four if there were none) define the header. After the header there may be any number (up to the transmission limited we discussed before) of data bytes and finally an optional TLP digest marking the last 32 bits.
Lets have a look at TLP prefixes. If the first three bits of a packet are `"100"b` we know that we have got ourself one. Otherwise it would indicate a `Fmt` field of a header. There may be any number of prefixes prepended to the packet and we know we are still parsing those as long as each 32 bit word still starts with this magic number. The next four bits define the type of prefix and all the remaining bits up to bit 0 can be used as a prefix payload. A list of current prefix types and their meaning can be obtained from Table 3 in the appendix.
The header of a TLP consists of four to eight bytes total. As one already saw in Table 1.1 there are the `Fmt` and `Type` fields defining the first byte. The content of the remaining three bytes is defined by the TLP type but they usually [37] contain their traffic class, request flags (memory alignment –

[36]They usually start with a maximum packet length of 128 Bytes as this seams to be the minimum length commonly supported, then exchange their hardware capabilities and choose the greatest number supported by both. For some reason this is not a standardized behaviour though.
[37]Except for legacy messages



0	reset P-Tile
1	send TLP
2	wait for next frame
3	report data access
4	report timeout
5	report error
A	expected error message
B	unexpected payload received
C	timeout after test case
D	unexpected error message

Figure 2.4: TLP generation state machine

little endian or big endian, presence of TLP processing hints (TPH) and error correction settings), a flag indicating if a digest is present after the payload and the length of the payload.
The header is followed by n bytes of data (or payload) where n is the length defined in the header. This part of a TLP is completely application dependant.
If the *TD* flag was set (*"1"*) there is still the digest containing error correction information. It is a 32 bit check sum for the payload.

2.5.4 Writing a TLP packet handler and processor

The last thing we had to do for these modules was to write the actual Transaction Layer Packet handler. The basic behaviour can be described with the automaton in Figure 2.4. The basic idea is that we enter a valid state and then try our fuzz cases.
We generate our TLP based on the configured masks. Their purpose is to define the patterns we would actually like to fuzz as we cannot search the entire PCIe address space in one run. The masks define the registers which we like to alter based on one of the following strategies:

- **iterate** over all bit combinations with increasing number patters
- **switch different bits** using an output combination o of $o_n = r \oplus (x + o_{n-1})$, where x is a circling checkerboard pattern (see info box 2.5.12) and r is the register seed.

We then shift each bit of the output patterns into its correct location configured by the positions register. The result is that we can define for each bit of a TLP if it should be *0*, *1*, or defined by a fuzzing pattern. This way our fuzzer is rather dumb but the best way we could come up with since we are bound to the limitations described in chapter 1.2.4.

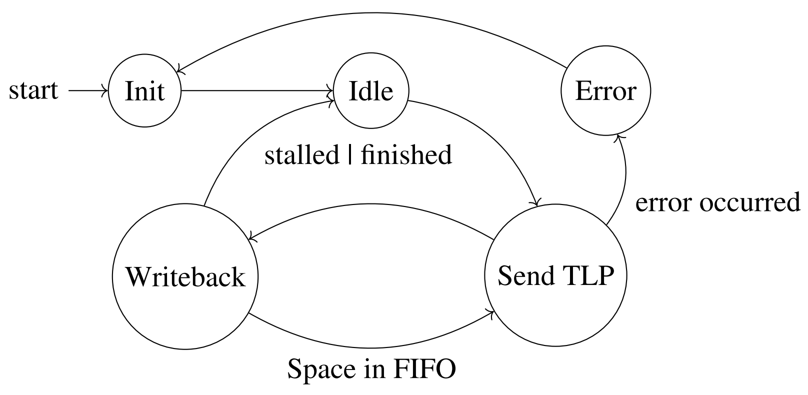


Figure 2.5: The general behaviour of the control module

Info 2.5.12

A **checkerboard pattern** is a pattern where a group of logical *1* are followed by a symmetric group of logical *0* and vice versa. A circling checkerboard pattern is one where for each new iteration (usually on the next clock cycle or state transition) the entire pattern does a barrel shift towards a constant side of a constant amount of bits.

2.6 Let us glue everything together: The control FSM

The final hardware module we wrote is the control state machine. Its purpose is to obey to the commands from the HPS and orchestrate the individual messaging modules.

2.6.1 The purpose of this module

As the TLP handlers only know of their own state but influence the total fuzzer we need an instance to keep track of all states the submodules are in. Furthermore this module keeps track with the HPS in order to stall the fuzzing if necessary. In short one can describe the behaviour of the state machine with the automaton in figure 2.5.

For the main part we generate fuzz cases, test them and report them back to the HPS if they are out of the ordinary. Assuming that we would find unexpected behaviour faster than we are capable of transmitting results to the HPS (or processing them with our software for that matter) we enter the idle state in order to let the HPS process. We also enter the idle state if the assigned task finished.

Assuming that there would be an error that a deeper module was not able to recover from, we report this error and reset our fuzzing hardware.

2.6.2 The different fault detection mechanisms

The final puzzle piece is the detection of faults. As already mentioned in the introduction, this is not as easy as we do not have access to the coverage our fuzz cases produce. We came up with the following possibility classes though.

First: Let's see if the designated error flags are set As one can see in Figure 2.4 we are capable of interpreting the error messages our DUT returns (if any). This causes the corresponding error flags to bet set in the return registers of the P-Tile enabling us to judge the test case.
It is also possible that the absence of such error reports indicates malfunction. Furthermore the PCIe protocol, like many other, defines answer time frames in which certain answers need to occur. If these constraints are violated (e.g. we get a timeout error) we are also notified and may react on such events.
We engineered our fuzzer to be configurable to assert the presence or absence of any combination of these misbehaviours.

Second: Let's check for unexpected behaviour Although we have not used this module in production yet we have build hooks into our fuzzing hardware to combine different properties of the total communication into assertions. For example one can configure the fuzzer to listen to unexpected incoming transmissions and report the last n transmitted Transaction Layer Packets if such an event occurs.
For example this could become handy if some packets cause the device under test to send out (potentially interesting) data.

2.7 Simulation

All modules are debugged using a combination of on-chip signal taps and model simulation. As we are currently struggling with a broken Linux kernel booting process, we decided to selectively include the simulation results of two critical modules within this work.

2.7.1 Simulating the AXI bridge

First of all we would like to show how our child module works. As the output of our formal verifier[38] is not very graphic (see Listing 4.5 in the appendix) we have decided to show wave form results from Modelsim, which is shipped in a stripped down version with Intel Quartus.

[38]We use SymbiYosys (`https://yosyshq.readthedocs.io/projects/sby/en/latest/`), a more powerful but also more complicated alternative would be CoSA (`https://github.com/cristian-mattarei/CoSA`).

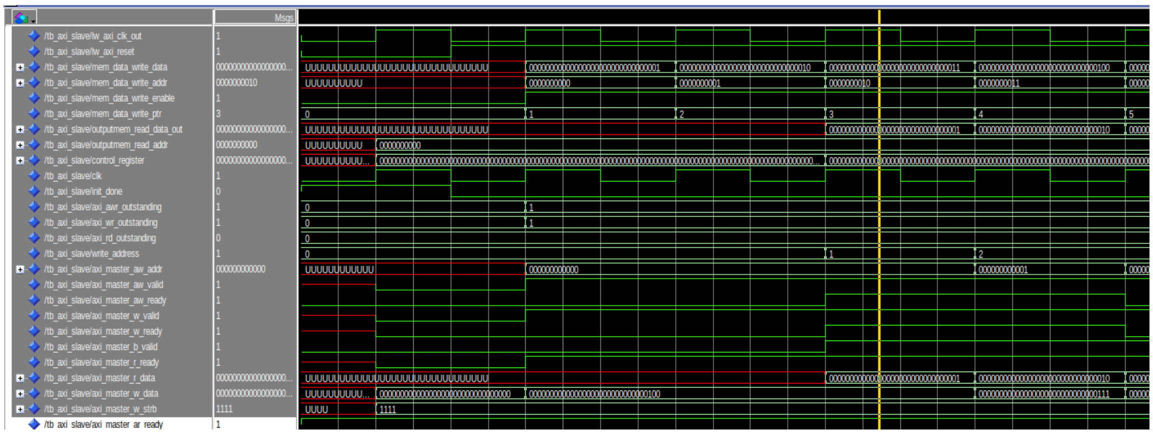

/tb_axi_slave/lw_axi_clk_out
/tb_axi_slave/lw_axi_reset
/tb_axi_slave/mem_data_write_data
/tb_axi_slave/mem_data_write_addr
/tb_axi_slave/mem_data_write_enable
/tb_axi_slave/mem_data_write_ptr
/tb_axi_slave/outputmem_read_data_out
/tb_axi_slave/outputmem_read_addr
/tb_axi_slave/control_register
/tb_axi_slave/clk
/tb_axi_slave/init_done
/tb_axi_slave/axi_awr_outstanding
/tb_axi_slave/axi_wr_outstanding
/tb_axi_slave/axi_rd_outstanding
/tb_axi_slave/write_address
/tb_axi_slave/axi_master_aw_addr
/tb_axi_slave/axi_master_aw_valid
/tb_axi_slave/axi_master_aw_ready
/tb_axi_slave/axi_master_w_valid
/tb_axi_slave/axi_master_w_ready
/tb_axi_slave/axi_master_b_valid
/tb_axi_slave/axi_master_r_ready
/tb_axi_slave/axi_master_r_data
/tb_axi_slave/axi_master_w_data
/tb_axi_slave/axi_master_w_strb
/tb_axi_slave/axi_master_ar_ready
Msgs

Figure 2.6: A screenshot from the write channel of our AXI bridge debug test bench

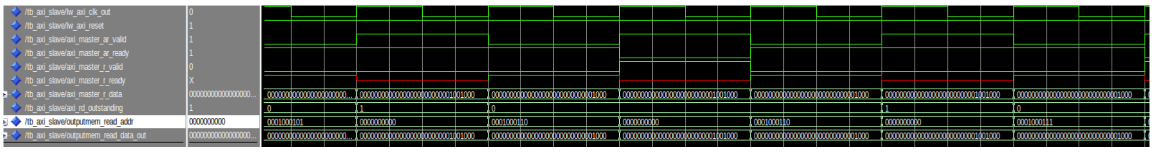

/tb_axi_slave/lw_axi_clk_out
/tb_axi_slave/lw_axi_reset
/tb_axi_slave/axi_master_ar_valid
/tb_axi_slave/axi_master_ar_ready
/tb_axi_slave/axi_master_r_valid
/tb_axi_slave/axi_master_r_ready
/tb_axi_slave/axi_master_r_data
/tb_axi_slave/axi_rd_outstanding
/tb_axi_slave/outputmem_read_addr
/tb_axi_slave/outputmem_read_data_out

Figure 2.7: A screenshot from the read channel of our AXI bridge debug test bench

In the Figure 2.6 one can observe that any input write requests from the AXI BUS are getting acknowledged and the output control registers are being updated accordingly. Furthermore one can observe in Figure 2.7 that any requested memory address is being properly delivered within two clock cycles.

2.7.2 Simulating the P-Tile control

Simulating the P-Tile behaviour is a done through connecting the Modelsim simulator with the corresponding P-Tile simulation binary. This binary in only intended to be used for regular PCIe behaviour though. Sending incorrect TLP causes this tool to crash. This way we can only show that we are capable of configuring the P-Tile correctly which we have done in Listing 4.6 in the appendix.
The important line is the one stating `Enumeration succeeded`. This means that the simulation tool was brought to a state where it can send and receive arbitrary TLP messages.

Table 2.3: Resource consumption and clock speeds of all major modules

Module	# ALUT	# reg. banks	# PLL	# BRAM	# DSP	Clocked at
clock and reset and initialization logic	637	1682	1	5	0	various speeds
fuzzing control	66	43	0	0	0	100 MHz
TLP generation	448	1436	0	0	1	350 MHz
HPS support	2170	2006	3	128	0	up to 500MHz
AXI Bridge	14	47	0	0	0	100 MHz

2.7.3 Achieved design quality

At the time of writing our synthesized design consumes $3432.5 * 10^4$ logic elements in total. This is approximately 1.5 percent of the overall size of the FPGA of which the majority is used for HPS support. After the place and routing steps are done this results in 3371 out of 1437240 ALUT[39] blocks being used, 5248 out of 1948800 register banks, 4 out of 24 PLLs, 135 out of 284672 BRAMs and 1 out of 4510 DSP blocks. This means that we have plenty of space for further refinements or extensions. Table 2.3 displays the resource consumption as well as clock speeds on a per-module basis.

Our modules achieve all required clock speeds of 100, 350 and 500 MHz respectively at a static power consumption of 19.9 watts with a maximum dynamic power consumption of 17.6 watts. The FPGA development kit is rated for up to 130 watts of power consumption so we are on the safe side.

Finally, our fuzzer is capable of running at the full PCIe 4.0 x8 data rate (about 15.75 GB/s) in linear mode, as long as the transmission of test cases is keeping up and the ring buffer does not stall. This would be about 30 million test cases per second. Assuming that every test case would produce a fault this number would decrease dramatically to only a couple of thousand test cases per second at most as the HPS could not keep up. Assuming that such a test case would crash significant components and larger parts of the PCIe network would need to recover we would only be capable of transmitting a hand full of test cases per second.

Searching the entire space of all TLP combinations for 512 Bit messages in such a linear fashion would take at least $1.45 * 10^{139}$ years and is thus not feasible. Instead it is more promising to concentrate on altering certain fields of TLP. A different approach would be, to let a classic fuzzer determine the areas of a TLP to fuzz. If one would fuzz attached components using a classic fuzzer dictating every single TLP we could still yield a couple thousand fuzz cases per second.

[39]Intel calls their logic cells ALUT which refers to the fact that they are build out of an ALU, LUT and a Mux.

3 How to use this fuzzer in the future

In the previous chapter we showed how we layed down the foundation for fuzzing PCIe devices that we can now work on. In a future work it is now possible to address the following topics:

3.1 Fuzzing the IOMMU

We would like to see if we can replicate the research from the Googles Security Team and scan an entire virtual address space for misconfigured access privileges. Doing so would prove the correct workings of our fuzzer as designing the control interface and correct enumeration features proved to be a major challenge.

It is also quite interesting to see if mayor operating system vendors have fixed the reported issues with the IO-MMU.

It would also be possible to go further and test if the IO-MMU (or any other MMUs involved with the PCIe network) is resilient to malformed memory transactions.

3.2 Fuzzing PCIe switches

As the PCIe network is a switched network with active network coupling devices these devices are as important to the overall security of the computer system as all members. Yet unlike Ethernet networks, there does not seam to be a lot of research interest into them. A broad scan of their behaviour could shine some first light into this matter.

3.3 Fuzzing other controllers

Furthermore we would like to fuzz the countless configuration interfaces exposed to the PCIe network that are unreachable from software running on CPUs as the hardware usually manages those. Examples include, but are not limited to, power management, enumeration notifications, bandwidth management, interrupt messages, MFVC, SR-IOV etc.

Doing so we need to find a way to generate fuzz cases efficiently as iterating over all possibilities is far too time consuming. We also need to find a way to collect feedback from the hardware that is being tested.

3.4 Looking into PCIe power management

During the development of our fuzzer we discovered that the actual implementations of power management features in PCIe devices is way more fragile than we would have anticipated. As a matter of fact we provoked malfunctions multiple times by accident. Given the huge potential attack surface of power management features for side channel attacks or inducing glitches in the device operations it would be an interesting target to look into.
To the authors knowledge there are plenty of research projects looking into controlling the power delivery of computer chips by directly manipulating the hardware in a physical. At the time of writing I am not aware of any project looking into this matter from the perspective of PCIe though.

3.5 Compute eXpress Link - The next big thing in data centres

CXL – which is short for Compute eXpress Link – is a protocol to interconnect device memories, including cache coherence guaranteeing mechanisms. While the next generation of Agilex FPGA are designed to have CXL offloading hardware, the protocol itself is defined as an additional layer over PCIe (or possibly any other protocol). Furthermore we have all the basic work in place to build a CXL debugging device on top of our TLP state modules. This would allow us to have a look at these new types of devices and search for potential issues.

3.6 Improving the fuzzer

Last but not least there is still plenty of room for improvements to the fuzzer. First of all it would be great to implement further pattern generation techniques. Furthermore the fault detection capabilities could still be improved. The fuzzer currently also lacks support for dynamic latency analysis for all send TLPs. Finally it would be helpful to expand the basic modules of the fuzzer towards a general TLP test framework.

4 Summary

Having a tool to analyse potential security flaws within the PCIe realm would be a beneficial addition to hardware debugging tools. As a consequence we developed an TLP message generator capable of sending arbitrary TLP as fuzz cases and registering the hardware responses. We proved its functionality in simulation.
Furthermore we developed an SoC connected to the FPGA using an AXI bridge and created an U-Boot configuration that is capable of being bootstrapped by the SDM. This involved porting Intels HPS reference implementations to our dev kit. We also composed an Linux-based operating system designed to control the fuzzing logic using the Yocto project. This includes a custom kernel module as well as userland software for convenience measures. While we were able to briefly test the prototype of the kernel module using QEMU there are ongoing issues with the modifications to the Linux boot code provided by Intel rendering us unable to test the fuzzer outside simulations for now.

Lessons learned We learned that, while it is not as easy as fuzzing software due to the common lack of output, it is ideed possible to test hardware using fuzzing though. It became clear that this approach has a lot of potential to find new issues within hardware as test benches of real world hardware are usually limited to the unit itself without extensive integration tests beyond ensuring basic operation. Moreover these unit test benches are often of poor quality.
Furthermore we learned that Intel does not ship their Agilex line of devices as complete products. This manifests as their shipped developent kits do not have completely working Linux support yet, feature missing and sometimes incorrect documentation and certain functions often simply do not work as intended. This also holds true for the software support for this series within their Quartus EDA suite which can best be described as being in beta testing status.
Last but not least we learned that there are a few ways to theoretically observe the behaviour of a black box hardware DUT but there is no default way to go about this. Instead one has to use domain dependent knowledge here.

Appendix

1 The build script

Listing 4.1: SD Image Build Script

```bash
#!/usr/bin/env bash

CLEAN=true
NUMBER_OF_CORES=32
QUARTUS_HOME=/opt/intelFPGA_pro/21.2

POKY_BRANCH=hardknott
META_INTEL_FPGA_BRANCH=master
META_INTEL_FPGA_REFDES_BRANCH=master
MENTOR_BRANCH=hardknott

#ATF_VER=v2.4.0
ATF_VER=v2.4.1
#ATF_VER=
UBOOT_VER=v2021.01

#LINUX_VER=5.4.114 # save kernel version in order for bb to choose correct syscall list
#LINUX_VER_APP=-lts
LINUX_VER=5.12
LINUX_VER_APP=

# Check for required external deps
if [[ -f ../output_files/top.sof ]]; then
  echo "Using ../output_files/top.sof as input for rbf generation"
else
  echo "Please make sure that ../output_files/top.sof exists"
  exit 1
fi

if [[ -d build_dir ]]; then
  if [[ "$CLEAN" == true ]]; then
    rm -rf build_dir
  else
    echo "Skipping build dir cleaning"
  fi
fi
mkdir -p build_dir
TOP_FOLDER=`pwd`
#cd build_dir

# Make sure the hw lib is in place
mkdir -p $TOP_FOLDER/repos
if [[ -d $TOP_FOLDER/repos/intel-socfpga-hwlib ]]; then
  #export CROSS_COMPILE=$TOP_FOLDER/repos/intel-socfpga-hwlib/tools/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu/
      bin/aarch64-linux-gnu-
  export CROSS_COMPILE=$TOP_FOLDER/repos/gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu/bin/aarch64-none-linux-gnu
      -
else
  pushd $TOP_FOLDER/repos > /dev/null
  echo "Crosscompiler not found. Performing setup."
  # commented out as currently broken (only arm works atm)
  #git clone https://github.com/altera-opensource/intel-socfpga-hwlib || exit
  #cd intel-socfpga-hwlib/tools
  #bash ./install_linaro.sh
  wget https://developer.arm.com/-/media/Files/downloads/gnu-a/10.2-2020.11/binrel/gcc-arm-10.2-2020.11-x86_64-aarch64
      -none-linux-gnu.tar.xz
  tar -xf gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu.tar.xz
  rm gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu.tar.xz
  export CROSS_COMPILE=`pwd`/gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu/bin/aarch64-none-linux-gnu-
  popd > /dev/null
fi

export ARCH=arm64
BUILDDIR=$TOP_FOLDER/build_dir

# compile bootloader
if [[ -d repos/u-boot-socfpga-rel_socfpga_v2020.10_21.05.01_pr ]]; then
  cd repos/u-boot-socfpga-rel_socfpga_v2020.10_21.05.01_pr
else
  cd $BUILDDIR
  git clone https://github.com/altera-opensource/u-boot-socfpga
  cd u-boot-socfpga
  git checkout socfpga_$UBOOT_VER || exit
  echo "fetched u-boot @$UBOOT_VER"
fi
make clean && make mrproper
#it's unclear if the line below is required or not
#sed -i 's/SOCFPGA_AGILEX/SOCFPGA/g' configs/socfpga_agilex_defconfig
if [[ -f .config ]]; then
  rm .config
fi
sed -i 's/socfpga_agilex_socdk/socfpga_agilex_fuzzer/g' configs/socfpga_agilex_defconfig || exit
sed -i 's/socfpga_agilex_socdk_qspi/socfpga_agilex_fuzzer/g' arch/arm/dts/Makefile || exit
#echo '/dtb-$(CONFIG_ARCH_SOCFPGA) += /a socfpga_agilex_fuzzer.dtb\\/' >> arch/arm/dts/Makefile
cp $TOP_FOLDER/ovrfiles/dt/*.dts* arch/arm/dts/ || exit
if [[ -z ATF_VER ]]; then
  DEVICE_TREE=socfpga_agilex_fuzzer.dtb make socfpga_agilex_defconfig || exit
else
  make socfpga_agilex_atf_defconfig || exit
fi
make -j $NUMBER_OF_CORES || exit
cp ./u-boot.img ../../build_dir
cp ./spl/u-boot-spl-dtb.hex ../../build_dir
cd $TOP_FOLDER

# compile linux
cd build_dir
# Intel provides an (slightly outdated and) customized version of the linux
# kernel capable of running on an HPS. So lets use that one.
if [[ "$CLEAN" == true ]]; then
  rm -rf linux-socfpga
fi
if [[ -d linux-socfpga ]]; then
  cd linux-socfpga
  git pull
else
  git clone https://github.com/altera-opensource/linux-socfpga
  cd linux-socfpga
  git checkout -b fw_build -t "origin/socfpga-$LINUX_VER""$LINUX_VER_APP"
  echo "CONFIG_JFFS2_FS=y" >> arch/arm64/configs/defconfig
  echo "CONFIG_OF_CONFIGFS=y" >> arch/arm64/configs/defconfig
  sed -i 's/CONFIG_ALTERA_SYSID=m/CONFIG_ALTERA_SYSID=y/g' arch/arm64/configs/defconfig
  echo "dtb-\$(CONFIG_ARCH_AGILEX) += socfpga_agilex_fuzzer.dtb" >> arch/arm64/boot/dts/intel/Makefile
  echo "dtb-\$(CONFIG_ARCH_AGILEX) += socfpga_agilex_minimal.dtb" >> arch/arm64/boot/dts/intel/Makefile
fi
cp ../../ovrfiles/dt/* arch/arm64/boot/dts/intel/
make clean && make mrproper
#make socfpga_defconfig
make defconfig
#make -j $NUMBER_OF_CORES Image dtbs modules && make -j $NUMBER_OF_CORES modules_install INSTALL_MOD_PATH=
      modules_install || exit
make -j $NUMBER_OF_CORES Image dtbs || exit

#cp arch/arm64/boot/Image ../kernel_image
cp arch/arm64/boot/dts/intel/socfpga_agilex_fuzzer.dtb ../
cp arch/arm64/boot/dts/intel/socfpga_agilex_minimal.dtb ../
#cp -r modules_install ../kernel_modules
cd ..
```

```

# These functions are shamelessly copied (and then modified) from the yocto examples

MACHINE=agilex
IMAGE=grsd # nand # valid are nand ghrd qspi
location=$TOP_FOLDER/build_dir

## Spin up python web server for bb support in background and trap its exit
echo "Spun up web server on port 8000 for `pwd`"
python3 -m http.server &
trap "echo Tring to close python webserver && trap - SIGTERM && kill -- -$$" SIGINT SIGTERM EXIT

sanity_bitbake() {
while true; do
  BITBAKE_PROCESS_RUNNING=`ps aux | grep bitbake | wc -l`
  if [ $BITBAKE_PROCESS_RUNNING -eq 1 ]; then
    break;
  else
    echo -e "\n[INFO] There is already an instance of bitbake process running in the background. Waiting.."
    sleep `expr $RANDOM % 30`
  fi
done
}

environment_cleanup() {
  if [ -d "$BUILDDIR" ]; then
    echo -e "\n[INFO] Cleanup the /tmp, /conf folders in the workspace for next build"
    pushd $BUILDDIR > /dev/null
      rm -rf $MACHINE-rootfs/tmp/
      rm -rf $MACHINE-rootfs/conf/

      if [ -d $MACHINE-images ]; then
        echo "[INFO] Cleanup images folder in the workspace for next build"
        rm -rf $MACHINE-images
      fi
    popd > /dev/null
  fi

  if [ ! -d $BUILDDIR/$MACHINE-rootfs ]; then
    echo -e "\n[INFO] Create build workspace"
    mkdir -p $BUILDDIR/$MACHINE-rootfs
  fi

  if [ ! -d $BUILDDIR/$MACHINE-images ]; then
    echo -e "\n[INFO] Create image staging area"
    mkdir -p $BUILDDIR/$MACHINE-images
  fi
  STAGING_FOLDER=$BUILDDIR/$MACHINE-images

  if [ -d "$BUILDDIR/meta-intel-fpga" ]; then
    echo -e "\n[INFO] Remove meta-intel-fpga and meta-intel-fpga-refdes if they are already exist in BUILDDIR"
    pushd $BUILDDIR > /dev/null
      rm -rf meta-intel-fpga/
      rm -rf meta-intel-fpga-refdes/
    popd > /dev/null
  fi
}

# Update existing meta layers or clone a new one if it does not exists
get_latest_meta() {
  pushd $BUILDDIR > /dev/null
    # Make sure build_dir/poky is in correct state
    if [ -d "poky" ]; then
      echo -e "\n[INFO] Poky source tree available. Proceed to update poky source tree ($POKY_BRANCH) branch."
      pushd poky > /dev/null
        git checkout master
        git branch -D $POKY_BRANCH || true
        git fetch origin
        git pull
        git checkout $POKY_BRANCH
      popd > /dev/null
    else
    echo -e "\n[INFO] Poky source tree not available. Proceed to download poky source tree ($POKY_BRANCH) branch."
    git clone -b $POKY_BRANCH https://git.yoctoproject.org/git/poky.git
  fi

  # make sure the meta layers are in the correct state
  if [ -d "meta-intel-fpga" ]; then
    pushd meta-intel-fpga > /dev/null
      echo -e "\n[INFO] meta-intel-fpga source tree available. Proceed to update meta-intel-fpga source tree (master
          ) branch."
      git checkout -- .
      git checkout $META_INTEL_FPGA_BRANCH
      git fetch origin
      git pull
    popd > /dev/null
  else
    echo -e "\n[INFO] meta-intel-fpga source tree not available. Proceed to download meta-intel-fpga source tree (
          master) branch."
    git clone -b $META_INTEL_FPGA_BRANCH https://git.yoctoproject.org/git/meta-intel-fpga
  fi

  if [ -d "meta-intel-fpga-refdes" ]; then
    echo -e "\n[INFO] meta-intel-fpga-refdes source tree available. Proceed to update meta-intel-fpga-refdes source
          tree (master) branch."
    pushd meta-intel-fpga-refdes > /dev/null
      git checkout -- .
      git checkout $META_INTEL_FPGA_REFDES_BRANCH
      git fetch origin
      git pull
    popd > /dev/null
  else
    echo -e "\n[INFO] meta-intel-fpga-refdes source tree not available. Proceed to download meta-intel-fpga-refdes
          source tree (master) branch."
    git clone -b $META_INTEL_FPGA_REFDES_BRANCH https://github.com/altera-opensource/meta-intel-fpga-refdes.git
  fi

  if [ -d "meta-mentor" ]; then
    echo "TODO: add propper handling" # TODO
  else
    git clone -b $MENTOR_BRANCH https://github.com/MentorEmbedded/meta-mentor.git
    #git checkout $MENTOR_BRANCH
  fi

  ## Patch in meta-mel
  #cp -a ./meta-mentor/meta-mel/lib ./meta-intel-fpga/ # TODO check if really not required

  # copy core and image tools from refdes to base repo
  echo "[INFO] Copying core and image tools from refdes to base repo"
  cp -r ./meta-intel-fpga-refdes/recipes-core ./meta-intel-fpga/
  cp -r ./meta-intel-fpga-refdes/recipes-support ./meta-intel-fpga/
  cp -r ./meta-intel-fpga-refdes/recipes-tools ./meta-intel-fpga/
  cp -r ./meta-intel-fpga-refdes/recipes-bsp/u-boot ./meta-intel-fpga/recipes-core

  cp -r ./meta-intel-fpga-refdes/recipes-images ./meta-intel-fpga/
  cp -r ./meta-intel-fpga-refdes/recipes-gsrd/socfpga-gsrd-webcontent ./meta-intel-fpga/recipes-images/
  cp -r ./meta-intel-fpga-refdes/recipes-gsrd/socfpga-gsrd-lighttpd-conf ./meta-intel-fpga/recipes-images/

  mkdir -p meta-intel-fpga/recipes-bsp/bitsream
  cp ../ovrfiles/hw-ref-design.bb ./meta-intel-fpga/recipes-bsp/bitsream/hw-ref-design.bb
  cp ../ovrfiles/packagegroup-common-essential.bb ./meta-intel-fpga/recipes-images/packagegroups/packagegroup-common
       -essential.bb
  cp ../ovrfiles/packagegroup-common-essential/* ./meta-intel-fpga/recipes-images/packagegroups/

  if [ -d "meta-openembedded" ]; then
    echo -e "\n[INFO] meta-openembedded source tree available. Proceed to update meta-openembedded source tree (
          $YOCTO_BRANCH) branch."
    pushd meta-openembedded > /dev/null
      git checkout master
      git branch -D $POKY_BRANCH || true
      git fetch origin
      git pull
      git checkout $POKY_BRANCH
    popd > /dev/null
    else
      echo -e "\n[INFO] meta-openembedded source tree not available. Proceed to download meta-openembedded source tree
          ($YOCTO_BRANCH) branch."
      git clone -b $POKY_BRANCH https://git.openembedded.org/meta-openembedded
    fi

    get_version_info
  popd > /dev/null
}

configure_meta() {
  pushd $BUILDDIR > /dev/null
    if [ ! -z $ATF_VER ]; then
      ATF_BRANCH=socfpga_$ATF_VER
      # Get COMMIT ID HASH from arm-trusted-firmware
      echo -e "\n[INFO] Update ATF recipe with latest source revision for ATF $ATF_VER "
      COMMIT_HASH="$(git ls-remote https://github.com/altera-opensource/arm-trusted-firmware.git $ATF_BRANCH | awk '{
          print $1}')"
      # Change SRCREV in ATF recipe to get latest commit
      pushd meta-intel-fpga/recipes-bsp/arm-trusted-firmware > /dev/null
        sed -i /SRCREV\ =\ \"/d arm-trusted-firmware_`cut -d. -f1-2 <<< "$ATF_VER"`.bb
        echo "SRCREV = \"$COMMIT_HASH\"" >> arm-trusted-firmware_`cut -d. -f1-2 <<< "$ATF_VER"`.bb
      popd > /dev/null
    else
      rm -rf meta-intel-fpga/recipes-bsp/arm-trusted-firmware
    fi

    if [ ! -z $UBOOT_VER ]; then
      UBOOT_SOCFGPA_BRANCH=socfpga_$UBOOT_VER
      # Get COMMIT ID HASH from u-boot-socfpga
      echo -e "\n[INFO] Update u-boot recipe with latest source revision for u-boot $UBOOT_VER "
      COMMIT_HASH="$(git ls-remote https://github.com/altera-opensource/u-boot-socfpga.git $UBOOT_SOCFGPA_BRANCH | awk
          '{print $1}')"
      # Change SRCREV in u-boot recipe to get latest commit
      pushd meta-intel-fpga/recipes-bsp/u-boot > /dev/null
        echo "Override u-boot-socfpga SRC_REV in recipe with commit ID $COMMIT_HASH"
        sed -i /SRCREV\ =\ \"/d u-boot-socfpga_$UBOOT_VER.bb
        echo "SRCREV = \"$COMMIT_HASH\"" >> u-boot-socfpga_$UBOOT_VER.bb
      popd > /dev/null
    fi

    if [ ! -z $LINUX_VER ]; then
      LINUX_SOCFPGA_BRANCH="socfpga-$LINUX_VER"$LINUX_VER_APP
      # Get COMMIT ID HASH from linux-socfpga
      echo -e "\n[INFO] Update kernel recipe with latest source revision for kernel $LINUX_VER "
      COMMIT_HASH="$(git ls-remote https://github.com/altera-opensource/linux-socfpga.git $LINUX_SOCFPGA_BRANCH | awk
          '{print $1}')"
      # Change SRCREV in kernel recipe to get latest commit
      pushd meta-intel-fpga/recipes-kernel/linux > /dev/null
        echo "Override linux-socfpga$LINUX_VER_APP SRC_REV in recipe with commit ID $COMMIT_HASH"
        sed -i /SRCREV\ =\ \"/d "linux-socfpga$LINUX_VER_APP"_$LINUX_VER.bb
        echo "SRCREV = \"$COMMIT_HASH\"" >> "linux-socfpga$LINUX_VER_APP"_$LINUX_VER.bb
        echo "Override LINUX_VER in recipe with version $LINUX_VER"
        sed -i /LINUX\_VERSION\ =\ \"/d "linux-socfpga$LINUX_VER_APP"_$LINUX_VER.bb
        echo "LINUX_VERSION = \"$LINUX_VER\"" >> linux-socfpga"$LINUX_VER_APP"_$LINUX_VER.bb
      popd > /dev/null
    fi
  popd > /dev/null
}

setup_bb_build_env() {
  pushd $BUILDDIR > /dev/null
    echo -e "\n[INFO] Source poky/oe-init-build-env to initialize poky build environment"
    source poky/oe-init-build-env $BUILDDIR/$MACHINE-rootfs/

    # Settings for local.conf
    echo -e "\n[INFO] Update local.conf"
    sed -i /MACHINE/d conf/local.conf
    sed -i /UBOOT_CONFIG/d conf/local.conf
    sed -i /IMAGE\_TYPE/d conf/local.conf
    sed -i /SRC\_URI\_/d conf/local.conf

    echo "MACHINE = \"${MACHINE}\"" >> conf/local.conf
    echo "DL_DIR = \"$location/downloads\"" >> conf/local.conf
    echo 'DISTRO_FEATURES_append = " systemd"' >> conf/local.conf
    echo 'VIRTUAL-RUNTIME_init_manager = "systemd"' >> conf/local.conf
    echo "IMAGE_TYPE = \"$IMAGE\"" >> conf/local.conf
    echo 'IMAGE_FSTYPES += "jffs2 tar.gz"' >> conf/local.conf
    echo 'EXTRA_IMAGE_FEATURES += "tools-sdk tools-debug"' >> conf/local.conf
    echo 'IMAGE_ROOTFS_SIZE = "262144"' >> conf/local.conf
    # Linux
    echo "PREFERRED_PROVIDER_virtual/kernel = \"linux-socfpga$LINUX_VER_APP\"" >> conf/local.conf
    if [ ! -z $LINUX_VER ]; then
      echo "PREFERRED_VERSION_linux-socfpga$LINUX_VER_APP = \"`cut -d. -f1-2 <<< "$LINUX_VER"`%\"" >> conf/local.conf
    fi
    # U-boot
    echo "UBOOT_CONFIG = \"$UB_CONFIG\"" >> conf/local.conf
    echo 'PREFERRED_PROVIDER_virtual/bootloader = "u-boot-socfpga"' >> conf/local.conf
    if [ ! -z $UBOOT_VER ]; then
      echo "PREFERRED_VERSION_u-boot-socfpga = \"$UBOOT_VER%\"" >> conf/local.conf
    fi

    # ATF
    if [ ! -z $ATF_VER ]; then
      echo "PREFERRED_VERSION_arm-trusted-firmware = \"`cut -d. -f1-2 <<< "$ATF_VER"`\"" >> conf/local.conf
    else
      #sed -i 's/arm-trusted-firmware\ bash/bash/g' meta-intel-fpga/recipes-core/u-boot/u-boot-socfpga_v20%.bbappend
      #sed -i 's/socfpga_agilex_atf_defconfig/socfpga_agilex_defconfig/g' meta-intel-fpga/recipes-bsp/u-boot/u-boot-socfpga-common.inc
      #sed -i 's/socfpga_agilex_qspi_atf_defconfig/socfpga_agilex_qspi_defconfig/g' meta-intel-fpga/recipes-bsp/u-boot/u-boot-socfpga-common.inc
      #sed -i 's/agilex-socdk-atf/agilex-socdk' meta-intel-fpga/conf/machine/agilex-extra.conf
      echo "Disabling ATF support in u-boot. Patching from `pwd`"
      sed -i 's/arm-trusted-firmware\ bash/bash/g' $TOP_FOLDER/build_dir/meta-intel-fpga/recipes-core/u-boot/u-boot-socfpga_v20*.bbappend || exit
      # Also deactivate itb patching
      sed -i 's/cp \$.DEPLOY_DIR_IMAGE..bl31.bin /touch /g' $TOP_FOLDER/build_dir/meta-intel-fpga/recipes-core/u-boot/u-boot-socfpga_v20*.bbappend || exit
      sed -i 's/socfpga_agilex_atf_defconfig/socfpga_agilex_defconfig/g' $TOP_FOLDER/build_dir/meta-intel-fpga/recipes-bsp/u-boot/u-boot-socfpga-common.inc || exit
      sed -i 's/socfpga_agilex_qspi_atf_defconfig/socfpga_agilex_qspi_defconfig/g' $TOP_FOLDER/build_dir/meta-intel-fpga/recipes-bsp/u-boot/u-boot-socfpga-common.inc || exit
      sed -i 's/agilex-socdk-atf/agilex-socdk/g' $TOP_FOLDER/build_dir/meta-intel-fpga/conf/machine/agilex-extra.conf || exit
      sed -i 's/agilex-socdk-atf/agilex-socdk/g' $TOP_FOLDER/build_dir/agilex-rootfs/conf/local.conf || exit
    fi

    # MACHINE specific settings
    echo "require conf/machine/agilex-extra.conf" >> conf/local.conf

    # Settings for bblayers.conf
    echo -e "\n[INFO] Update bblayers.conf"
    echo 'BBLAYERS += " ${TOPDIR}/../meta-intel-fpga "' >> conf/bblayers.conf # We'd like basic HW support
    #echo 'BBLAYERS += " ${TOPDIR}/../meta-intel-fpga-refdes "' >> conf/bblayers.conf # We only use parts of the reference image
    echo 'BBLAYERS += " ${TOPDIR}/../meta-openembedded/meta-oe "' >> conf/bblayers.conf # A userland? Yes, please
    echo 'BBLAYERS += " ${TOPDIR}/../meta-openembedded/meta-networking "' >> conf/bblayers.conf
    echo 'BBLAYERS += " ${TOPDIR}/../meta-openembedded/meta-python "' >> conf/bblayers.conf
    echo 'BBLAYERS += " ${TOPDIR}/../meta-openembedded/meta-filesystems "' >> conf/bblayers.conf # 'Cause sshfs is awesome
    echo 'BBLAYERS += " ${TOPDIR}/../meta-mentor/meta-mentor-common "' >> conf/bblayers.conf
    echo 'BBLAYERS += " ${TOPDIR}/../meta-mentor/meta-mel "' >> conf/bblayers.conf # We need kernel patching tools as our board isn't mainline supported.

    pushd $BUILDDIR > /dev/null
    cd ..
    echo "Patching configs relative to `pwd`"
    # TODO: Use sed to patch selected dtb in ovrfiles/conf/agilex.conf
    rm ./meta-intel-fpga/conf/machine/agilex.conf || exit
    rm ./meta-intel-fpga/conf/machine/agilex-extra.conf || exit
    cp ../ovrfiles/conf/* meta-intel-fpga/conf/machine/ || exit

    # Patch merged intel-fpga-layer
    #sed -i "s/u-boot-socfpga-scr/u-boot-socfpga/g" ./meta-intel-fpga/conf/machine/agilex-extra.conf || exit
    #echo "KERNEL_DEVICETREE += \"socfpga_agilex_fuzzer.dtb\"" >> ./meta-intel-fpga/recipes-kernel/linux-socfpga.inc
    recipetool appendsrcfile -wm agilex ./meta-intel-fpga virtual/kernel ../ovrfiles/dt/socfpga_agilex_fuzzer.dtsi 'arch/${ARCH}/boot/dts/intel/socfpga_agilex_fuzzer.dtsi'
    recipetool appendsrcfile -wm agilex ./meta-intel-fpga virtual/kernel ../ovrfiles/dt/socfpga_agilex_fuzzer.dts 'arch/${ARCH}/boot/dts/intel/socfpga_agilex_fuzzer.dts'
    #recipetool appendsrcfile -wm agilex ./meta-intel-fpga virtual/kernel ../ovrfiles/dt/socfpga_agilex_ghrd.dtsi 'arch/${ARCH}/boot/dts/socfpga_agilex_ghrd.dtsi' # The file in this dir is only required because poky is wiered about invoking the kernel makefile
    #recipetool appendsrcfile -wm agilex ./meta-intel-fpga virtual/kernel ../ovrfiles/dt/socfpga_agilex_fuzzer.dts 'arch/${ARCH}/boot/dts/socfpga_agilex_fuzzer.dts' # The file in this dir is only required because poky is wiered about invoking the kernel makefile
    #recipetool kernel_add_dts ./meta-intel-fpga ../ovrfiles/dt/socfpga_agilex_fuzzer.dts
    #echo "require recipes-kernel/linux/linux-dtb.inc" >> ./meta-intel-fpga/recipes-kernel/linux/linux-socfpga-lts_%.bbappend
    echo 'KERNEL_DEVICETREE += "intel/socfpga_agilex_fuzzer.dtb"' >> ./meta-intel-fpga/recipes-kernel/linux/"linux-socfpga$LINUX_VER_APP"_%.bbappend
    recipetool appendsrcfile -wm agilex ./meta-intel-fpga virtual/kernel ../ovrfiles/dt/Makefile.patch 'arch/${ARCH}/boot/dts/intel/Makefile.patch'
    #recipetool appendsrcfile -wm agilex ./meta-intel-fpga virtual/kernel ../ovrfiles/dt/DummyMakefile.patch 'arch/${ARCH}/boot/dts/Makefile.patch'
    popd > /dev/null
  popd > /dev/null
}

build_linux_distro() {
  pushd $BUILDDIR > /dev/null
    echo -e "\n[INFO] Clean up previous kernel build if any"
    bitbake virtual/kernel -c cleanall
    echo -e "\n[INFO] Clean up previous u-boot build if any"
    bitbake u-boot-socfpga -c cleanall
    echo -e "\n[INFO] Clean up previous ghrd build if any"
    #bitbake hw-ref-design -c cleanall
    echo "=======================WARNING======================="
    echo "Commented out refds cleaning"

    echo -e "\n[INFO] Start bitbake process for target config.."
    #bitbake console-image-minimal gsrd-console-image 2>&1
    # WARN not building the minimal image as well might break things.
    bitbake console-image-minimal || exit
  popd > /dev/null
}

build_image() {
  echo -e "\n[INFO] Copy the build output and store in $STAGING_FOLDER\n"
  pushd $location/$MACHINE-rootfs/tmp/deploy/images/$MACHINE/ > /dev/null

    cp -vrL *-$MACHINE.tar.gz $STAGING_FOLDER/  || echo "[INFO] No tar.gz found."
    cp -vrL *-$MACHINE.jffs2 $STAGING_FOLDER/ || echo "[INFO] No jffs2 found."
    cp -vrL *-$MACHINE.wic $STAGING_FOLDER/   || echo "[INFO] No jffs2 found."


    cp -vrL zImage $STAGING_FOLDER/ || cp -vrL Image $STAGING_FOLDER/ || echo "[INFO] No zImage / Image found."


    cp -vrL *.dtb $STAGING_FOLDER/  || echo "[INFO] No dtb found."
  popd > /dev/null

  mkdir -p $STAGING_FOLDER/u-boot-$MACHINE-socdk-atf

  ub_cp_destination=$STAGING_FOLDER/u-boot-$MACHINE-socdk-atf
  pushd $location/$MACHINE-rootfs/tmp/work/$MACHINE-poky-*/u-boot-socfpga/1_v20*/build/socfpga_$MACHINE*/ > /dev/null
    cp -vL u-boot $ub_cp_destination
    cp -vL u-boot-dtb.bin $ub_cp_destination
    cp -vL u-boot-dtb.img $ub_cp_destination
    cp -vL u-boot.dtb $ub_cp_destination
    cp -vL u-boot.img $ub_cp_destination
    cp -vL u-boot.map $ub_cp_destination
    cp -vL spl/u-boot-spl $ub_cp_destination
    cp -vL spl/u-boot-spl-dtb.bin $ub_cp_destination
    cp -vL spl/u-boot-spl.dtb $ub_cp_destination
    cp -vL spl/u-boot-spl.map $ub_cp_destination


    cp -vL spl/u-boot-spl-dtb.hex $ub_cp_destination
    cp -vL u-boot.itb $ub_cp_destination
  popd > /dev/null

  pushd $ub_cp_destination > /dev/null
    chmod 644 u-boot-dtb.img
    chmod 644 u-boot.img
    chmod 744 u-boot.itb || echo "[INFO] File u-boot.itb not found for this build configuration."
  popd > /dev/null

  pushd $location/$MACHINE-rootfs/tmp/work/$MACHINE-poky-*/u-boot-socfpga-scr/1.0*/deploy-u-boot-socfpga-scr > /dev/
      null
    cp -vL u-boot.scr $ub_cp_destination
  popd > /dev/null

  pushd $location/$MACHINE-rootfs/tmp/deploy/images/$MACHINE > /dev/null
    cp -vrL ${MACHINE}_${IMAGE}_ghrd/ $STAGING_FOLDER/.
    #cp -vrL ${MACHINE}_ghrd/ $STAGING_FOLDER/.
  popd > /dev/null
  echo -e "\nBeginning merge sequence\n"
  BUILDDIR=$TOP_FOLDER/build_dir
  pushd $BUILDDIR > /dev/null
    echo -e "\n===== DEBUG =====\nBuilding image in `pwd`"
    echo -e "\nCalling:\n"
    echo bash $TOP_FOLDER/createImage.bash $STAGING_FOLDER $MACHINE $BUILDDIR $QUARTUS_HOME
    echo ""
    sudo bash $TOP_FOLDER/createImage.bash $STAGING_FOLDER $MACHINE $BUILDDIR $QUARTUS_HOME
  popd > /dev/null
}

build_rbf_file() {
  echo "Debug: Current working directory: `pwd`"
  pushd $BUILDDIR > /dev/null
  # The existance of ../output_files/top.sof has already been assured
  #bash $QUARTUS_HOME/nios2eds/nios2_command_shell.sh quartus_pfg -c ../../output_files/top.sof $BUILDDIR/flash_image.
      jic -o device=MT25QU02G -o flash_loader=AGFB014R24A2E3VR0 -o hps_path=$BUILDDIR/u-boot-spl-dtb.hex -o mode=
      ASX4 || exit
  echo "Creating rbf file..."
  bash $QUARTUS_HOME/nios2eds/nios2_command_shell.sh quartus_pfg -c ../../output_files/top.sof $BUILDDIR/flash_image.
      core.rbf -o hps_path=$BUILDDIR/u-boot-spl-dtb.hex || exit
  popd > /dev/null
}

# build userland
cd $BUILDDIR
sanity_bitbake
if [[ "$CLEAN" == true ]]; then
  environment_cleanup
  echo "=======================WARNING======================="
  echo "Cleaned BB files prior to build"
fi

get_latest_meta
configure_meta

if [ "$IMAGE" == "qspi" ]; then
  UB_CONFIG="$MACHINE-socdk-$IMAGE-atf"
else
  UB_CONFIG="$MACHINE-socdk-atf"
fi

build_rbf_file
setup_bb_build_env
build_linux_distro
build_image


echo "\n\n=========================== FINISHED ==========================="
echo "\nPlease have a look at the $STAGING_FOLDER folder for general purpose images."
echo "The sd card optimized image has been written to $BUILDDIR/sdcard-build/sdimage.img"
echo "Burn this image to an micro sd card and use it as your boot device."
echo "Use the jic file located at $BUILDDIR/flash_image.jic to program the fpga"
```

2 The kernel modules source code overview

Listing 4.2: The device tree include source

```
/*
* Add this piece of dtsi fragment as #include "socfpga_agilex_ghrd.dtsi"
* in the file socfpga_agilex_socdk.dts. Compile it in the kernel along with
* socfpga_agilex.dtsi.
*/

/{
  soc {

    led_pio: gpio@1080 {
        compatible = "altr,pio-1.0";
        reg = <0xf9001080 0x8>;
        altr,gpio-bank-width = <4>;
        #gpio-cells = <2>;
        gpio-controller;
        resetvalue = <0>;
        status = "disabled";
    };

    fzz_data_in: gpio@1090 {
        compatible = "altr,pio-1.0";
        reg = <0xf9001090 0x10>;
        //interrupt-parent = <&intc>;
        //interrupts = <0 18 4>;
        altr,gpio-bank-width = <32>; // 8 bit width
        //altr,interrupt-type = <2>;  // Should be TYPE=EDGE
                                //altr,interrupt_type = <2>;  // Should be DERIVED_TYPE=EDGE
        #gpio-cells = <2>;
        gpio-controller;
        status = "disabled";
    };

    fzz_data_out: gpio@10a0 {
        compatible = "altr,pio-1.0";
        reg = <0xf90010a0 0x8>;
        altr,gpio-bank-width = <32>;
        #gpio-cells = <2>;
        gpio-controller;
        resetvalue = <0>;
        status = "disabled";
    };

    fzz_ctrl_out: gpio@10b0 {
        compatible = "altr,pio-1.0";
        reg = <0xf90010b0 0x8>;
        altr,gpio-bank-width = <8>;
        #gpio-cells = <2>;
        gpio-controller;
        resetvalue = <0>;
        status = "disabled";
    };

    // Deactivated for now as I'm trying off-the-shelf components even dough they're not that well suited
    /*fuzzing_ctrl: gpio@2000 {
      compatible = "itsuzl,fzzctl-1.0";
      reg = <0xf9002000 0x1000>;
      status = "disabled";
    }; */

    /*dipsw_pio: gpio@1070 {
        compatible = "altr,pio-1.0";
```

```
            reg = <0xf9001070 0x10>;
            interrupt-parent = <&intc>;
            interrupts = <0 17 4>;
            altr,gpio-bank-width = <4>;
            altr,interrupt-type = <3>;
                                                altr,interrupt_type = <3>;
            #gpio-cells = <2>;
            gpio-controller;
        }; */

        /*trigger_pio: gpio@1040 {
            compatible = "altr,pio-1.0";
            reg = <0xf9001040 0x20>;
            altr,gpio-bank-width = <4>;
            #gpio-cells = <2>;
            gpio-controller;
            resetvalue = <0>;
        }; */

        soc_leds: leds {
          compatible = "gpio-leds";
          status = "disabled";

          led_fpga0: fpga0 {
              label = "fpga_dbg_led0";
              gpios = <&led_pio 0 1>;
          }; //end fpga0 (led_fpga0)

          led_fpga1: fpga1 {
              label = "fpga_dbg_led1";
              gpios = <&led_pio 1 1>;
          }; //end fpga1 (led_fpga1)

          led_fpga2: fpga2 {
              label = "fpga_dbg_led2";
              gpios = <&led_pio 2 1>;
          }; //end fpga2 (led_fpga2)

          led_fpga3: fpga3 {
              label = "fpga_dbg_led3";
              gpios = <&led_pio 3 1>;
          }; //end fpga3 (led_fpga3)
        };

    };
};
```

Listing 4.3: The device tree source

```
// SPDX-License-Identifier:     GPL-2.0
/*
 * Copyright (C) 2019, Intel Corporation
 */
#include "socfpga_agilex.dtsi"
#include "socfpga_agilex_fuzzer.dtsi"

/ {
    model = "SoCFPGA Agilex SoCDK";

    aliases {
        serial0 = &uart0;
        ethernet0 = &gmac0;
        //ethernet1 = &gmac1;
        //ethernet2 = &gmac2;
    };

    chosen {
        stdout-path = "serial0:115200n8";
    };

	/*leds {
		compatible = "gpio-leds";
		hps0 {
			label = "hps_led0";
			gpios = <&portb 20 GPIO_ACTIVE_HIGH>;
		};

		hps1 {
			label = "hps_led1";
			gpios = <&portb 19 GPIO_ACTIVE_HIGH>;
		};

		hps2 {
			label = "hps_led2";
			gpios = <&portb 21 GPIO_ACTIVE_HIGH>;
		};
	}; */

	memory {
		device_type = "memory";
		/* We expect the bootloader to fill in the reg */
		reg = <0 0 0 0>;
	};

	soc {
		clocks {
			osc1 {
				clock-frequency = <25000000>; //25 MHz
			};
		};
	};
};

&gpio1 {
	status = "okay";
};

&led_pio {
	status = "okay";
};

&soc_leds {
	status = "okay";
};

&fzz_data_in {
	status = "okay";
};

&fzz_data_out {
	status = "okay";
};

&fzz_ctrl_out {
	status = "okay";
};

//&fuzzing_ctrl {
//	status = "disabled"; // for now
//};

&gmac0 {
	status = "okay";
	phy-mode = "rgmii";
	phy-handle = <&phy0>;

	max-frame-size = <9000>;

	mdio0 {
		#address-cells = <1>;
		#size-cells = <0>;
		compatible = "snps,dwmac-mdio";
```

```
      phy0: ethernet-phy@0 {
        reg = <4>;

        txd0-skew-ps = <0>; /* -420ps */
        txd1-skew-ps = <0>; /* -420ps */
        txd2-skew-ps = <0>; /* -420ps */
        txd3-skew-ps = <0>; /* -420ps */
        rxd0-skew-ps = <420>; /* 0ps */
        rxd1-skew-ps = <420>; /* 0ps */
        rxd2-skew-ps = <420>; /* 0ps */
        rxd3-skew-ps = <420>; /* 0ps */
        txen-skew-ps = <0>; /* -420ps */
        txc-skew-ps = <900>; /* 0ps */
        rxdv-skew-ps = <420>; /* 0ps */
        rxc-skew-ps = <1680>; /* 780ps */
      };
    };
};

&mmc {
    status = "okay";
    cap-sd-highspeed;
    broken-cd;
    bus-width = <4>;
};

&uart0 {
    status = "okay";
};

/*&usb0 {
 *  status = "okay";
 *  disable-over-current;
 *};
 */

/*&watchdog0 {
    status = "okay";
}; */ //Testing disabled watchdog

/*&i2c1 {
    status = "okay";
}; */

/*&qspi {
    status = "okay";
    flash@0 {
      #address-cells = <1>;
      #size-cells = <1>;
      compatible = "mt25qu02g";
      reg = <0>;
      spi-max-frequency = <100000000>;

      m25p,fast-read;
      cdns,page-size = <256>;
      cdns,block-size = <16>;
      cdns,read-delay = <1>;
      cdns,tshsl-ns = <50>;
      cdns,tsd2d-ns = <50>;
      cdns,tchsh-ns = <4>;
      cdns,tslch-ns = <4>;

      partitions {
        compatible = "fixed-partitions";
        #address-cells = <1>;
        #size-cells = <1>;

        qspi_boot: partition@0 {
          label = "Boot and fpga data";
          reg = <0x0 0x03FE0000>;
        };

        qspi_rootfs: partition@3FE0000 {
            label = "Root Filesystem - JFFS2";
            reg = <0x03FE0000 0x0C020000>;
        };
    };
  };
}; */
```

Listing 4.4: The axi bridge driver source

```
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/io.h>
#include <linux/miscdevice.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/uaccess.h>

static int fuzzer_probe(struct platform_device *pdev);
static int fuzzer_remove(struct platform_device *pdev);
static ssize_t fuzzer_read(struct file *file, char *buffer, size_t len, loff_t *offset);
static ssize_t fuzzer_write(struct file *file, const char *buffer, size_t len, loff_t *offset);

// Device instance struct
struct custom_fuzzer_dev {
    struct miscdevice miscdev;
    void __iomem *regs;
    u32 fuzzer_state_reg;
    u32 fuzzer_cmd_reg;
    u32 old_rptr;
};

// device tree mapping
static struct of_device_id custom_fuzzer_dt_ids[] = {
    {
        .compatible = "itsuzl,fzzctl-1.0"
    },
    { /* eot */ }
};

// register device mapping
MODULE_DEVICE_TABLE(of, custom_fuzzer_dt_ids);

// platform framework for idb mapping
static struct platform_driver fuzzer_platform = {
    .probe = fuzzer_probe,
    .remove = fuzzer_remove,
    .driver = {
        .name = "fuzzing module bridge driver",
        .owner = THIS_MODULE,
        .of_match_table = custom_fuzzer_dt_ids
    }
};

// misc framework idb mapping
static const struct file_operations axi_bridge_fops = {
    .owner = THIS_MODULE,
    .read = fuzzer_read,
    .write = fuzzer_write
};

static int fuzzer_init(void)
{
    // The linux kernel somehow still uses ANSI C89
    int ret_val = 0;
    pr_info("Initializing fuzzer bridge module\n");
    ret_val = platform_driver_register(&fuzzer_platform);
    if(ret_val != 0) {
        pr_err("platform_driver_register returned error code %d\n", ret_val);
        return ret_val;
    }
    pr_info("AXI bridge init successfull.\n");
    return 0;
}

static int fuzzer_probe(struct platform_device *pdev)
{
    struct custom_fuzzer_dev *dev;

    int ret_val = -EBUSY;
    struct resource *r = 0;

    r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if(r == NULL) {
        pr_err("Unable to map MMIO regs to own address space using IORESOURCE_MEM\n");
        goto bad_exit_return;
    }

    // dev = kzalloc(&pdev->dev, sizeof(struct custom_fuzzer_dev), GFP_KERNEL); // TODO find out why this isn't
         working
    dev = devm_kzalloc(&pdev->dev, sizeof(struct custom_fuzzer_dev), GFP_KERNEL);
    dev->regs = devm_ioremap_resource(&pdev->dev, r);
    if(IS_ERR(dev->regs)) {
      pr_info("Failed to map memory region for AXI bridge:\n");
        return PTR_ERR(dev->regs);
    }

    // Stop the fuzzer in case it is running
    dev->fuzzer_cmd_reg = 0x0000;
    iowrite32(dev->fuzzer_cmd_reg, dev->regs);

    dev->miscdev.minor = MISC_DYNAMIC_MINOR;
    dev->miscdev.name = "fuzzeraxibridge";
    dev->miscdev.fops = &axi_bridge_fops;

    ret_val = misc_register(&dev->miscdev);
    if(ret_val != 0) {
        pr_info("Couldn't register misc device :(");
        return ret_val;
    }

    // register device instance with platform driver
    platform_set_drvdata(pdev, (void*)dev);

    pr_info("Successfully registered new fuzzer instance\n");
    return 0;
}

static ssize_t fuzzer_read(struct file *file, char *buffer, size_t len, loff_t *offset)
{
    int success = 0;
    struct custom_fuzzer_dev *dev = container_of(file->private_data, struct custom_fuzzer_dev, miscdev);

    // return status register
    success = copy_to_user(buffer, &dev->fuzzer_state_reg, sizeof(dev->fuzzer_state_reg));
    if(success != 0) {
      pr_info("Failed to fetch fuzzer status register content\n");
      return -EFAULT;
    }

    u32 new_rptr = (ioread32(dev->fuzzer_state_reg, dev->regs) & 00111111111111000000000000000000b) >> 20;
    while (new_rptr != &dev->old_rptr) {
      int i;
      for(i = 0; i < 4; i++) {
    success = copy_to_user(buffer, &dev->regs + &dev->old_rptr + i, sizeof(u32));
      }
     &dev->old_rptr = &dev->old_rptr + 4;
    }
    if(success != 0) {
        pr_info("There was an updated ring pointer position but fetching it failed.\n");
        return -EFAULT;
    }
    return 0;
}

static ssize_t fuzzer_write(struct file *file, const char *buffer, size_t len, loff_t *offset)
{
    int success = 0;
    struct custom_fuzzer_dev *dev = container_of(file->private_data, struct custom_fuzzer_dev, miscdev);
    success = copy_from_user(&dev->fuzzer_cmd_reg, buffer, sizeof(dev->fuzzer_cmd_reg));
    if(success != 0) {
        pr_info("Failed to read fuzzer command from process\n");
        return -EFAULT;
    } else {
        iowrite32(dev->fuzzer_cmd_reg, dev->regs);
    }

    // TODO only ack commands really send
    return len;
}

static int fuzzer_remove(struct platform_device *pdev)
{
    struct custom_fuzzer_dev *dev = (struct custom_fuzzer_dev*) platform_get_drvdata(pdev);
    // Stop the fuzzer
    iowrite32(0x0000, dev->regs);
    misc_deregister(&dev->miscdev);
    pr_info("AXI bridge deregistered\n");
    return 0;
}

static void fuzzer_exit(void)
{
    platform_driver_unregister(&fuzzer_platform);
    pr_info("Fuzzer AXI bridge successfully unregistered\n");
}

// Tell the kernel which functions are the initialization and exit functions
module_init(fuzzer_init);
module_exit(fuzzer_exit);

// Define information about this kernel module
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Leon Dietrich <leon.dietrich@student.uni-luebeck.de>");
MODULE_DESCRIPTION("Controls an FPGA based PCIe Fuzzer");
MODULE_VERSION("0.1");
```

3 Additional simulation output listings

Listing 4.5: End of SymbiYosys output of AXI bridge verification

```
SBY 19:13:37 [tb_axibridge_bmc] Removing directory 'tb_axibridge_bmc'.
SBY 19:13:39 [tb_axibridge_cover] Removing directory 'tb_axibridge_cover'.
SBY 19:13:47 [tb_axibridge_prove] Removing directory 'tb_axibridge_prove'.
SBY 19:13:47 [tb_axibridge_prove] Copy 'tb_axibridge.vhd' to 'tb_axibridge_prove/src/tb_axibridge.vhd'.
SBY 19:13:47 [tb_axibridge_prove] Copy 'rcon.mem' to 'tb_axibridge_prove/src/rcon.mem'.
SBY 19:13:47 [tb_axibridge_prove] engine_0: smtbmc
SBY 19:13:47 [tb_axibridge_prove] base: starting process "cd tb_axibridge_prove/src; yosys -ql ../model/design.log ../model/design.ys"
SBY 19:13:47 [tb_axibridge_prove] base: finished (returncode=0)
SBY 19:13:47 [tb_axibridge_prove] smt2: starting process "cd tb_axibridge_prove/model; yosys -ql design_smt2.log design_smt2.ys"
SBY 19:13:47 [tb_axibridge_prove] smt2: finished (returncode=0)
SBY 19:13:47 [tb_axibridge_prove] engine_0.basecase: starting process "cd tb_axibridge_prove; yosys-smtbmc --presat --unroll --noprogress -t 4  --append 0 --dump-vcd engine_0/trace.vcd --dump-vlogtb engine_0/trace_tb.v --dump-smtc engine_0/trace.smtc model/design_smt2.smt2"
SBY 19:13:47 [tb_axibridge_prove] engine_0.induction: starting process "cd tb_axibridge_prove; yosys-smtbmc --presat --unroll -i --noprogress -t 4  --append 0 --dump-vcd engine_0/trace_induct.vcd --dump-vlogtb engine_0/trace_induct_tb.v --dump-smtc engine_0/trace_induct.smtc model/design_smt2.smt2"
SBY 19:13:48 [tb_axibridge_prove] engine_0.basecase: ##   0:00:00  Solver: yices
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: ##   0:00:00  Solver: yices
SBY 19:13:48 [tb_axibridge_prove] engine_0.basecase: ##   0:00:00  Checking assumptions in step 0..
SBY 19:13:48 [tb_axibridge_prove] engine_0.basecase: ##   0:00:00  Checking assertions in step 0..
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: ##   0:00:00  Trying induction in step 4..
SBY 19:13:48 [tb_axibridge_prove] engine_0.basecase: ##   0:00:00  Checking assumptions in step 1..
SBY 19:13:48 [tb_axibridge_prove] engine_0.basecase: ##   0:00:00  Checking assertions in step 1..
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: ##   0:00:00  Trying induction in step 3..
SBY 19:13:48 [tb_axibridge_prove] engine_0.basecase: ##   0:00:00  Checking assumptions in step 2..
SBY 19:13:48 [tb_axibridge_prove] engine_0.basecase: ##   0:00:00  Checking assertions in step 2..
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: ##   0:00:00  Trying induction in step 2..
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: ##   0:00:00  Trying induction in step 1..
SBY 19:13:48 [tb_axibridge_prove] engine_0.basecase: ##   0:00:00  Checking assumptions in step 3..
SBY 19:13:48 [tb_axibridge_prove] engine_0.basecase: ##   0:00:00  Checking assertions in step 3..
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: ##   0:00:00  Trying induction in step 0..
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: ##   0:00:00  Temporal induction succeeded!
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: ##   0:00:00  Writing trace to VCD file: engine_0/trace_induct.vcd
SBY 19:13:48 [tb_axibridge_prove] engine_0.basecase: ##   0:00:00  Status: passed
SBY 19:13:48 [tb_axibridge_prove] engine_0.basecase: finished (returncode=0)
SBY 19:13:48 [tb_axibridge_prove] engine_0: Status returned by engine for basecase: pass
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: ##   0:00:00  Writing trace to Verilog testbench: engine_0/trace_induct_tb.vhd
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: ##   0:00:00  Writing trace to constraints file: engine_0/trace_induct.smtc
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: ##   0:00:00  Status: passed
SBY 19:13:48 [tb_axibridge_prove] engine_0.induction: finished (returncode=1)
SBY 19:13:48 [tb_axibridge_prove] engine_0: Status returned by engine for induction: pass
SBY 19:13:48 [tb_axibridge_prove] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:01 (1)
SBY 19:13:48 [tb_axibridge_prove] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:01 (1)
SBY 19:13:48 [tb_axibridge_prove] summary: engine_0 (smtbmc) returned pass for basecase
SBY 19:13:48 [tb_axibridge_prove] summary: engine_0 (smtbmc) returned pass for induction
SBY 19:13:48 [tb_axibridge_prove] DONE (UNKNOWN, rc=4)
```

Listing 4.6: Output of the P-Tile verification tool

```
Starting Altera IP Simulation helper.
Copyright (C) Altera Corporation 2000-2003
Starting Simulation of file ../../src/ip/p_tile.ip
=================================== LOG ===================================
[ INFO ] Reset was triggered.
[ WARN ] Invalid power state detected.
[ WARN ] Entering D4
[ WARN ] Entering D4
[ WARN ] Entering D4
[ WARN ] Entering D4
[ WARN ] Entering D4
[ WARN ] Entering D4
[ INFO ] Powerstate reset
[ INFO ] Sending REASSOC
[ INFO ] Register 0x00104194 was configured
[ INFO ] Register 0x00000038 was configured
[ WARN ] TLP error management was set to manual mode.
[ INFO ] Register 0x00000039 was configured
[ INFO ] Register 0x00104068 was configured
[ INFO ] Enumeration succeeded. Device ID: 0x80865227@0:0:0:0
[ WARN ] SR-IOV support was enabled through reconfiguration interface but wasn't configured in IP parameter list.
```

4 Tables

Table 1: UBoot command prompt reference

Command	Behaviour
`base`	Manage memory base offsets
`bdinfo`	debug hardware offsets that are used to fill empty fiels in the passed DTB.
`blkcache`	`Diagnose \Gls{mmc} blocks`
`boot[_,d,efi,elf,i,m,p,vx,z]`	boot different image formats
`bridge`	configure `\Gls{sdm}` bridge
`chpart`	change active boot partition
`clocks`	Display clock properties
`cmp`	compare memory contents at `address`
`coninfo`	`UART` debugging
`cp`	Copy `length` bytes of memory from `address1` to `address2`
`crc32`	Various checksum tools
`dcache`	Data cache setup management tools (Processor L1 data cache)
`dhcp`	Obtain an IP Address using dhcp, then try to boot using DHCP Option 66
`dm`	UBoot driver debugging tools
`echo`	Output things to the command line
`editenv`	Manipulate UBoot envirnoment variables in a high level fashion
`env`	Set or reset environment variables
`erase`	Delete data partitions
`exit`	exit scripts or return from functions
`ext4load`	load files from specified Ext4 paths to the specified memory location
`ext4ls`	list files in the specified Ext4 path
`ext4size`	Obtain the size of a specified Ext4 file
`false|true`	Boolean constants for evaluations in scripts
`fatinfo`	print debugging information about FAT file systems
`fatload`	load files from specified FAT path to the specified memory location

Table 1: UBoot command prompt reference

Command	Behaviour
`fatls`	list files in specified FAT path
`fatmkdir`	create a new directory in a FAT partition
`fatrm`	remove a file from a FAT partition
`fatsize`	Obtain a file size from a FAT partition
`fatwrite`	Write a file to a FAT partition
`fdt`	Manipulate loaded flattened device trees
`flinfo`	print flash memory debug informations
`fpga`	Release reset of FPGA configuration in HPS-first boot mode
`fstype`	Guess the file system type of a partition
`go <address>`	Set the program counter to `address`
`gpio`	Manipulate GPIO pins that are directly connected to the HPS
`gzwrite`	Manipulate gz compressed archives
`i2c`	Interact with devices connected to a I²C bus
`icache`	Enable or disable processor L1 instruction cache
`iminfo`	decode Linux image header information
`imxtract`	Extract data from a zipped linux archive
`itest`	Integer comparison tools
`ln`	Manage symbolic links in file systems
`lzmadec`	decompress a memory region using the LZMA algorithm
`md <address> [bytes]`	Display decoded memory content from `address`, up to `bytes` total.
`mdio`	manage MDIO controllers for attached ethernet interfaces
`mii`	manage MII controllers for attached ethernet interfaces
`mm`	interactive memory content editor
`mmc`	manage attached MMC states
`mmcinfo`	print out MMC device debugging information
`mtd`	Manage MTD compatible NAND flash
`mtdparts`	Manage flash partitions

Table 1: UBoot command prompt reference

Command	Behaviour
`mtest`	Test defined memory regions
`mw`	fill memory with predefined patterns
`nfs`	Connect to a NFS server and boot the specified Linux image from it
`nm`	alter the content of memory at the specified location
`panic`	panic the SSBL
`ping`	send ICMP ping responses and wait for an answer
`pr`	FPGA partial reconfiguration utilities
`printenv`	print environment variables
`protect`	alter flash memory write protection
`random`	fill a specified memory location with random data
`reset`	issue reset signals for various hardware components
`rsu`	perform remote system upgrades of the *`\Gls{sdm}`* and FPGA
`run`	execute commands specified in environment variable
`save`	save content from memory to a file on permant storage
`setenv`	set an environment variable
`setexpr`	evaluate an expression and set the result as the value of the specified environment variable
`sf`	SPI flash management tools
`showvar`	print content of local variables
`size`	determine the size of a continues object in memory or mapped file
`sleep <seconds>`	pause the CPU for the *`seconds`* specified
`source`	execute spcified memory location as UBoot script
`sspi`	manage connected SPI devices
`test`	UNIX like test evaluation tool
`tftpboot`	boot an image from a TFTP share
`unlz4`	decompress a memory region using the LZ4 algorithm

Table 1: UBoot command prompt reference

Command	Behaviour
`unzip`	decompress a memory region using the ZIP algorithm
`usb`	manage devices connected via USB
`usbboot`	boot an image from an attached USB mass storage device
`version`	print information about the used compiler and compile time

Table 2: Data to Initialize a P-Tile channel

Address	Content
0x104194	*"XXXXXXXXXXXX11010110010100000000"h*
0x000038 *0x000039*	*"XXXXXXXXXXXXXXXXXXXXXXXX00000000"h* *"00000000000000000000000000000000"h*
0x104068	*"XXXX00000000110XXXXXXXXXXXXXXX11"h*

Table 3: Relevant TLP Prefix types

Type	Domain	Label	Purpose
MR-IOV	Local	*"0000"h*	Multi Root IO Virtualization Control Information
RMTRewrite	Local	*"0010"h*	Unknown to the author – Ignoring it did not do anything yet.
VendorPrefixL0	Local	*"1110"h*	Vendor specific prefix type.
VendorPrefixL1	Local	*"1111"h*	Vendor specific prefix type.
ExtTPH	End to End	*"0000"h*	Mark header as extended
PASID	End to End	*"0001"h*	PCIe PASID management
VendorPrefixE0	End to End	*"1110"h*	Vendor specific prefix type.
VendorPrefixE1	End to End	*"1111"h*	Vendor specific prefix type.
All other combinations are reserved for future use.			

5 The userland control script

Listing 4.7: The userland control script

```
#! /usr/bin/env python3

import json

class FuzzerState():

    def __init__(self, state_reg):
        self.init_done = bool(state_reg[3] & 00000001b)
        self.fuzzer_running = bool(state_reg[3] & 00000010b)
        self.ring_ptr = ((state_reg[3] & 0b11111100) << 6) | (state_reg[2] & 0b00000111)
        self.buffer_stalled = bool(state_reg[2] & 0b00001000)
        self.fuzzer_in_error_state = bool(state_reg[2] & 0b00010000)


previours_messages = []

def encode_as_json(arr, index):
    if arr in previous_messages:
        return False
    else:
        previous_messages.append(arr)
    d = {}
    d["index"] = index
    d["test_case"] = arr
    return json.dumps(d)


with open("/dev/fuzzer", "rwb") as ff:
    # stop the fuzzer -- just in case it is running
    state: FuzzerState = FuzzerState(ff.read(4))
    ff.write("STOP".encode("ASCII"))
    # configure start TLP
    start_address = [
            bytearray([0,0,0,0]),
            bytearray([0,0,0,0]),
            bytearray([0,0,0,0]),
            bytearray([0,0,0,0])]
    for sequence in start_address:
        ff.write(sequence))
    ff.write("START".encode('ASCII'))
    state: FuzzerState = FuzzerState(ff.read(4))
    print("Started Fuzzer")
    last_pointer = state.ring_ptr
    i = 0
    while state.fuzzer_running:
        state = FuzzerState(ff.read(4))
        test_case_stub = []
        while state.ring_ptr != last_pointer:
            last_pointer += 4
            if last_pointer > 512:
                last_pointer -= 512
            test_case_stub.append(ff.read(4))
        if len(test_case_stub > 0):
            js_string = encode_as_json(test_case_stub, i)
            if js_string:
                print("MESSAGE ", js_string)
            i += 1
    print("Finished.")
```

6 The FPGA HDL overview

This is an overview of the most important parts of the directory structure. As this list does not fit on a single page it is continued on the next one.

```
pcie_fuzzer/
├─hps_firmware
│ ├─build.bash ........................ Buildscript for userland and kernel
│ ├─make_sdimage.py ................... Partition helper script from Intel
│ ├─createImage.bash ......... Script to generate an image after userland compiling
│ └─ovrfiles .......................... Directory containig patch files
├─ip_upgrade_port_diff_reports ..... Quartus IP and QSYS management reports
├─pcie_fuzzer.qpf ..................... Quartus project file
├─pcie_fuzzer.qsf ..................... Constraints file
├─pcie_fuzzer.sdc ..................... Clocking information database
├─signal_tap.stp ...................... Signal tap configuration file
└─simulation
  ├─aldec ............................. aldec library setup
  ├─cadence ........................... cadence library setup
  ├─common ............................ Setup of common simulation tools
  │ ├─modelsim_files.tcl
  │ ├─ncsim_files.tcl
  │ ├─riviera_files.tcl
  │ ├─vcs_files.tcl
  │ ├─vcsmx_files.tcl
  │ └─xcelium_files.tcl
  ├─mentor ............................ Actaul simulation scripts
  │ ├─libraries
  │ ├─mentor_tb_axi_slave.do
  │ ├─mentor_tb_skid_buffer.do
  │ ├─mentor_tb_tlp_gen.do
  │ ├─modelsim.ini
  │ ├─msim_setup.tcl
  │ └─vsim.wlf
  ├─src ............................... Testbenches, Mocks and SymbiYosys files
  ├─synopsys
  ├─transcript
  └─xcelium
```

```
pcie_fuzzer/
├── src
│   ├── ip
│   │   ├── output_memory
│   │   ├── output_memory.ip .................................... Agilex BRAM IP
│   │   ├── PLL_100MHz_Locked
│   │   ├── PLL_100MHz_Locked.ip
│   │   ├── p_tile
│   │   ├── p_tile.ip
│   │   ├── reset_release
│   │   └── reset_release.ip ........ Required to schedule initialization after cold reset
│   ├── qsys
│   │   ├── ip
│   │   └── platform_designer_hps ................................. The HPS SoC
│   └── vhdl
│       ├── axi_lite_to_register_slave.vhd..................... The AXI bridge
│       ├── clock_divider.vhd
│       ├── domain_boundry_tools
│       │   ├── fast_to_slow_clk_OBUF.vhd
│       │   ├── skid_buffer.vhd
│       │   ├── slow_to_fast_clk_IBUF.vhd
│       │   └── slow_to_fast_edge_forward.vhd
│       ├── fuzzing_controller.vhd............................. The control FSM
│       ├── mux.vhd
│       ├── pcie_message_generator.vhd
│       ├── pcie_pwr_manager.vhd
│       ├── pci_port_handler.vhd
│       ├── pio_to_ctrl_reg_converter.vhd
│       ├── ptile_state_controller.vhd
│       ├── rslatch.vhd
│       ├── top.vhd ................................. The top level module of the design
│       └── tx_tlp_converter.vhd
├── synth_dumps
├── top_assignment_defaults.qdf
└── top.qws
```

Acronyms

ACPI	is short for Advanced Configuration and Power Interface and defines a set of interfaces modern x86 based computers provide for power and device management such as setting the screen brightness or putting the computer to sleep. It is often criticized for its ridicules complexity and the source of many issues as most software and hardware vendors fail to implement it correctly. 37
ALU	Arithmetic Logic Unit. 10
ALU	An arithmetic logic unit is a circuit that is capable of performing basic arithmetic operations such as full additions or logical comparisons. 10, 43
ATF	Arm Trusted Firmware. 21, 22
AXI	Advanced eXtensible Interface Bus. 29, 31, 34
AXI	AXI is an interconnect specified within the AMBA specifications by Arm. It is used to connect various modules on a chip by utilizing DMA. Due to it being usable royalty free and being relatively fast it became quite popular. 16, 29–31, 34, 47
BRAM	stands for Block Random Access Memory and is a piece of memory on the fabric of an FPGA or CPLD one can use in order to store data. Usually those blocks are somewhat flexible in terms of port configuration and allow one to store multiple KB without building registers from other logic blocks. 10, 21, 43
BUS	A Bus is a simple interconnect between two or more circuits enabling them to communicate with each other. Its primary characteristic is that it consists of multiple wires that are connected to every member and only one member can write on those wires at a time whereas all others can read at that time. 1, 8, 20, 42
CCU	A Cache Coherence Unit is a piece of hardware that ensures that when a value in memory is being modified all caches currently storing this value de-validate it in order to re-fetch it. This way all hardware that is using this value always has the correct version of it. 21
CPLD	Complex Programmable Logic Device. 9

CPLD A CPLD or simply PLD is a computer chip which also allows the user to specify the desired circuit but on a much smaller scale. A CPLD only allows a very limited number of sequential circuits and is usually not reconfigurable. 10

CPU A Central Processing Unit is a type of processor that usually acts as the main processor within a system. It consists of one or multiple processing cores that are usually quite fast at general purpose computations and management but need to rely on more specialized accelerators in order to efficiently process large amount of equable data. 1, 4, 24, 26, 28, 45

DHCP the Dynamic Host Configuration Protocol is a computer network protocol supplying computers in a network with various information on how deal with the network. Such information commonly contains, but is not limited to, the assigned IP address and host name, packet routing information, domain name resolution servers, boot image servers, access policies, etc. . 22

differential signalling Differential signalling is a technique where every bit is encoded on two lines that have inverted polarity. At the receiving end the difference between both lanes determines the bit that was transmitted. A positive polarity means a logical 1 and a negative polarity means a logical 0. A disturbance affects both lanes approximately equally if both lanes are placed close to each other and thus the difference between both signals stays the same. This way one can transmit at very high speeds without having tremendous signal errors. 17, 35

DIP switch A DIP switch (where DIP stands for dual inline package) is a (or series of) switches within a single package. They are commonly used to configure electronics. 17

DMA DMA stands for direct memory access. It is a technique where two communication partners do not schedule a data transmission with each other. Instead they each give each other an address to a buffer within their private memory region and read and write to that region directly. The upside of using this technique is that it is really fast. The downside is that one needs to implement numerous security measures in order to make sure that the other party only accesses memory it is allowed to, as memory is known to contain the most sensitive information (file and process trees, cryptographic keys, open and recently closed file contents, pictures of your cat, banking account credentials, etc.) a system has. 4, 5, 30

DSP A digital signal processor (in the context of an FPGA) is a hard logic block on the fabric of an FPGA capable of performing various complex operations such as FFT, floating point operations (including multiplication), big integer operations or in some cases even cryptographic functions such as single AES rounds. The purpose of these blocks are based on the common need of such features in most designs and allowing them to be packed more densely on the chip and operating at a much higher frequency. 10, 43

DUT A design under test is a piece of hardware (or sometimes software) design that is being verified for proper working. Such verification is often performed by simulation and comparison of output, formal verification, or simply by running it and observing whether or not it works . 7, 8, 14, 41, 47

E-Tile The E-Tile is a piece of hard IP on Intel Agilex F-Series FPGAs that provides high speed Ethernet (2 times 40GBit/s) connectivity. It is superseded by the F-Tile providing two transceivers with 400GBit/s throughput in each direction. 17

EDA Electronic Design Automation. 17, 34, 47

FCPS Fuzz Cases per Second. 6

FFT An Fast Fourier Transformation (FFT) is a mathematical operation outputting the composing frequencies and their share from a given input signal. 10

FIFO A FIFO (short for First In First Out) buffer is a type of buffer, commonly found in hardware, designed to queue packets in a way that the packets are read in the order they came in. 29

FOSS Free and Open Source Software is software that is provided to one with the free rights of usage in any kind of said software, modification of source code (which needs to be provided), learning from said source code and providing the source code to others under the same conditions. 10

FPGA Field Programmable Gate Array. iii, v, ix, 9–12, 14, 16–19, 21, 29, 35, 43, 46, 47, 67, 68

FPGA An FPGA (Field Programmable Gate Array) is a device (usually one or a set of computer chips) which allows to user to dynamically specify the circuit it reassembles. 4, 14, 29–31

FSBL A First Section Boot Loader (commonly referred to as "Firmware") is a short piece of software that is run prior to the actual boot loader and ensures that the actual bootloader meets a hardware setup that it can operate on. 20, 21

FSM Finite State Machine. 15

GPIO stands for General Purpose Input Output and describes hardware that can be utilized for various functionality in a flexible manner at runtime. 35, 67

GPU A Graphics Processing Unit is an accelerator that historically speaking was designed to process graphical tasks. Doing so they require huge amount of not that powerful stream lining processors that perform these tasks in parallel. At some point people discovered that one can not only process graphics data on such devices but any data that massively scales to parallelism. Since then they are freely programmable and used for all kind of processing. 4, 27

hard IP Hard IP are pieces of hardware that are already embedded on the FPGA outside the fabric. They are called hard because they are not implemented in soft logic and thus not modifiable by the end user. 10, 19

HDL hardware description language. 18, 35

heap The heap is the part of a processes memory that is used to store constant or large objects. 26

HPS The Hard Processor System is a companion processor composed of four modified Arm A53 processor cores and some periphery. As most FPGA designs require some form of processor for control one might as well embed one using fixed hardware on the FPGA. The Intel HPS is such a fixed processor. 15, 16, 18–21, 25, 28–32, 40, 43, 47, 67, 72

hypervisor A hypervisor is a computer (software) that hosts one or multiple virtual machines. 12

interrupt An interrupt is a signal lane that causes, when triggered, the CPU to stop what it is doing and jumping to a predefined address where the corresponding ISR is being held. 27

IP Intellectual Property. 10, 18, 21, 36

IP Intellectual Property in this context referees to foreign hardware pieces, usable under a license. 14, 15, 21, 30, 31

IRQ An Interrupt Request is the actual event of an interrupt being triggered. 26, 27

ISR An Interrupt Service Routine is a piece of code that is being jumped to when a corresponding IRQ occurs. Ideally this code handles what ever needs to be done when the interrupt is being triggered. 27

I²C I²C (which is short for inter integrated circuit, but no one actually says that) is an interconnect between different computer chips commonly found on PCBs. While it does not reach high transmission speeds, it is quite simple to implement and very robust and hence enjoys constant popularity. 19, 30, 67

JSON JavaScript Object Notation is a data format that was originally used for transmitting data in the context of websites. Due to its good human readability as well as good parsing properties it gained great popularity far beyond web pages and is one of the most common data exchange formats by now. 28

JTAG JTAG refers to Joint Test Action Group and is a debugging protocol where the outputs and inputs of device registers are chained together in order to read or write arbitrary data from device within that chain. This protocol is often used to program various devices. 16, 17

kernel A kernel is the part of an operating system that deals with all hardware interaction and schedules the processes to be run as well as providing interfaces to commonly used features such as networking or file systems. 16, 24, 29

LUT Lookup Table. 10

LUT A lookup table is a device which stores for each possible input sequence an output sequence. They can be used implement basic combinatorial logic statements. 10, 43

LVDS Low Voltage Differential Signalling. 18

meta layer A meta layer is a set of configuration files that specify how a specific part of an operating system is supposed to be built. 25

MFVC MFVC is short for Multi-Function Virtual Channel and can be described as feature where an enumerated PCIe devices is capable of changing its supported features (functions, capabilities) on the go. 45

MMC MMC (in this context) stands for Multi Media Card and is together with the NAND Flash protocol the most common way to provide storage to embedded devices. Applicable storage devices come in the form of SD-cards or are soldered to the board (eMMC, the e stands for embedded). 19

MMU An MMU (short for Memory Management Unit is a device that at least translates memory addresses between multiple virtual address spaces and the physical address space of the attached device. It is quite common for an MMU to also check the access privileges of the requesting device to see if the device or process is allowed to read or write to that particular address. Since MMUs tend to be very complex and need to be really fast in order to not slow down communications they always are an interesting attack target. 1, 4, 5, 11, 20, 21, 30, 45

MR-IOV MR-IOV stands for Multi Root - IO Virtualization and is similar to SR-IOV except that one feature might be shared across multiple virtual machines and there might be multiple virtual root complexes. This feature can also be used to communicate to a PCIe device within a different a different root complex domain possibly on a remote machine. 69

mux A Mux is a piece of hardware that is capable of selecting one out of many input signals and passing it through to the output based on a given signal selection input. 10, 21, 43

NIC Network Interface Card. 8

NIC A network interface card is a device that provides network (most commonly Ethernet) connectivity to a computer. 1

P-Tile Intel FPGA are equipped with certain hard IP they call Tile. The P-Tile is one that is capable of providing PCIe connectivity. Despite the clever naming, its successor IP for PCIe 5.0 with CXL offloading features is named R-Tile. 14, 15, 17, 35–37, 41, 42, 65, 76

PASID stands for Process Address Space ID and is a feature that allows a PCIe network to use multiple overlapping 64 bit address spaces. 69

PC The Program Counter is a register within a processor that stores the current instruction address. When an instruction is due for being executed the instruction will be loaded from the address specified by the program counter. After doing so the PC is incremented by the instructions length in order to point to the next instruction. 21, 26

PCB A printed circuit board is peace of compound material (usually glass fibre but other materials are used as well) containing computer chips as well as other electrical components and circuit traces connecting them. 1, 30

PCIe Peripheral Component Interconnect - Express. iii, v, 1, 2, 4–6, 8, 9, 12–15, 24, 27, 35–37, 39, 41–43, 45–47, 69

PCIe bifurcation is the technology to split multiple PCIe lanes that are bound together to form a logical link into multiple links in order to connect more devices to the same port. For example on might split an x16 link into four x4 links in order to connect four devices to this link or one might split an x4 link into four single lane links. One can use any divisor as long as it is a power of two. This technique is often used to connect more devices within a server or to need a single port for multiple lower speed peripherals within an embedded device in order to save costs. 35

PCIe enumeration PCIe enumeration refers to the process where a unique address is promoted to a newly connected device. In the past this was only done at the upstart of the computer but as soon as PCIe became hot plug capable it is being done every time a new device gets connected. 17

Phy A phy is a circuit or chip that is responsible for translating digital signals (which are usually of short reach) into longer range signals for usage on a physical transmission medium. 14, 17

PLL Phase Locked Loop. 37

PLL A phase locked loop is a device capable of synthesizing based on a given frequency A a secondary frequency B. The ratio of both signal frequencies is constant and user defined. The phase shift between A and B is also constant and locked to a defined target. Such a device is useful whenever one needs to cross clock boundaries. 10, 14, 43

QEMU which is short for Quick EMUlator is an open source simulator for computer architectures one can use to test operating system features. 47

register A register is a piece of hardware that stores a value which it can alter. Such alternation may may occur in the form of overwriting the value with a new one, performing arithmetic operations on it or performing logical operations. Other operations are conceivable as well. 1, 2, 13, 21, 26, 29, 34, 37, 43

ring buffer A ring buffer is buffer consisting of a shared memory and two pointer registers. The first register is being incremented every time data is being inserted into the buffer. The second register is being incremented every time data is being read from the memory. Every now and then these registers overflow causing them to start at address 0 again thus the name. In order to prevent data loss at the event of one side being faster than the other one has to stall under the condition of both registers containing the same content. 29, 43

SDM The Secure Device Manager is a piece of hardware on higher tier FPGAs from Intel. It is, among other things, responsible for configuring the FPGA, bootstrapping the HPS, performing bitstream authentication and decryption as well as resetting parts of the hardware that either became unresponsive or requested a reset. 17, 20–22, 47

sed sed is a streamlining editor found on most Linux or UNIX based operating Systems. 25

SIMD stands for Single Instruction Multiple Data and means instructions that apply the same operation to multiple data entries at the same time. 26

skid buffer A skid buffer is a special type of pipelined FIFO buffer negotiating between two devices on different clock domains about sending data, stalling the sending one if the receiver is not ready and notifying the receiver about data integrity thus preventing data loss or corruption. A special property of a skid buffer is that it does not introduce further latency if both devices can keep up with the transmission speed. 37

SoC A System On a Chip is complete computer equipped with all required device to perform normal operation within a single computer chip package or a single region of a larger chip. 14–16, 18, 20, 21, 24, 47, 72

SPI stands for Serial Peripheral Interface — A popular external chip interconnect. 31

SR-IOV SR-IOV stands for Single Root - IO Virtualization and is a technique, were a single PCIe device can be forwarded to multiple virtual machines without involving a special emulation driver at the hypervisor. 13, 35, 45

SSBL The Second Section Boot Loader (commonly referred to as "Bootloader") is a piece of software that is started by the FSBL and in turn is responsible for further hardware setup and booting (a.k.a. starting) an operating system or bare metal application. It is usually the first piece of software that enables user interaction. 22, 68

SSH Secure SHell is a network protocol were one can control and transmit data to and from a remote computer. It provides strong encryption and provides decent data integrity checks. 15

stack The stack is the part of a processes memory that is being used to store small temporary objects. 26

TLP Transaction Layer Packet. iii, v, 2, 5, 11–14, 16, 36, 38–43, 46, 47, 75

UART UART, which is short for Universal Asynchronous Receiver Transmitter is a common type of serial interface. It is commonly used to connect to serial consoles. 15, 19, 22, 24

VM A virtual machine is a virtualized computer, running, possibly together with a lot of other VMs, on a real computer or other VM. Virtual machines are commonly used to share compute resources of a physical machine as it is quite seldom that a single service would utilize the resources of the computer it is running on all the time. That way one can provide large amount of processing power to a single service and utilize the physical hardware one got in a more efficient manner resulting in less required hardware and thus lower cost. In a cloud environment it is crucial that different virtual machines ca not access each others data. 13

wave form A wave form is a set of output signals of a hardware module. They consist of their voltage levels plotted over the time of observation. 33

List of Figures

List of Tables

Listings

References

[DPM11] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. What if you can't trust your network card? In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings*, volume 6961 of *Lecture Notes in Computer Science*, pages 378–397. Springer, 2011. doi:10.1007/978-3-642-23644-0_20.

[DPVL10] Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can you still trust your network card. *CanSecWest/core10*, pages 24–26, 2010.

[GH06] David Gibson and Benjamin Herrenschmidt. Device trees everywhere. *OzLabs, IBM Linux Technology Center*, 2006.

[GS17] Neville Grech and Yannis Smaragdakis. P/Taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):102:1–102:28, 2017. doi:10.1145/3133926.

[GWY+15] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In Bhavani M. Thuraisingham, XiaoFeng Wang, and Vinod Yegneswaran, editors, *Security and Privacy in Communication Networks - 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, volume 164 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 330–347. Springer, 2015. doi:10.1007/978-3-319-28865-9_18.

[Han17] Julia Hansbrough. Fuzzing pci express: security in plaintext. Technical report, Google Security Team, 2017. URL: https://cloud.google.com/blog/products/gcp/fuzzing-pci-express-security-in-plaintext.

[KLR+20] Mohamed Amine Khelif, Jordane Lorandel, Olivier Romain, Matthieu Regnery, Denis Baheux, and Guillaume Barbu. Toward a hardware man-in-the-middle attack on pcie bus. *Microprocess. Microsystems*, 77:103198, 2020. doi:10.1016/j.micpro.2020.103198.

[KNMS20] Yohei Kuga, Ryo Nakamura, Takeshi Matsuya, and Yuji Sekiya. NetTLP: A development platform for PCIe devices in software interacting with hardware. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 141–155. USENIX Association, 2020. URL: `https://www.usenix.org/conference/nsdi20/presentation/kuga`.

[Law14] Jason Lawley. Understanding performance of PCI Express systems. *WP350 (v1. 2). Xilinx*, 97, 2014.

[LDL14] Bing Li, Yan Ding, and Yong Liu. Circuit design of PCI Express retry mechanisms. *WSEAS Transactions on Circuits and Systems*, 2014.

[LPJ$^+$18] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Trans. Reliab.*, 67(3):1199–1218, 2018. `doi:10.1109/TR.2018.2834476`.

[LZZ18] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecur.*, 1(1):6, 2018. `doi:10.1186/s42400-018-0002-y`.

[MANK16] Benoît Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. Bypassing IOMMU protection against I/O attacks. In *2016 Seventh Latin-American Symposium on Dependable Computing, LADC 2016, Cali, Colombia, October 19-21, 2016*, pages 145–150. IEEE Computer Society, 2016. `doi:10.1109/LADC.2016.31`.

[MHH$^+$19] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. `doi:10.1109/TSE.2019.2946563`.

[MRG$^+$19] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. `doi:10.14722/ndss.2019.23194`.

[PCI06] PCI-SIG. *PCI Express Base Specification*, 2006. revision 2.0.

[Pra95] Vaughan R. Pratt. Anatomy of the pentium bug. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Den-*

mark, May 22-26, 1995, Proceedings, volume 915 of *Lecture Notes in Computer Science*, pages 97–107. Springer, 1995. doi:10.1007/3-540-59293-8_189.

[Ruy20] Björn Ruytenberg. Breaking Thunderbolt Protocol Security: Vulnerability Report, 2020. Public version. URL: https://thunderspy.io/assets/docs/breaking-thunderbolt-security-bjorn-ruytenberg-20200417.pdf.

[SAA$^+$20] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: high-dimensional hypervisor fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. URL: https://www.ndss-symposium.org/ndss-paper/hyper-cube-high-dimensional-hypervisor-fuzzing/.

[Sha87] Fred R. Shapiro. Etymology of the computer bug: History and folklore. *American Speech*, 62(4):376–378, 1987.

[SHK$^+$20] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2541–2557. USENIX Association, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/song.

[SHW$^+$19] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. Yosys+nextpnr: An open source framework from Verilog to bitstream for commercial FPGAs. In *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*, pages 1–4. IEEE, 2019. doi:10.1109/FCCM.2019.00010.

[USB21] USB Implementers Forum, Inc. *USB4 Base Specification*, 2021. URL: https://www.usb.org/document-library/usb4tm-specification.

[VD19] Stephen Van Doren. HOTI 2019: Compute Express Link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 18–18. IEEE, 2019.

[VKC17] Akanksha Verma, Amita Khatana, and Sarika Chaudhary. A comparative study of black box testing and white box testing. *Int. J. Comput. Sci. Eng*, 5(12):301–304, 2017.