UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

FPGA partial reconfiguration and automatic variant generation as a side-channel attack countermeasure with functional HDL Clash

Just van Westerveld

Master thesis

December 2020

Graduation committee:
prof.dr.ing. D.M. Ziener
ir. H.H. Folmer
dr.ing. E. Tews
dr.ir. J. Kuper

Computer Architecture for Embedded Systems
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
PO Box 217
7500 AE Enschede
The Netherlands

Acknowledgements

During my graduation project I've had the help, support and pleasant company of many people, and I take this opportunity to express my heartfelt gratitude. First and foremost, I'd like to thank my supervisors, ir. Hendrik Folmer, dr.ir. Jan Kuper and prof.dr.ing. Daniel Ziener. Your continued willingness to make time for regular meetings has been invaluable. I am also grateful to Ali Asghar, MSc for his feedback.

I would like to thank the fellow students and employees working at the CAES research group of the University of Twente. I have much enjoyed the many lively and fun discussions during the coffee breaks and lunch walks with many of you. Furthermore, I am very grateful to the people of QBayLogic B.V. for offering me to do a part of my graduation project as an internship. Your devotion to your main product, the Clash language and its compiler, has inspired me, and I've found working with Clash to be a real joy.

Finally, I'd like to thank my family and friends for their support. In particular, and last but far from least, I thank my parents for their unwavering support and guidance.

Abstract

This thesis proposes the automatic variant derivation of given hardware architectures using functional HDL Clash. The purpose of those variants is implementation diversity, which would involve their alternated use to introduce temporal jitter as a side-channel attack countermeasure. One systematic derivation approach involving left-fold higher-order functions is worked out as a proof of concept. Through traversal of the abstract syntax tree representation of hardware designs, it could be applied wherever possible in the `MixColumns` step of an AES cipher implementation. This fully automatic compile-time application yielded over 50000 variants, their suitability for impeding side-channel attacks in is yet to be verified.

Additionally, the core components needed to realize a Clash FPGA design featuring partial dynamic self-reconfiguration are identified and implemented. A design element representative of a reconfigurable region is defined to have different compilation and simulation behaviour. It compiles to the HDL sources needed by FPGA EDA tools to enable partial reconfiguration, and it simulates — like on an actual FPGA — the exchange of modules. In combination with a novel partial reconfiguration controller design, simulation and measurements from a Xilinx FPGA were shown to be true to each other.

Contents

1 Introduction

Automatic implementation diversity In recent years, side-channel attacks (SCA) have been demonstrated to be a major treat to unprotected cryptographic implementations [1]. As cryptography is widely used for the secure storage and communication of sensitive data that should only ever be accessible to trusted entities [2], research of countermeasures has been very active.

Side-channel attacks and fault attacks are classes of non-invasive physical attacks in computer and embedded hardware security [3]. They exploit information that inadvertently leaks from an algorithm's *implementation* in a device through that device's intrinsic physical interfaces with the outside world rather than weaknesses in the algorithm itself. Exploitable side-channels include power consumption [4], electromagnetic radiation [5] and timing behaviour [6]. Many powerful attacks depend on statistical analysis of many aligned measurements of a device's side-channels while in operation. In practice, countermeasures aim to raise the cost of mounting a successful attack beyond the gain due to a success by raising the number of required measurements [7].

Some countermeasures for hardware implementations utilize *implementation diversity* to introduce *temporal* or *spatial jitter* to side-channel leakages [8, 9, 7]. Here, some parts of the algorithm are alternately carried out by functionally equivalent but physically different *variants* of the implementation. For the countermeasure to work, these variants need to have differing side-channel leakage characteristics. They can be derived at either the *algorithm level* by transforming the algorithm specification or at the implementation level by transforming how the design is synthesized, placed and routed. Implementation diversity has been shown to mitigate attacks to some extend [8, 9, 7], and can be used in combination with other countermeasures to further increase the level of protection.

One purpose of this work was to explore a generic way of deriving hardware variants at the algorithm level. To the best of my knowledge, an automated way of doing this in the context of computer and embedded hardware security has not been attempted before. The merits of this approach are that implementation diversity countermeasure could be automatically applied to any suitable algorithm, thereby facilitating fast application and prototyping of hardware designs with this countermeasure. This led to the first two research questions:

1. How to derive functionally equivalent variants of a given hardware architecture in a systematic way so that their alternated use as implementation diversity can be a countermeasure against side-channel attacks?

2. Can such systematic derivations of variants be automated?

In answering these questions, the Clash functional hardware description language (HDL) was used because of its powerful language features. This gave rise to the third research question:

3. Given that the formalism of Clash is strongly mathematical, would Clash be a suitable language for performing such derivations?

FPGA partial reconfiguration All variants of a design utilizing implementation diversity need to be usable in runtime. The straight-forward solution would be to implement all variants in hardware side-by-side. To save hardware resources, the variants could alternatively be alternately implemented on the same hardware if a reconfigurable computing platform is used. Such platforms can be found in many complex embedded systems. They provide a middle-ground between the high flexibility of software and the high-performance of custom hardware.

Field-programmable gate arrays (FPGAs) are a widely used platform for the deployment of reconfigurable computing applications [10]. FPGAs have a range of configurable hardware resources [11]. How these behave and interconnect is determined by an FPGA's *configuration*, which can be changed after production of the FPGA itself. The process of *dynamic partial reconfiguration* (DPR) changes part of such a configuration and enables runtime circuit modifications on demand. A common use case is exchanging design parts that are never needed at the same time to save on hardware resources: a smaller and cheaper FPGA that consumes less power may suffice [12].

Another purpose of this study was to identify, investigate and, if necessary, implement the core components needed to realize an FPGA design featuring DPR. Specifically, designs in the Clash functional hardware description language are considered since that language was used for the aforementioned implementation diversity related work. Support for simulation is desirable for the purpose of functional verification. To the best of my knowledge, how a design in a functional HDL such as Clash could be specified, simulated and deployed when using DPR was never examined in detail. This led to the final two research questions:

4. How to simulate the runtime change of a hardware architecture that results from partial dynamic FPGA reconfiguration in Clash?

5. How can the Clash compiler be made to produce the sources needed by FPGA EDA tools to enable partial reconfiguration?

1.1 Document outline

After discussing some relevant background information and related work in chapters 2 and 3, Chapter 4 works out an approach to automating implementation diversity at the algorithm level. Chapter 5 realizes a workflow for partial reconfiguration enabled FPGA designs using functional HDL Clash. Chapter 6 then combines both works to apply implementation diversity in an AES implementation. Chapter 7 finishes with some conclusions and recommendations for future work.

2 Background

This chapter gives background information about side-channel and fault attacks in Section 2.1, about FPGAs and their ability of partial reconfiguration in sections 2.2 and 2.3 and about the Clash HDL in Section 2.4.

2.1 Non-invasive physical attacks in embedded system security

This section discusses two non-invasive physical attacks and how they can compromise the security of cryptographic implementations.

Side-channel attacks (SCAs) are a class of non-invasive physical attacks [3]. They passively exploit information leaking from an algorithm's implementation in a system in hard- or software and that system's intrinsic physical interfaces with the outside world rather than weaknesses in the algorithm itself. This makes them hard to prevent with there being many different reasons for leakages in the design space [13]. Exploitable sources of information include a system's power consumption [4], electromagnetic radiation [5] and timing behaviour [6]. For example, bit flips in a digital system affect its instantaneous power consumption and can be related to the data that it is processing. Successful applications of side-channel attacks enable the extraction of sensitive data from a system, even when the algorithm that it implements is considered mathematically secure. Known weaknesses in the algorithm can however aid the effectiveness of these attacks.

Fault attacks (FAs) are a type of non-invasive physical attacks [3]. Here, the system is actively brought outside of its intended operating conditions by altering its environment. This is done with the intention of obtaining information from the errors caused by the faults that this induces. For example, the temperature could be changed [14], the power supply or clock signal [15] may be modified at specific moments in time (glitches), or the system could be subjected locally to intense light beams [16] or electromagnetic fields [17]. Fault injection is a probabilistic process with a certain chance that an injection leads to an exploitable error.

2.1.1 Cryptography

Electronic systems may contain or operate on sensitive data that should only ever be accessible by trusted entities. *Encryption* is commonly used for the secure storage and communication of such data [2]. It is the process of encoding to an unintelligible alternative representation of data that can only be reversed through *decryption* by a party that has the right *secret key*. *Ciphers*, the encryption and decryption algorithms, are designed such that decryption without knowledge of the correct key is very, very computationally expensive. Possession of the encrypted data is therefore rendered useless without knowing the corresponding key. Encryption keys are a common target of physical attacks.

The widely embraced Kerckhoffs's principle states that the security of cryptosystems should depend solely on secrecy of the key, not on the obscurity of the cipher used. In line with this principle, many encryption standards, such as the wildly popular *Advanced Encryption Standard* (AES) [18], are scrutinized by the scientific community. Yet, implementations of ciphers, even those that are considered mathematically secure, may still be attacked successfully by exploiting their side channels to unveil the secret key in a limited amount of time.

2.1.2 Power analysis attacks

Power analysis attacks are a commonly used subset of side-channel attacks in practical attacks. They were introduced by Kocher et al. [4] and use a system's power consumption side-channel by measuring power over time at its power supply to obtain power traces. Örs et al. were the first to show experimental results on an FPGA in 2003 [19].

Simple power analysis (SPA) considers only few measurements and can succeed when measurements show distinctive patterns depending on the secret information. This is the case when whether or not certain operations are performed depends on the data being processed [20]. This attack can typically be mitigated by simply removing this dependency, though less sensitive information such as the cipher being used or the number of rounds of a round-based cipher being used may still be uncovered. System in- and output may not need to be known for the attack to be successful.

Differential power analysis (DPA) is more sophisticated and uses statistical analysis of many aligned power traces and typically some corresponding system in- and outputs. SPA may be used first to determine the algorithm under attack. Chunks of the secret key are targeted individually in a divide-and-conquer fashion. Tiny differences in power consumption can be discerned and correlated to the most likely values of the secret key chunks using a hypothetical leakage model.

CMOS logic power dissipation Correlation side-channel attacks require leakage models to relate a system's side-channel measurements to its internal data. In case of power analysis of digital electronics, considering the dynamic power dissipation of logic gates is appropriate. The CMOS style of digital circuit design and fabrication process is used for the vast majority of integrated circuits [21], including FPGAs. PMOS and NMOS are combined to make up digital logic gates. For example, Figure 2.1 pictures a CMOS inverter.

Apart from static leakage currents as a source of CMOS logic power dissipation, *dynamic* currents result from the switching of a transistor from a logic zero to a one or vice versa. Due to non-zero rise- and fall-times of transistors, there is a short time during which they (partially) conduct simultaneously, thereby creating a low-resistance path from V_{DD} (positive voltage) to V_{SS} (negative voltage, typically ground) through their source and drain contacts. Furthermore, such switching results in the charging and discharging of several load capacitances C_L. The dynamic power consumption could for example be modelled as in Equation 2.1 as described by Standaert et al. in [23]. Here, the product of the clock frequency f and the

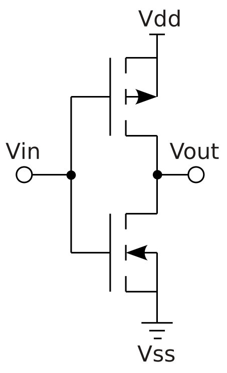

Figure 2.1: *A CMOS inverter. Source: [22].*

probability of a logic $0 \rightarrow 1$ transition $P_{0\rightarrow 1}$ is referred to as the *switching activity*. The highest peak in power consumption will appear during this rising transition $(0 \rightarrow 1)$ because of the charging of capacitances [23].

$$P_{dyn} = C_L V_{DD}^2 P_{0\rightarrow 1} f \tag{2.1}$$

2.2 Reconfigurable hardware: FPGAs

Off-the-shelf microprocessors such as microcontrollers, CPUs and GPUs provide easy means to running computationally-intensive applications in software. Yet, these flexible general-purpose processing platforms have overhead in that they are not tailored to the application at hand. In situations where power, area or speed requirements are stringent, it may make more sense to go for a custom implementation in hardware. Such a custom design could either encompass the entire application or offload and accelerate some part of the workload from a general-purpose microprocessor.

In such situations, *Application Specific Integrated Circuits* (ASICs) could provide the optimal solution given today's semiconductor manufacturing processes. With ASIC designs needing to be built from the ground up, the possibilities are almost endless, and very fine-grained design and fine-tuning are possible. However, ASICs have low post-production flexibility because of their fixed hardware resources, and their design is quite involved. Reusable *Intellectual Property* (IP) blocks may be acquired and used to reduce engineering effort somewhat, but the non-recurring engineering (NRE) costs are high, and a manufacturing cycle alone can take months per design iteration.

When the application at hand does not warrant such engineering expenses in both time and money, reconfigurable computing featuring *Programmable Logic Devices* (PLDs) strikes a balance between ASICs and general-purpose computing platforms in power, performance, flexibility, design effort and cost. Many PLD technologies exist [24], including CPLDs, PALs/GALs, MPGAs and FPGAs, of which at least the last is widely used [10].

Field-Programmable Gate Arrays (FPGAs) are integrated circuits providing a wealth of hardware resources designed to be configurable after production of the FPGA itself [11]. They therefore enable highly application-specific hardware implementations while being an off-the-shelf product without any manufacturing time at each design iteration with them being configurable in a matter of seconds. Figure 2.2 gives an overview of the core resources that can be distinguished on any FPGA:

- *Configurable Logic Blocks* (CLBs) that implement logic functions using Look-Up Tables (LUTs) and flip-flops
- *Configurable I/O blocks* (IOBs) that facilitate off-chip connections
- *Configurable routing* that interconnects all resources

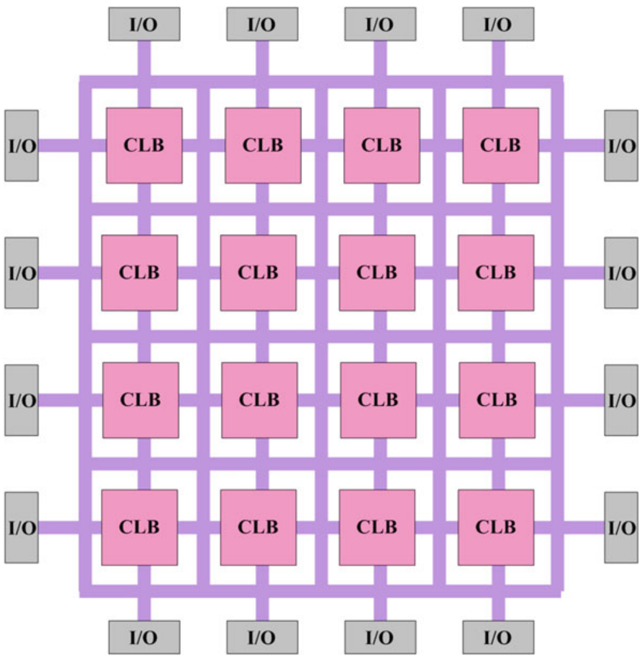

Figure 2.2: *Schematic depiction of a basic FPGA architecture. Source: [11].*

In addition to this, most modern FGPAs also feature fixed DSP and large memory (*block RAM*) blocks interspersed with the other configurable resources for accelerating some common logic functions and memory tasks. Some devices combine configurable logic with a hard processor core on the same chip in a SoC-like fashion to alleviate the need for, also popular, soft processor cores implemented in the configurable logic.

As opposed to ASIC designers, FPGA designers generally do not need to concern themselves with clock and reset nets, clock sources, power distribution, manufacturing process variations, parasitic capacitances or I/O communication designs as all such things are already taken care of in the design of the FPGA itself. An FPGA's configurability does come at a cost with the overhead in silicon area, propagation delay and power consumption being significant [25]. Yet, FPGAs provide faster time to market and are generally cheaper overall when the volume of production is limited [26, 10].

2.3 FPGA partial reconfiguration

This section introduces partial reconfiguration: the process of altering part of an FPGA design in runtime.

2.3.1 Changing the configuration memory

The functionality of an FPGA is fully determined by its *configuration* as stored in its *configuration memory* [27]. Changing this memory's contents enables the implementation of new designs and can be accomplished through *reconfiguration*. A configuration is referred to as the whole of the FPGA's currently implemented functionality. This implies that reconfiguration, no matter how big or small the reconfigured area, yields a different configuration.

Full reconfiguration concerns the whole of an FPGA's configuration memory. It leaves no trace of any previous implementation, and therefore does not require any consideration regarding whether the new configuration is compatible with any previous ones. This kind of reconfiguration is most commonly used, if only because of it being required at power-up with most FPGAs utilizing volatile in nature SRAM-based configuration memories [28, 11].

On the other hand, selectively reconfiguring part of an FPGA's configuration memory is referred to as *partial reconfiguration* (PR) [27]. Execution on the whole FPGA is paused when performing *static* partial reconfiguration, whereas *dynamic* (or: *active*, *run-time*) *partial reconfiguration* (DPR) allows changing part of the design while leaving the rest of the circuitry functional and intact without interruption.

Configuration *bitstreams* determine what the contents of an FPGA's configuration memory should look like, and therefore how an FPGA is to be configured [29]. They are the main compilation output of *electronic design automation* (EDA) tools. *Partial bitstreams* are relevant to partial reconfiguration as they concern only part of the configuration memory. FPGA (re)configuration is the act of feeding an FPGA *configuration interface* with the desired bitstream. FPGAs may feature such interfaces that are accessible from within the FPGA fabric itself to enable *self-reconfiguration*.

2.3.2 Reconfigurable regions and modules

Each physical region designated for partial reconfiguration in an FPGA design is referred to as a *partially reconfigurable region* (PRR), or *reconfigurable region* for short. Conversely, the remainder of the area is called the *static region*. The configuration in a reconfigurable region can be alternately defined by *reconfigurable modules* (RMs). Any such region needs to be at least as large as its largest module.

2.3.3 Use cases

When some parts of an FPGA design never need to be active at the same time (i.e. in a mutually exclusive fashion), they may be implemented as reconfigurable modules in a reconfigurable region. Smart applications of this time-multiplexing of FPGA area enable a variety of use cases [12]. Hardware resources can be saved with not all modules needing to be implemented on the FPGA fabric simultaneously as would be the case in a static design. A *smaller* and *cheaper* FPGA that consumes *less power* may suffice. This benefit increases with more modules per region.

Moreover, PR may help in situations where an active FPGA design needs to be updated with minimal downtime or without losing state. A partial bitstream may target only changed regions or even solely consist of the difference between the old and new configurations [30], thereby reducing the bitstream size and reconfiguration time. This can also be applied for fast device start-up, with the logic that is not required in a very short time span after power-on being configured later.

Finally, PR may be utilized as part of countermeasures for certain physical attacks, an idea explored further in this thesis.

2.3.4 Vendor support

The support of and interface for partial reconfiguration on commercial FPGA devices differs from vendor to vendor. Most major FPGA vendors — like Intel, Xilinx and Lattice — nowadays support partial reconfiguration of at least part of their product families. The FPGA resources that qualify and the granularity and speed with which they can be reconfigured vary.

2.3.5 Reconfiguration controllers

To reconfigure a reconfigurable region, an FPGA's *configuration interface* needs to be supplied with the desired partial bitstream. The soft- or hardware responsible for this process is referred to as the *Partial Reconfiguration Controller* (PRC). On receiving a trigger for reconfiguration, a PRC has to carry out several tasks. Note that some of these tasks are not essential, but may be desirable for practical and easier applications of PR. These tasks are generally categorized in the *before*, *during* and *after* phases of reconfiguration.

Before Optionally wait for or notify the old module to save its state, flush its pipeline or otherwise give the green light for reconfiguration.

During Retrieve the desired partial bitstream from a storage location and supply it to a configuration interface of the FPGA.

Optionally isolate the module boundaries to avoid seemingly random signals from the region under reconfiguration to propagate through the static region. Conversely, dynamic (i.e. changing) signals coming from the static into the reconfigurable region may also be undesirable during reconfiguration when it is not reset afterwards [27].

After Optionally restore the new module's state or assert its reset signal to bring it in a predefined state.

2.4 Hardware description language Clash

Digital hardware designs are generally made using a *hardware description language* (HDL). This section introduces one such language: Clash.

VHDL and Verilog are the widely-used traditional HDLs at the *register-transfer level* (RTL) of abstraction. Such languages describe *structure*: the flow of digital signals between the hardware *registers* that store state between clock cycles in synchronous designs. Conversely, *high-level synthesis* (HLS) uses algorithmic descriptions of a hardware design.

In a standard EDA tool flow, HDL descriptions are *synthesized* into *netlists*: descriptions of *logic gates* and how they interconnect. *Place and route* then maps the logic gate descriptions onto the available hardware resources. Typically only a subset of a HDL is synthesizable. Simulators exist so that the behavioural correctness can be tested during design.

HLS tools provide a higher level of abstraction by working on algorithmic description of a desired hardware behaviour. Such tools derive a corresponding implementation in hardware and perform pipelining — the insertion of registers — given size and latency constraints. The fact that most HLS tools use imperative programming languages can be problematic in the sense that their sequential nature makes that they do not capture the parallel nature of hardware well. This may frustrate the transparency of the synthesis process: it can be hard to develop a feel for what hardware structure will result from a given HLS description.

Clash [31, 32] is a functional HDL that uses the syntax and semantics of the functional programming language Haskell [33]. It is of a high abstraction level in the sense that it supports Haskell's powerful language constructs such as higher-order functions, type classes and parametric and ad hoc polymorphism. On the other hand, it is not so much high-level in the sense that it still works on the RTL level of abstraction and the designer still has to insert every single register. Clash being strongly typed and pure (expressions are referentially transparent) enables highly-parametrizable designs and makes it suit the immutable nature of hardware well. For those uninitiated with Haskell and Clash, some language aspects are introduced in Appendix A.

Hardware descriptions in Clash are supported by the free and open source Clash compiler. Figure 2.3 visualizes what an FPGA design flow looks like. The compiler's interactive shell (REPL) enables circuit simulation by evaluating the Haskell functions that any Clash design consists of. The compiler can compile Clash descriptions to VHDL, Verilog and SystemVerilog to enable the use of conventional EDA tools for synthesis, placement and routing.

Clash features a templating system that allows the annotation of Clash functions with *HDL primitives*. They specify exactly what the compilation output for the corresponding function should look like in the compiler's supported output HDLs. The function declaration is then only used for simulation purposes. Use of HDL primitives allows for the integration of existing non-Clash IP blocks.

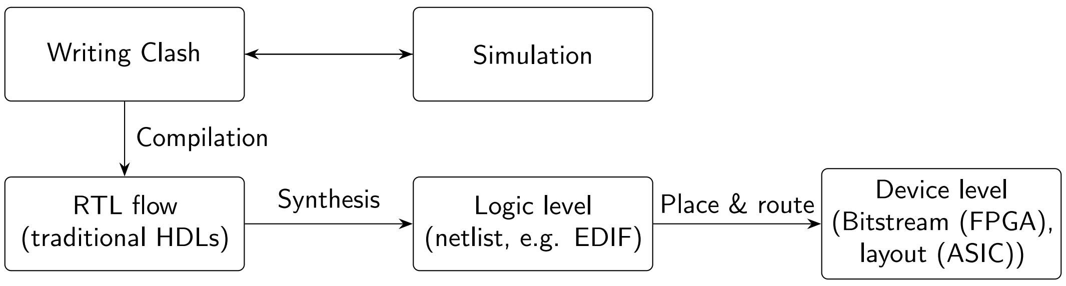
Writing Clash
Simulation
Compilation
RTL flow (traditional HDLs)
Synthesis
Logic level (netlist, e.g. EDIF)
Place & route
Device level (Bitstream (FPGA), layout (ASIC))

Figure 2.3: *What an FPGA hardware design flow using Clash may look like.*

3 Related work

In this chapter, some relevant related work is outlined and discussed. The first section considers mitigation techniques to certain physical attacks with a focus on those utilizing implementation diversity to do so. The second section considers approaches to modelling and simulating FPGA designs using PR.

3.1 Side-channel and fault attack countermeasures

The existence of side-channel and fault attacks has shown that cryptosystem security comes and goes with not just security of the cipher, but also with security of the hardware platform that the cipher is implemented on. Many studies have focussed on side-channel and fault attack mitigations techniques. This section outlines some of those that concern hardware implementation and utilize FPGA partial reconfiguration to introduce implementation diversity.

Since fault attacks are active, a different class of countermeasures deals with trying to detect fault insertion and to take action based on that. This can be accomplished through either error checking logic or sensors. This class of countermeasures is not considered further in this thesis.

A number of samples is required to mount a successful side-channel or fault attack. In practice, countermeasures aim to raise that number, and thereby make an attack more difficult [7]. Ideally, countermeasures are sufficient to render attacks *infeasible*, that is: to raise their cost beyond the gain due to a success. Countermeasures utilizing implementation diversity generally do so to introduce *temporal* or *spacial jitter*.

In case of temporal jitter, the instance of time at which certain sub-operations are performed depends on the active reconfigurable module and is different for various executions of an algorithm because of PR. This breaks the assumption of differential side-channel and fault attacks that changes in side-channel measurement traces depend solely on changes in system input signals; they now also depend on which variant is active. For such a countermeasure to be effective, it is essential that an attacker is not able to discern the individual variants or force a single one to remain active. Otherwise, traces could be statistically analysed per variant which would largely nullify this kind of countermeasure. The overall encryption time should remain constant to prevent timing analysis attacks from giving away the active variant.

Addition of spacial jitter by continually changing where certain sub-operations of an algorithm are performed on an FPGA can counter fault attacks as the area that an attacker needs to target is not constant. Furthermore, certain transient faults that are induced in a reconfigurable region can be undone through reconfiguration [7].

We distinguish two levels at which variants for implementation diversity can differ. At the *algorithm* level, the algorithm that is implemented on the FPGA is varied. Each variant will have a different RTL description. Conversely, implementation diversity at the *implementation level* is used to refer to approaches that alter synthesis (and thus the netlist), placement and/or routing to obtain variants. Here, the RTL description does not change.

Mentens et al. (2008) Mentens et al. [7] appear to have been the first to propose physical attack countermeasures involving PR. They consider two approaches, one involving temporal jitter, the other both temporal and spatial jitter.

The first approach concerns the architecture level and adds temporal jitter to an AES-128 implementation using a dynamically reconfigurable switch matrix. AES is a round-based block cipher that each round performs four sub-operations sequentially: AddRoundKey, SubstituteBytes, ShiftRows and MixColumns, in that order [18]. Through the switch matrix, two registers in the static region are each inserted at locations in between two of the four or after the fourth AES round sub-operations. After which sub-operation each is wired up depends on the switch matrix configuration, which is changed through PR.

The authors note that there are eight configurations with differing temporal shift for the in SCA commonly targeted SubstituteBytes sub-operation when using the fact that SubstituteBytes and ShiftRows can be swapped. They claim that the number of measurements required to break the implementation using a DSCA is increased by a factor of over three under conservative assumptions. With regards to fault analysis attacks, they claim that the countermeasure is effective due to the aforementioned general reasons as to how temporal jitter can help in that regard.

With only the switch matrix being reconfigured and with it consisting of only a bunch of wires, the reconfigurable region can be very small and reconfiguration fast. The number of possible configurations is small.

As a second approach to further increase the resistance of the implementation to fault analysis attacks that target a system locally, such as with optical ones, the authors also propose a countermeasure utilizing both temporal and spatial jitter. This concerns both the architecture and implementation levels. When applied to AES, the reconfigurable modules are the four AES sub-functions with and without an appended register. With them being relocated in runtime, the probability of successfully targeting them with local fault injection processes such as optical ones now has a lower probability.

Hettwer et al. (2019) Hettwer et al. [8] ventured exchanging 128 different variants of the entire datapath of an 8-bit serialized AES-128 encryption implementation. These variants differ at the implementation level as they are derived from the same synthesized netlist. Their mutual differences in physical layout come from the fact that for each variant implementation 80% of the slices in the reconfigurable partition are randomly prohibited for placement. Practical measurements are shown to be effective against EM analysis and FAs with resistance against power-based SCAs being increased by a factor of 2 to 3 at a significantly higher hardware resource utilization.

Bow et al. (2020) Bow. et al. [9] build on the concepts introduced in [34]. They consider an AES cipher with 18 reconfigurable regions for SBOX implementations that are regularly reconfigured. One of them is used at a time for the SubstituteBytes step of AES. The SBOX variants are derived at the implementation level by using four different standard cell libraries during synthesis. Moreover, more variants are added through, among other things, addition of random buffer delays to in- and outputs or to internal nodes, addition of dummy fan-out capacitive loads, and through manipulation of the clock tree. Proof of concept experiments suggest a significant increase in correlation power analysis resistance.

In conclusion The few approaches that apply implementation diversity with variants differing at the algorithm level, like [7], use variants that are manually tailored to and optimized for the application at hand. Conversely, the approach proposed in this thesis is more general, with variants of a given hardware architecture being automatically derived. Whether this more general approach can yield variants that are sufficiently effective as a countermeasure will need to be shown.

3.2 Reconfiguration modelling & simulation

Designing for FPGAs brings several additional challenges when utilizing partial reconfiguration. The design parts that are only ever needed in a mutually exclusive fashion need to be identified or designed as such. Their internal state may need to be stored and restored and the timing needs to work out. Having *electronic design automation* (EDA) tool support can therefore be highly beneficial.

Simulation is widely-used as part of *functional verification*: verifying that a design conforms to specification. There are several tools and frameworks that support the definition of reconfigurable modules and functional simulation of their exchange in reconfigurable regions that do so in high-level synthesis languages such as SystemC. This thesis, however, focusses on defining and simulating designs using DPR at a lower abstraction level: RTL. This facilitates cycle-accurate analysis of timing behaviour and the uncovering of faulty designs. In order to simulate the full design, support for simulating the exchange of modules is required as opposed to simulation of individual configurations.

Xilinx Vivado, the vendor EDA tool for Xilinx FPGAs, supports standard simulation of individual configurations, but not of the process of partial reconfiguration itself [27, p. 85]. This is, however, still possible to accomplish through the manual insertion and management of multiplexers, to be used in simulation only. This is an approach supported and automated by Intel Quartus, the vendor EDA tool for Intel FPGAs, as illustrated in Figure 3.1 [35]. The PR Activate input in that figure should be active during simulation of reconfiguration to have the output set to unknown values.

ReSim [36] is is a library written in SystemVerilog that works similar to Intel's approach in the sense that it operates at the RTL level and that is automatically inserts simulation-only multiplexers. It does so through something that represents

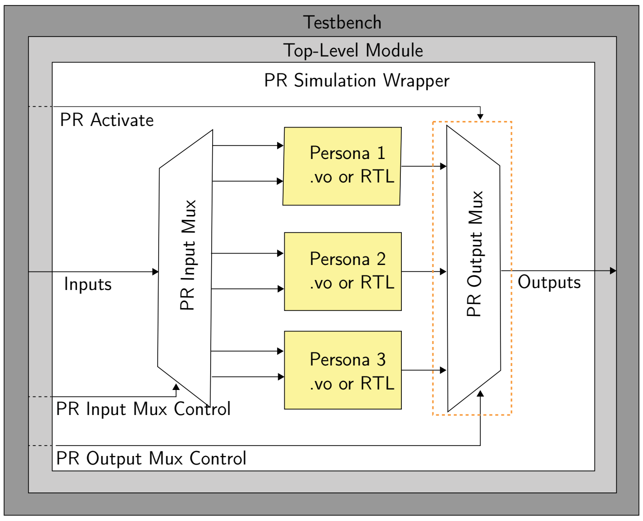

Figure 3.1: *The wrapper for PR simulation provided by Intel Quartus Prime Pro Edition. Intel uses the term* persona *for reconfigurable module. Source: [35].*

the reconfigurable region named the *extended portal*. ReSim features simulation-only bitstreams that are similar in layout compared to real bitstreams. Such bitstreams are retrieved from a simulation-only bitstream storage location by the PRC and led to an artefact representative of possible configuration interfaces. That artefact extracts the region and module identifiers for the extended portal to know the reconfiguration status and which module to multiplex next. On initiating PR, the extended portal selects the new module that is now under configuration. It is fed with undefined "x" values to help test the initialization mechanism that may be present to bring the module in a predefined state after reconfiguration. Meanwhile, the extended portal output is also set to undefined "x" values to help test the region isolation logic.

The PR library presented in this thesis is similar in functionality compared to the approaches taken by Intel and in ReSim. In line with the latter, undefined "x" values are propagated out of the reconfigurable region to help test any region isolation logic. The module under reconfiguration is not already selected during simulation of reconfiguration, but this should not prevent testability of the module initialization logic. The bitstreams used in simulation are user-provided, and can therefore contain anything. Unlike with ReSim, they are not used to extract the reconfiguration status and active regions. Instead, this information needs to be passed explicitly to each reconfigurable region simulation artefact. This digital signal is unused outside of simulation and therefore does not take any hardware resources. For cycle accurate simulations representative of an actual hardware implementation, the module bitstream sizes need to be back-annotated after synthesis and implementation. To the best of knowledge, the PR library presented in this thesis is the first to be written in and to support hardware designs and simulations with DPR in Clash.

4 Automatic implementation diversity at the algorithm level

As discussed in Chapter 3, related work, implementation diversity can be an effective countermeasure to side-channel and fault attacks. It uses the fact that a desired behaviour of a digital circuit can generally be realized in many different ways where each may have different side-channel leakages. The limited amount reconfigurable resources on an FPGA limits this somewhat, but these resources can still be interconnected and configured in many different ways and still have the same functional result. In that context, this chapter explores generic approaches to deriving variants of an implementation with the purpose of automating implementation diversity at the algorithm level. This relates to the first two research questions.

Several concepts for deriving variants at the algorithm level are explored in Section 4.1. One of those concerns higher-order functions, a commonly used abstraction in functional programming languages that is present in Clash and can be used describe *structure* in hardware. This concept is worked out to a set of provably-correct *transformation rules* that describe how certain *left fold* higher-order functions can be rewritten to functionally equivalent yet structurally different variants in Section 4.2.

For automating the application of transformation rules like the aforementioned, an appropriate compiler interface is picked in Section 4.3. The choice fell on Template Haskell. Its compile-time metaprogramming facilities are used in Section 4.4 to implement a modified version of one of the left fold transformation rules as a proof of concept. The resulting *transformation primitive* is a function that receives a Clash expression and returns expressions for all transformation rule outcomes if all conditions for the rule are met.

Transformation primitives like the above kick into action *only* when encountering an expression that matches the transformation rule conditions, for example on an expression that is a call to a left fold function. It is however desirable to match on any suitable expression, even if it is a subexpression that is part of a larger one. Section 4.5 proposes a way of constructing *variant transformations*: functions that, given a section of Clash code, takes any transformation primitive, applies it wherever possible and accumulates all resulting variants. Furthermore, an implementation is proposed for applying a given variant transformation to a Clash function or variable declaration and bringing any resulting variants in scope for them to be usable in the rest of an FPGA design.

4.1 Concepts

This section considers some general concepts for acquiring several variants of a design, a prerequisite for adding implementation diversity. The variants need to be functionally equivalent but exhibit different side-channel leakages in relation to one another.

Altering data representations Any value in Clash has a type. Each member of that type will get a different bit-representation when compiled. At least $\lceil {}^{2}\log\ n \rceil$ bits are required to represent values of a type with n inhabitants. Unless the type in question has no or a single inhabitant, this injective mapping to binary can be performed in different ways. For example, a value of the `Colour` data type of Listing 4.1 is either `Red`, `Green` or `Blue`, so $\lceil {}^{2}\log\ 3 \rceil = 2$ bits are required to represent those values. Therefore, the binary numbers 00, 01, 10 and 11 can be arbitrarily assigned to represent `Colour`'s values.

```
data Colour = Red
            | Green
            | Blue
```

Listing 4.1: *A data type with three data constructors, each representing a colour.*

How many and which bits on wires in digital hardware flip from a zero to a one and vice versa as inflicted by values changes depends on how values are represented in binary. Changing these representations affects side-channel leakages and might change the location of where an attacker may wish to inject a fault.

Altering arithmetic operation implementations Several implementations and algorithms for arithmetic operations exist. For example, an adder could be implemented as ripple-carry or carry-lookahead using FPGA CLBs or implemented using FPGA DSP blocks. Each may exhibit different side-channel leakage characteristics.

Rewriting higher-order functions Higher-order functions (HoFs) are a commonly used abstraction in functional programming languages. Such functions take a function as an argument or produce it as a result. For example, `zipWith` from the Haskell Prelude module is a HoF that zips two lists using a given binary function; `zipWith (+) [1,2,3] [4,5,6]` evaluates to `[5,7,9]`. Jan Kuper et al. [37] note that HoFs express structure and may be seen as hardware architectures as things like memory usage and execution time — relevant when executing on a microprocessor — have been abstracted away from.

It is possible to rewrite certain higher-order functions using transformation rules that are provably correct and meaning preserving [38, 37, 39]. The transformation outcomes can be seen as functionally equivalent variants and may be used to introduce spatial and temporal jitter.

Retiming Retiming is the process of moving the structural locations of registers in digital circuitry in a manner where the same functional behaviour is retained. When moving a register over e.g. a logic gate, one register must be consumed at every gate input and one placed at every gate output or vice versa. Additional difficulty comes when the registers have reset values.

Retiming is typically performed as an optimization step to improve circuit performance by shortening the critical path or to reduce circuit area or power consumption. It affects when operations are performed and can be used to introduce temporal jitter.

Conclusion All previously mentioned approaches have potential in that they could be used to obtain functionally equivalent variants that may have differing side-channel leakages. In the end, higher-order functions were pursued. They are an iconic feature of Clash, the somewhat unconventional HDL that I work with, and there are systematic approaches that are provably correct in which variants can be ascertained.

4.2 Higher-order function transformation rules

This section outlines some transformation rules on higher-order functions (HoFs) as proposed in existing literature.

Higher-order functions are a commonly used abstraction in functional programming languages. They are used by Rinse Wester in [38] to express a mathematical formulation of a DSP algorithm that is subsequently transformed using a set of transformation rules that distribute computations over *time* and *space*. In general, computations can be implemented in time by performing operations sequentially or in space by performing operations in parallel. This translates to an increase in execution time or chip resources respectively in case of hardware design. The exact distribution between these two is controlled by a parameter introduced by the proposed transformations rules which allows for an optimal trade-off. These rules are applicable to several common higher-order functions and are proven to be meaning-preserving.

In [37], Jan Kuper et al. propose very similar transformation rules. They propose annotating imperative code fragments with Haskell higher-order functions that indicate the *computational structure*. The annotations are mathematical in nature and allow for provably correct transformations on them with the goal of finding a good mapping to the target hardware platform that is to execute the algorithm. Corresponding transformations that are hard to find by themselves have to be performed on the imperative code fragments. To demonstrate the idea, the paper outlines a few transformation rules that are proven to be correct.

Both works introduce transformation rules on the commonly used left fold function. In functional programming, folding refers to a family of higher-order functions that can reduce container-like data structures to a single return value using a given binary reduction function. Given some constraints on the reduction function, the concept of implementation diversity can be applied to folding functions particularly well as there are many different ways in which the reduction can then be arranged.

GHC's Prelude module exports `foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b`. It takes as arguments a reduction function, an initial value and a foldable data structure, in that order. Its structure is illustrated in Figure 4.1. The left fold function and the below transformation rules apply to any foldable and splittable data structure, including vectors, but they will be explained using lists. As an example of folding a list, `foldl (+) 0 [1,2,3]` would evaluate as `((0+1)+2)+3` to `6`. For background information on the likes of Haskell type signatures and function application, please refer to appendix A.

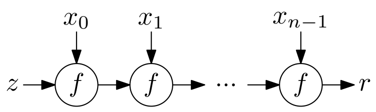

Figure 4.1: *The structure of* `r = foldl f z xs`*.*

Transformation rule 1 Any list `xs` of length $N \times M$ can be *unconcatenated* by splitting it up in N sublists of length M. The result is a *list of lists* `xss`. The folding can therefore be split up in N sections as illustrated in Figure 4.2. This transformation can be stated as:

```
foldl f z xs = foldl (foldl f) z xss
```
(4.1)

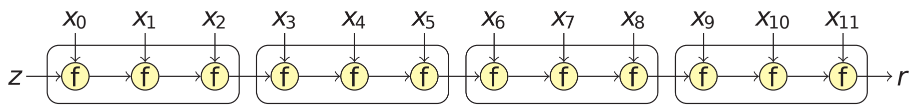

Figure 4.2: *An example of the structure resulting from applying the transformation rule of Equation 4.1. Here, $N = 4$ and $M = 3$. Source: [37], modified to use different variable names.*

Note that the four groups in the example of Figure 4.2 are occurrences of the inner fold (the one in parentheses) of Equation 4.1. They are folded by the outer fold, which uses the inner fold as its reduction function. This grouping allows for the inner fold to be implemented in *space* with it being computed in a single time instance, and for the outer fold to be implemented in *time* with each application of its reduction function being performed sequentially, over multiple clock cycles. This is illustrated in Figure 4.3. The overall ratio of distribution between time and space is determined by how the original list `xs` is unconcatenated to the list of lists `xss`. In a hardware design, only one instance of the inner fold (i.e. a single group) would need to be implemented, plus a memory element (such as a register) for storing the intermediate value of the outer fold.

Transformation rule 2 When the reduction function `f` is *associative* and has an *identity element* that is used as the initial value `z`, folding left could be restructured as illustrated in Figure 4.4. This transformation can be stated as:

```
foldl f z xs = foldl f z (map (foldl f z) xss)
```
(4.2)

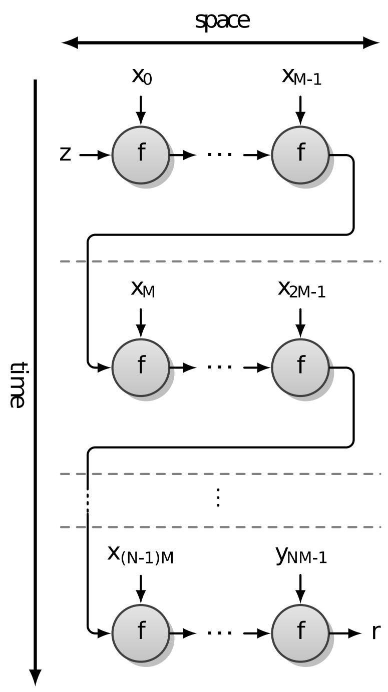

Figure 4.3: Left fold distributed over time and space. Source: [38, Fig. 4.8], modified to use different variable names.

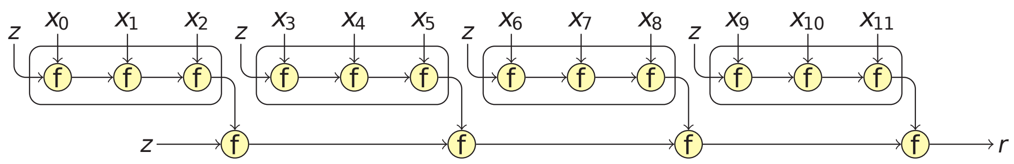

Figure 4.4: An example of the structure resulting from applying the transformation rule of Equation 4.2. Source: [37], modified to use different variable names.

Transformation rule 3 Finally, when the reduction function `f` is *also commutative*, that is: in addition to being associative and the initial value being the corresponding identity element, folding left could be restructured as illustrated in Figure 4.5. This transformation can be stated as:

```
foldl f z xs = foldl f z (foldl (zipWith f) zs xss)
```
(4.3)

Here, `zs` is a list with the initial value repeated N times. Note that remarks similar to the aforementioned about distributing the computation over space and time can be made about these latter two transformation rules.

4.3 Transformation automation: API choice

In order to automatically perform transformations of a given Clash hardware design, an appropriate interface with the compiler must be used. The Clash compiler exposes several with potential, each operating on a different representation of the

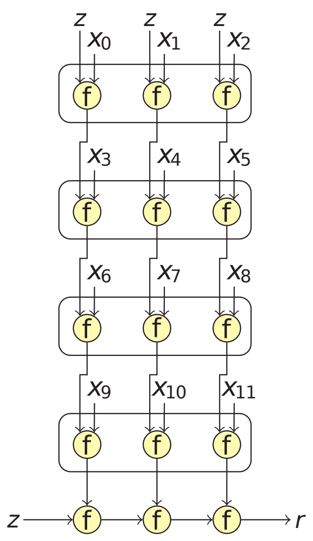

Figure 4.5: *An example of the structure resulting from applying the transformation rule of Equation 4.3. Source: [37], modified to use different variable names.*

user's design. They are compared in this section with the purpose of deliberately selecting one based on a trade-off between versatility and ease of use.

GHC Rewrite Rules GHC, the Haskell compiler Clash uses behind the scenes, features rewrite rules for optimization purposes. By putting one or more rules in a RULES pragma, any matching expression is replaced with an alternate one. For example, an optimization rule for rewriting two mappings over a list is depicted in Listing 4.2. If more than one rule matches, GHC arbitrarily chooses one to apply. All rules are implicitly exported from the module where they are defined. Therefore, the rule is in effect in any module that imports it, directly or indirectly.

```
{-# RULES
      "map/map"  forall f g xs.  map f (map g xs) = map (f . g) xs
  #-}
```

Listing 4.2: *A GHC rewrite rule from a map over a map over a list to a single map with the original functions composed. Source: GHC documentation (`https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#rewrite-rules`).*

GHC plug-ins Alternatively, GHC compiler plug-ins could be used for variant generation. Behind the scenes, GHC is used as a front-end library by Clash. In the compilation process, Clash lets GHC translate a given user design to its internal intermediate representation named Core. This language is very concise; many of the possible syntactic constructs in Haskell have been gotten rid of. Clash translates GHC Core to its own Clash Core representation to then submit to further transformations and eventually pretty-print the result in the user-selected target HDL.

The use of GHC makes that its plug-in functionality[1] can be used. GHC Core plug-ins insert an extra compilation pass on the Core representation into the compiler pipeline, and may therefore be used to implement the variant generation transformations. In order for such a hypothetical transformation plug-in to know where to try and apply these transformations, Haskell annotations pragmas may be used. They allow the attachment of data to identifiers in user Haskell code that is preserved during compilation. This data may then be retrieved by a plug-in, in this case to inform it which program sections to apply transformations to.

Template haskell with QuasiQuotations Template Haskell (TH) [40] is an extension to the GHC compiler. It allows for compile-time metaprogramming by constructing or manipulating *abstract syntax tree* (AST) representations of Haskell code. The TH AST is represented using an ordinary Haskell data type (ADT). *Quotation* bring Haskell code to such a representation, and *splicing* does the inverse. Please refer to Appendix B for more information.

Quotation and splicing enable the writing of variant generation transformations that operate at the TH AST level. Such a transformation may modify given source code, i.e. an expression or declaration, and opt for inserting additional variants.

Comparison Rewrite rules allow for very straight-forward rewriting of expressions. However, they can never add any new declarations or expressions for derived variants as only one rule is applied arbitrarily when multiple match. Furthermore, rewrite rules are static, and no functions can be used to determine the resulting expression based on the matching input expression. Lastly, rewrite rules cannot be restricted to certain sections of code that may be of interest for variant generation.

GHC core plug-ins are a lot more versatile. They operate on the very concise Core intermediate representation of the compiler where many syntactic constructs of Haskell have been gotten rid of. Transformation plug-ins that work at this level would be small and apply to many semantically equivalent Haskell language constructs. The scope where transformations are applied can be limited through the use of annotation pragmas.

Template Haskell-based transformations would operate on an AST representation of code that comprises the rich Haskell syntax in full. Rich interplay of splicing and quotation may nevertheless enable concise transformations. Limiting the scope of such transformations is implicit as the relevant code sections need to be enclosed by quotation brackets.

Template Haskell is the route taken given yours truly's familiarity with it from implementing the ILA Clash core proposed in Section 5.5 and given its presumed higher accessibility for those not involved in GHC compiler development when

[1]`https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/extending_ghc.html#compiler-plugins`

compared to GHC Core plug-ins. Rewrite rules weren't much of a contender given their many shortcomings.

4.4 Left fold transformation rule implementation

Now that a suitable compiler API is chosen, one of the transformation rules from Section 4.2 on left fold higher-order functions is modified to be applicable in more ways and implemented as a proof of concept. The resulting *transformation primitive* returns as variants all possible ways of applying the rule if given a suitable call to `foldl`.

4.4.1 Reconsidering the original transformation rules

All left fold transformation rules from Section 4.2 involve splitting an $N \times M$ long foldable (i.e. container-like) and splittable data structure, such as a list, in N equally sized chunks of M elements. This is desirable when wishing to efficiently distribute the computation over time and space: the N chunks can be processed consecutively on the hardware required for simultaneously processing M elements plus a register for storing intermediate values.

To use one of the transformation rules for generating variants for implementation diversity, its parameter must be varied. The parameter controls how the folded data structure is split. Finding all possible ways of splitting boils down to finding all possible factors N and M that have as product $N \times M$. An algorithm for that is provided in Listing 4.3.

```haskell
factorsOfProduct :: Integral a => a -> [(a, a)]
factorsOfProduct x =
  [ (a,b)
    | a <- [1..x]
    , b <- [1..x]
    , a * b == x
  ]
```

Listing 4.3: *A function that finds all possible pairs of positive integer factors that have a given positive integer number as their product. It uses a list comprehension to do so.*

Note that this problem does not have many solutions for small products. For example, the `MixColumns` step of the AES encryption standard — an algorithm for which implementation diversity could be beneficial — could be described by left fold higher-order functions with data structures containing five elements. Only two pairs of numbers have 5 as their product: `factorsOfProduct 5` evaluates to `[(1,5),(5,1)]`, sufficient for a mere number of two variants per left fold. Since `MixColumns` could be described using four such folds, the variants could however be combined to have a total number of $2^4 = 16$ variants.

If this approach were taken, many transformation outcomes require differing numbers of clock cycles to complete. It would complicate automating implementation diversity as the rest of the design would need to accommodate for varying timing behaviour and could allow an attacker to distinguish some variants through timing analysis. Registers could be added to the shorter variants as a work-around to even things out.

Instead, note that the left fold operation can easily be distributed in more ways when staying within a single clock cycle. The $N \times M$ long list needs not be split in N chunks of equal length M. Rather, it could be split in any way as long as the individual chunk sizes sum up to $N \times M$. Finding all possible ways of splitting with these loosened constraints therefore involves finding all possible combinations of terms that have as sum $N \times M$. Listing 4.4 has an algorithm for that. It enables many more ways of splitting a left fold as could be described for the `MixColumns` step of AES: evaluation of `termsOfSum 5` yields 15 results.

```haskell
-- | Finds all possible positive terms of a given integer sum.
--
-- For example:
-- > termsOfSum 4
-- [[1,1,1,1],[2,1,1],[1,2,1],[3,1],[1,1,2],[2,2],[1,3]]
termsOfSum
  :: Integral a
  => a
  -- ^ A sum of which to find terms
  -> [[a]]
  -- ^ All possible combinations of non-zero natural numbers that have the
  -- given number as sum
termsOfSum = Prelude.init . go []
  where
    go :: Integral a => [a] -> a -> [[a]]
    go xs 0 = [xs]
    go xs rem =
      [ xs'
        | p <- [1..rem]
        , xs' <- go (p:xs) (rem-p)
      ]
```

Listing 4.4: *A recursive function that finds all possible non-zero terms of a given sum. It uses a list comprehension to do so.*

Because its simplicity, it was decided to go for this approach of not distributing transformation outcomes over time, but rather staying within a single clock cycle. From the transformation rules discussed in Section 4.2, this immediately disqualifies rule 1 (Equation 4.1, illustrated in Figure 4.2) for implementation in the context of implementation diversity given that all possible transformation outcomes are equivalent when not distributing over time. Furthermore, the approach of arbitrarily splitting the folded data structure in chunks that need not be equal in size is easily applied to rule 2 (Equation 4.2, illustrated in Figure 4.4), whereas this is not quite self-evident in case of rule 3 (Equation 4.3, illustrated in Figure 4.5). As such, it was decided to implement transformation rule 2 as a proof of concept.

4.4.2 Implementation on TH expressions

This section presents an implementation of the left fold transformation transformation rule 2 (Equation 4.2, illustrated in Figure 4.4) from Section 4.2. This is without distribution over multiple clock cycles, as was argued in Section 4.4.1. The implementation is presented in Listing 4.5 and will be explained piece by piece below.

```haskell
tfFoldl :: Exp -> (Maybe [Exp])
tfFoldl (AppE (AppE (AppE (VarE ((== 'foldl) -> True)) f) init) vec)
  | lengthTH vec > 1
    && (    (f == VarE '(+)      && init == LitE (IntegerL 0))
        || (f == VarE '(*)      && init == LitE (IntegerL 1))
        || (f == VarE 'and      && init == LitE (IntegerL 1))
        || (f == VarE 'and      && init == ConE 'True)
        || (f == VarE 'or       && init == LitE (IntegerL 0))
        || (f == VarE 'or       && init == ConE 'False)
        || (f == VarE 'xor      && init == LitE (IntegerL 0))
        || (f == VarE 'xor      && init == ConE 'False)
        || (f == VarE 'xor      && init == VarE 'zeroBits)
        || (f == VarE 'mappend && init == VarE 'mempty)
        || (f == VarE '(<>)     && init == VarE 'mempty)
       ) = Just $ foldlTH fNew init . listToVecTH' <$> foldedVariants
  | otherwise = Nothing
 where
  fNew = AppE (VarE 'keep) f
  chunkVariants = splitPlacesTH <$> termsOfSum (lengthTH vec) <*> [vec]
  foldedVariants = ( \v -> if lengthTH v == 1
                           then headTH v
                           else myFoldlTH f init v
                   ) <<$>> chunkVariants
  (<<$>>) = fmap . fmap
tfFoldl x = Nothing
```

Listing 4.5: *The left fold transformation primitive. Uses functions that are provided in Appendix C.*

A left fold function call as a TH expression Template Haskell (TH) allows modification of Haskell code (and thus also Clash hardware descriptions) by operating on an abstract syntax tree (AST) representation of them. Any left fold in a clash description is a function call, and any function call is part of an expression. Any Template Haskell representation of an expression is of type `Exp` as exported by module `Language.Haskell.TH`. Specifically, a call to `foldl` from the Clash Prelude module looks as shown in Listing 4.6 and illustrated in Figure 4.6. Here, `f` represents the reduction function, `z` the initial value, and `xs` a vector that is to be folded.

```
AppE (AppE (AppE (VarE Clash.Sized.Vector.foldl) f) z) xs
```

Listing 4.6: *A fully applied function call to `foldl`.*

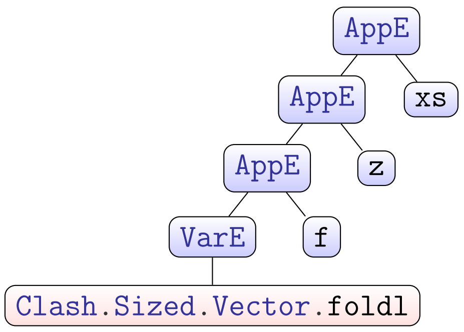

Figure 4.6: *A visualization of a fully applied function call to `foldl` as a tree.*

Transformation primitives Transformation rule transformation implementations like this one will be referred to as *transformation primitives*. Such primitives need to be of the type `Exp -> Maybe [Exp]`. On receiving a TH expression, they must return a list of variants wrapped in a `Just` when the rule fired, and `Nothing` otherwise. The extra `Maybe` in the return value type is because of implementation reasons discussed in Section 4.5.2.

Detecting eligible left fold function calls Unlike data constructor functions, regular Haskell functions like `foldl` cannot be pattern matched on. So in order to detect such left folds, a fully applied function call as a TH `Exp` like in Listing 4.6 is instead matched on.

A call to `foldl` must meet a few criteria to be eligible. As it isn't straightforward to determine whether a given reduction function is associative and whether it has an identity element, only a select few that are known fulfil these requirements are allowed. This is checked in Listing 4.5 lines 4 to 13. Furthermore, it serves no purpose to consider folding any vector of length zero or one. As such, sufficient lengthiness is asserted in Listing 4.5 lines 3. This used function `lengthTH` as defined in Listing C.3.

Constructing new left folds The original vector must be split in as many different as ways as possible to construct the different variants. In line 18 of Listing 4.5, function `termsOfSum` from Listing 4.4 is used to determine how that must be done, and `splitPlacesTH` from Listing C.8 is then used to split the original vector that way.

At line 19 to 22 of Listing 4.5, each of the resulting vector chunks is kept as is when of length one and wrapped in a new call to `foldl` when of greater length. This is done using `foldlTH` from Listing 4.7. The original reduction function is used with the addition of a `keep` attribute, the need of which is discussed in Section 4.4.3. The original initial value is used as well. Finally, the folded chunks are transformed into a vector using `listToVecTH'` from Listing C.9 and passed in a final call to `foldl` at line 14.

```
foldlTH f init vec = AppE (AppE (AppE (VarE 'foldl) f) init) vec
```

Listing 4.7: *Constructs a new fully applied function call to `Clash.Prelude.foldl`.*

Note that the variants resulting from an application of this transformation primitive may be transformed again. This is accomplished through its repeated application as accomplished in Section 4.5.

Note that the above implementation operates solely on the Template Haskell representation of Clash expressions, which can be a bit clunky. Alternatively, it could have used a calls to functions operating on the regular Haskell representation of code. Two examples using this approach are provided in Listing 4.8. They can be deployed in a more versatile manner, but they can be harder to understand and write due to having to deal with the `Q` monad and `Exp` return values all the time. Implementations like these have been experimented with, but could not be finished in time. They would still need to use the current approach in part, because there is no way of splitting vectors like `splitPlacesTH` from Listing C.8 does due to vectors having their lengths encoded in their types.

```
lengthTH2 :: Exp -> ExpQ
lengthTH2 v = do
  length' <- [| length |]
  return $ length' `AppE` v

lengthTH3 :: Exp -> Exp
lengthTH3 v = (VarE 'length) `AppE` v
```

Listing 4.8: *Alternative approaches to determining the length of a vector compared to Listing C.3.*

4.4.3 Forcing EDA tools to keep structure

EDA tools, particularly those for FPGAs, have many optimization steps in pursuing minimal hardware resource usage, power usage and propagation delay. It was observed that Xilinx Vivado synthesis optimizations transformed the variants returned by the left fold transformation primitive from Section 4.4.2 such that there did not appear to be any difference between the synthesized variants. This is troublesome as equivalent "variants" do not lead to implementation diversity. Disabling certain synthesis optimizations could work around the issue but is undesirable in general as that would negatively affect the entire user design.

Vivado can be instructed to leave certain signals be that would otherwise be absorbed into logic blocks or otherwise pruned [41]. Such signals should be annotated by the `keep` attribute to force their keeping during synthesis, or by `dont_touch` to additionally force their keeping during implementation.

Clash allows for the annotation of signals with HDL user attributes. However, the feature is experimental, and limited testing showed it working improperly more often than not[2]. As a workaround for this bug, a Clash HDL primitive was written. It emits a given attribute when compiled to VHDL.

Clash primitives do not seem to support insertion of code in the declaration section (i.e. before the `begin` keyword) of a VHDL architecture statement — the location where the VHDL syntax dictates attributes to go. To work around this fact, the new primitive declares an extra signal in a VHDL block statement.

4.5 Generic transformations through AST traversal

It is useful to be able to limit the scope of a user's design where a transformation is applied. With regards to protection against certain physical attacks through implementation diversity, only the sensitive logic may need to be considered. In that given scope, the transformation should be applied wherever possible. And in case of transformations that may yield multiple variants, all variants need to be accumulated.

This section introduces a generic solution for these problems by making *variant transformations*. They can be made to work with any given transformation primitive that works on AST representations of code, such as the one worked out in Section 4.4. Section 4.5.1 first considers existing approaches to AST traversal and serves as a prelude to Section 4.5.2 where a new approach that supports transformation primitives is introduced. It traverses an AST, tries to apply the transformation primitive wherever possible and accumulates all variants. Section 4.5.3 finally deals with obtaining the AST representation of code in the first place and with putting any resulting variants back in as new function or variable declarations.

4.5.1 Scrap your boilerplate

Template Haskell operates on TH AST representations of user code. Transformations on such complex data structures quickly induce lots of code that solely performs traversal. Such *boilerplate* code can be rid of using the freely available libraries discussed in this section. It serves as a background introduction for the next section, 4.5.2.

Let us take the small arithmetic language from Listing 4.9 as a running example. `Expr` expresses integer values contained by `Val` and operations on them as an AST. For example, $4-2$ may be represented as `Sub (Val 4) (Val 2)` in our new language as illustrated in Figure 4.7. Suppose that we wish to simplify our language by removing `Sub`. After all, any subtraction can be expressed with the help of addition and negation. Function `f` from Listing 4.10 applies this transformation

[2]The Clash compiler issue report that details the issue: `https://github.com/clash-lang/clash-compiler/issues/1082`

through pattern matching given an existing Expr. If it matches Sub, the substitution is applied, otherwise the original Expr is returned.

```
data Expr = Val Int
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Neg Expr
          deriving (Show, Eq, Data)
```

Listing 4.9: *A simple arithmetic language.*

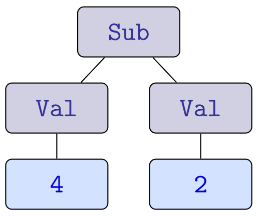

Figure 4.7: *AST for $4-2$.*

```
f :: Expr -> Expr
f (Sub x (Neg y)) = Add x y  -- Substitute x - (-y) with x + y
f (Sub x y) = Add x (Neg y)  -- Substitute x - y with x + (-y)
f x = x
```

Listing 4.10: *Rewrite when given a subtraction node, otherwise return the argument untouched.*

When applying f to our example subtraction AST, it neatly returns Add (Val 4) (Neg (Val 2)) as one would expect. Unfortunately though, f only applies its transformation on subtractions at the *root* of an AST and its usability is therefore quite limited. To resolve this, f could be rewritten to Listing 4.11's f' to call itself recursively on all recursive data constructor fields: its *children*. Code like this works perfectly well but is somewhat undesirable to write and maintain, particularly for larger data structures with many constructors such as the Template Haskell AST. There is a lot of added *boilerplate* code for traversal that has nothing much to do with the transformation's intent of altering only a small subset of a data structure. It can be error-prone to write and likely needs to be changed when the data type changes, like is the case with the Template Haskell AST because of its regular backwards incompatible changes between major GHC releases.

```
f' :: Expr -> Expr
f' (Sub x (Neg y)) = Add (f' x) (f' y)  -- Substitute and recurse
f' (Sub x y) = Add (f' x) (Neg (f' y))  -- Substitute and recurse
f' (Add x y) = Add (f' x) (f' y)        -- Recurse on Expr fields
f' (Mul x y) = Mul (f' x) (f' y)        -- Recurse on Expr fields
f' (Neg x)   = Neg (f' x)               -- Recurse on Expr fields
f' x         = x
```

Listing 4.11: *Traverse the recursive data type to try and apply a transformation on any arbitrarily deep node.*

Scrap Your Boilerplate (SYB) is a Haskell library providing *generic programming* [42]. It enables generic traversals and queries on data structures by taking care of traversing over the (possibly recursive) fields of data constructors. For the sake of simpler terminology, the rest of this thesis regards *tree* data structures specifically as the TH AST is the intended field of application. For a data type to be supported, it needs to be an instance of the `Data` typeclass, which can be derived automatically when enabling the DeriveDataTypeable GHC extension[3].

Uniplate is another Haskell library that has similar goals to the SYB work but is simpler and faster [43]. It can be used easily through automatically derivable `Data` instances from SYB, but manual `Uniplate` or `Biplate` typeclass instances may alternatively be written for significantly higher performance. Applying Uniplate's `transform` to `f` yields a function with the same functionality as `f'`. It traverses a given data structure and applies `f` to the root of an AST and all its children.

4.5.2 Variant transformations

Libraries such as Uniplate provide powerful and concise means to generic transformations of, for example, ASTs. They do, however, not support transformations that may return multiple variants. This section proposes an implementation that does just that without sacrificing the simplicity of only having to pattern match on the constructors that are of interest like with Listing 4.10's function `f`.

Transformations that traverse a given data structure to apply a transformation rule wherever possible while accumulating all resulting variants will be referred to as *variant transformations*. A data type for them is provided in Listing 4.12. To run a variant transformation, it should be passed to accessor function `runT` to obtain function of type `a -> [b]` that is contained within. Multiple variant transformations can be composed using the `(||>)` operator as defined in Listing 4.13.

```
newtype Transformation a b = Transformation {runT :: a -> [b]}
```

Listing 4.12: *A no-op newtype wrapper for variant transformations so that typeclass instances could be declared.*

```
(||>) :: Transformation a b -> Transformation b c -> Transformation a c
tf1 ||> tf2 =
  Transformation $ \x ->
    let ys = runT tf1 x
    in  concat (runT tf2 <$> ys)
```

Listing 4.13: *An operator for composing variant transformations.*

Variant transformations can be constructed using function `mkT` from Listing 4.14. As can be seen, it operates on types that are instances of both the `Eq` and `Data` typeclasses.

[3]Data can be automatically derived for user-provided types: The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.2. The GHC Team. `http://www.haskell.org/ghc/docs/6.12.2/html/users_guide/`.

```haskell
mkT
  :: forall a. (Eq a, Data a)
  => (a -> Maybe [a])
  -- ^ The transformation primitive
  -> Transformation a a
mkT tf =
  Transformation $ \x ->
    case go Nothing (contexts x) of
      Nothing  -> [x] -- If the transformation resulted in no variants,
       ↪  just return the original value.
      Just out -> nub out -- `nub` shouldn't be needed, but may help prune
       ↪  duplicates at times.
 where
  go
    :: (Eq a, Data a)
    => Maybe [a]
    -- ^ Accumulation of variants, if any.
    -> [(a, a -> a)]
    -- ^ Elements and corresponding contexts.
    -> Maybe [a]
    -- ^ Variants, if any. Nothing otherwise.
  go acc [] = acc
  go acc ((element, context):cs) =
    case tf element of
      Nothing ->
        go acc cs -- Transformation didn't match, check the next element.
      Just vs ->
        let redf :: (Eq a, Data a) => Maybe [a] -> a -> Maybe [a]
            redf acc' v =
              if v == element
              then go (Just [context element]) cs <> acc'
              -- ^ If one of the variants returned by a transformation
               ↪  primitive is its input, don't transform that variant
               ↪  again as that would result in an infinite loop with the
               ↪  variants being duplicated over and over.
              else Just ((runT $ mkT tf) (context v)) <> acc'
        in  foldl' redf Nothing vs <> acc
```

Listing 4.14: *An approach to making variant transformations given a transformation primitive.*

Transformation primitives As its first argument, `mkT` takes a function of type `a -> Maybe [a]` that will be referred to as a *transformation primitive*. Such a primitive should match at the root of an AST of type `a` and must return either a list of resulting variants wrapped in a `Just` when it wishes to change something in the AST or add any variants, or a `Nothing` otherwise. An example of a transformation primitive is the left fold transformation rule implementation discussed in Section 4.4.2.

When the primitive application returns an empty list wrapped in a `Just`, the variant is pruned as an empty list is returned. When the primitive application returns a singleton list (i.e. with a single element) wrapped in a `Just`, this is like doing a regular transformation; no variants are added or pruned. There are a few requirements on what primitives may and may not do as discussed in the paragraph titled drawbacks below.

The use of `Maybe` was opted for for performance reasons. The implementation discussed below requires knowledge of whether a transformation primitive matched, and it is much cheaper to check for a `Nothing` than to check whether there are any differences between the trees given to and returned by a transformation primitive.

Context is key The variant transformation that `mkT` returns is a wrapper around the actual transformation function of type `a -> [a]`. Of the tree that is the argument, all subtrees and their *contexts* are determined by calling Uniplate's `contexts` function, hence the `Data` contraint on `a`. All subtrees and their contexts of Figure 4.7's AST for $4-2$ are illustrated in Figure 4.8. A subtree's context is the original tree with a *hole* in place of the subtree. One obtains the original tree when plugging the subtree into the hole of its context.

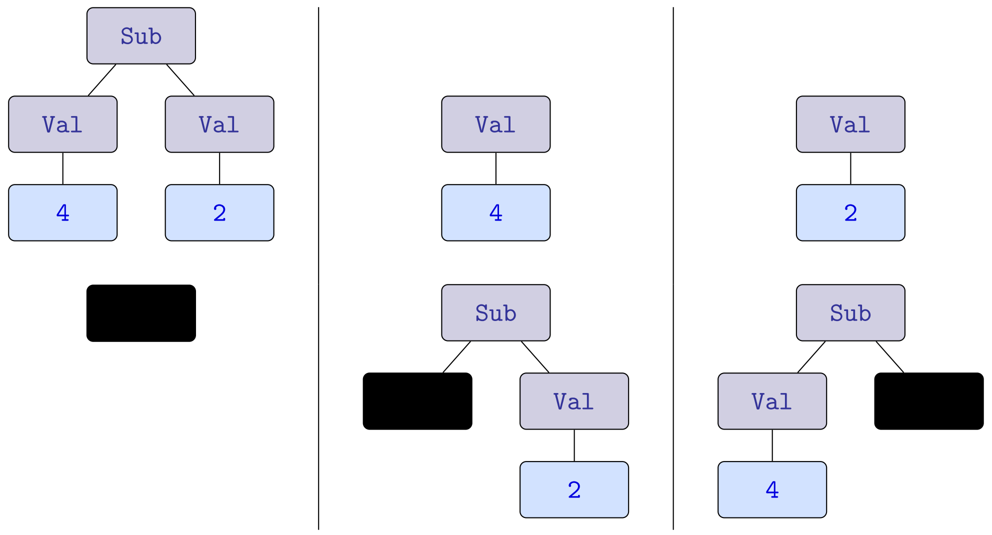

Figure 4.8: At the top all three subtrees for the AST of $4-2$ in Listing 4.9's language, at the bottom their corresponding contexts with a hole in place of the subtree.

Note that 4 and 2 are not considered to be subtrees. This is because Uniplate functions only perform single type traversal, in this example of `Expr` values and their `Expr` fields. The numbers are fields of a different type, namely `Int`. Multi-type traversal is possible using `Biplate` and its functions, but it was not needed and therefore out of the project scope.

Traversal `mkT` traverses a given tree by applying a given transformation primitive to all possible subtrees. When the primitive application returns a `Nothing`, it continues with the next subtree. Otherwise — when it returns a list wrapped in a `Just` — each element is put into the proper context and passed in a recursive call to `mkT`. There is one exception: when an element is found to be equivalent to the subtree passed to the primitive, it puts it into context, adds it as a variant and continues with the next subtree. Passing it in a recursive call to `mkT` would result in infinite recursion. This equality check gave rise to the `Eq` constraint on a. All results are concatenated and returned.

Drawbacks and limitations The fact `mkT` recursively calls itself on any variants returned by the transformation primitive that are not equivalent to the primitive input makes that it will recurse on any such variants until the primitive does not match on them any more and returns a `Nothing`. To avoid infinite recursion, it is therefore important that the primitive will not match on variants after a finite number of applications any more. Note what this implies:

- Transformation primitives should not contain mappings that are each other's inverse. For example, if some element A is transformed to some other element B, the primitive may not also transform B to A.
- Transformation primitives should not simply add depth to a given tree, for example by wrapping it in an identity node. They should either:
  - return reduced variants (like a left fold on a smaller vector than the input as happens with the fold left transformation primitive discussed in Section 4.4.2), or
  - return a variant that does not match any more (for example by replacing a matching call to `foldl` with a non-matching one to `myFoldl` that is declared as `myFoldl = foldl`).

4.5.3 Splicing variants as new declarations

This section answers two questions. How can a variant transformation be applied to a given section of a Clash hardware design? And how can the resulting variants be used as regular Haskell in the rest of the design, for example in adding implementation diversity?

A solution that answers these two questions involves a single Clash declaration of a variable or function. It comprises the function `varDecs` as defined in Listing 4.17.

Its use is demonstrated in Listing 4.15 and results in the log of Listing 4.16 being printed when compiled.

```
$(varDecs
   (mkT tfFoldl)
   [d| fun = foldl (+) 0 ((1::Unsigned 8):>2:>3:>4:>5:>Nil) |]
 )
```

Listing 4.15: *An example application of the left fold variant transformation on a variable declaration.*

```
[INFO] Left fold transformation rule fired.
[INFO] Number of variants generated: 15.
[INFO] Variant declaration names are "fn_0" through "fn_14".
```

Listing 4.16: *Part of the Clash compilation log of the code of Listing 4.15.*

Function `varDecs` takes a variant transformation of type `Transformation Exp Exp` and the Template Haskell representation of a single Clash function or variable declaration. The latter is of type `DecsQ`, a type synonym for `Q [Dec]`. As shown in the example, such a `DecsQ` can be obtained by *quotation* of a declaration using the TH quotation brackets. The declarations returned by `varDecs` can be *spliced* to obtain their regular Haskell representation.

At lines 4 through 9 of Listing 4.17, the input declaration is deconstructed into the declaration name, its arguments in case of a function declaration, its body and any potential where clause contents. Line 12 asserts that there is no where clause, as its content would not be considered when applying the variant transformation.

Line 17 applies the variant transformation wherever possible in the body of the declaration. For any resulting variants, a new declaration name is constructed at line 19. These names are constitute the original input declaration name appended with an underscore and a number between 0 and n where n is the number of variants minus one.

Starting at line 24, helper function `mkVariant` is declared for constructing new declarations for the variants. It supplements such new declarations for the variants with a corresponding `NoInline` pragma and a `Synthesize` annotation. This makes that the new variant declarations will be compiled as a separate component, something particularly useful when the variants will be partially reconfigured on an FPGA as discussed in Chapter 5. The helper function is used at line 37 to construct the new variant declarations using a parallel list comprehension.

One additional declaration is defined at line 40: a list of all variants declarations. This allows easy verification of the apparent functional equivalence of the variants. Its evaluation for the example of Listing 4.15 looks as follows:

```
> fun
[15,15,15,15,15,15,15,15,15,15,15,15,15,15,15]
```

```haskell
varDecs :: Transformation Exp Exp -> DecsQ -> DecsQ
varDecs tf inputDecsQ = do
  inputDecs <- inputDecsQ
  let (name, mArgs, expr, whs) =
        case inputDecs of
          [FunD name [Clause args (NormalB expr) whs]]
            -> (name, Just args, expr, whs)
          [ValD (VarP name) (NormalB expr) whs]
            -> (name, Nothing, expr, whs)
          _ -> error "[ERROR] This function (varDecs) only supports a "
                  <> "single variable or function declaration as input!"
  when (notNull whs) $
    fail "[ERROR] This function (varDecs) does not support a declaration "
      <> "with a where clause! Try and use let bindings instead. Note: "
      <> "this check could be savely removed, but then no variants would "
      <> "be generated for declarations in the where clause."
  let variantExps = (runT tf) expr
  traceM $ "[INFO] Number of variants generated: " <> show (length variantExps) <> "."
  let variantNames = [ mkName $ nameBase name ++ '_' : show i
                       | i <- [0..(length variantExps)-1] ]
  traceM $ "[INFO] Variant declaration names are \""
        <> nameBase (head variantNames) <> "\" through \""
        <> nameBase (last variantNames) <> "\"."
  let mkVariant name mArgs expr whs = [fun, pragNoInl, pragAnnSyn]
       where
        fun = case mArgs of
          Just args -> -- It's a function declaration
            FunD name [Clause args (NormalB expr) whs]
          Nothing -> -- It's a variable declaration
            ValD (VarP name) (NormalB expr) whs
        -- NoInline pragma to ensure Clash picks up on the TopEntity annotation below
        pragNoInl = PragmaD $ InlineP name NoInline FunLike AllPhases
        -- TopEntity annotation pragma so that Clash compiles the declaration as a separate component. Required so that it can be used as a partial module in a design using partial FPGA reconfiguration.
        pragAnnSyn = PragmaD $ AnnP
          (ValueAnnotation name)
          (AppE (VarE 'defSyn) (LitE $ StringL $ nameBase name))
  let variantDecs = [ mkVariant name mArgs expr whs
                      | name <- variantNames
                      | expr <- variantExps ]
  let variantNamesDec = ValD (VarP name) (NormalB $ ListE $ VarE <$> variantNames) []
  return $ variantNamesDec : concat variantDecs
```

Listing 4.17: *Apply a variant transformation to a given function or variable declaration and return the declaration for any resulting variants.*

5 Partial reconfiguration using Clash

In this chapter, the aim is to investigate *how* one can realise a *DPR-enabled FPGA design* that can also be simulated, all using Clash. Main research questions 4 and 5 will be answered in doing so. The core components needed are identified and implemented in a generic manner, largely independent of any one FPGA vendor. The results are combined in a library, allowing one to easily attain a Clash-based design flow featuring DPR. Its full use would result in a design resemblant of Figure 5.1, an overview that may be useful to refer back to while reading this chapter. It could be used to realise a design with implementation diversity using the variants generated through the techniques discussed in Chapter 4.

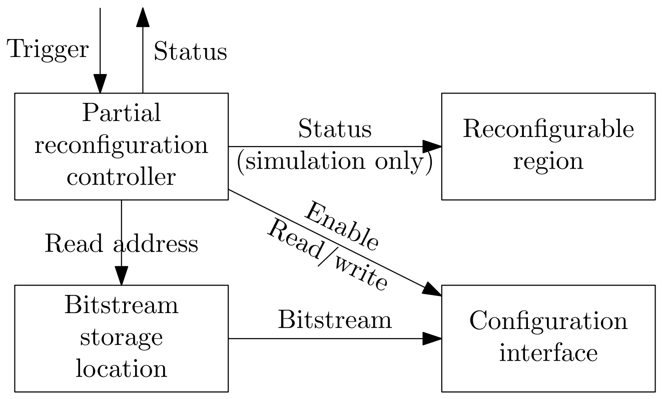

Figure 5.1: *A schematic overview of what a DPR design using the DPR library in this thesis would look like.*

Partial bitstreams are used to partially reconfigure an FPGA. A designer must designate the reconfigurable modules of a design in order to have EDA tooling generate the corresponding partial bitstreams. Approaches to this are considered in Section 5.1.

An EDA tool must furthermore be notified of any reconfigurable region: the section of FPGA fabric that is to alternately implement a number of reconfigurable modules. Section 5.2 considers a way of grouping these modules and introduces a design element that truly represents a reconfigurable region. This design element is special in that it can *simulate the process of undergoing reconfiguration* in the Clash simulator. Like on a real FPGA, modules can be exchanged, and the signals going into the static region carry undefined values during that process. This novel implementation enables cycle-accurate simulation of DPR designs for the express purpose of functional verification.

Partial reconfiguration is not something that happens automatically. When the situation calls for different functionality in runtime, the partial bitstream corresponding to the module that implements that new functionality needs to be retrieved from some storage location and fed to one of the FPGA's configuration interfaces. Section 5.3 introduces a simple yet effective controller for managing these tasks.

The final component needed for implementing a self-DPR design is a configuration interface to the FPGA. Section 5.4 introduces a Clash primitive for instantiating the ICAP, an internal configuration interface for Xilinx devices.

In Section 5.6, a simple hardware design is worked out to check that all DPR-related work from this chapter works properly and together. After implementing it on an FPGA, measurement and simulation waveforms are compared.

5.1 Defining reconfigurable modules

In order to deploy an FPGA design using partial reconfiguration, an EDA tool such as Vivado must produce a full bitstream containing the default configuration, and one partial bitstream for each reconfigurable module in the design. For it to do that, it requires the design sources for the static region and all reconfigurable modules.

To realise a DPR workflow with Clash, the proper design sources need to be provided by the Clash compiler. Clash produces self-contained HDL sources in the user-selected target HDL, i.e. VHDL or (System)Verilog, for any *top-level* function declaration. Such declarations must meet at least one of two conditions: the function must be named `topEntity` or have a `TopEntity` annotation. This is illustrated in Listing 5.1. Each reconfigurable module therefore needs to be represented as a function declaration in a Clash design. Composition of simple functions allows for more complex designs.

```haskell
-- < module definition and imports >

topEntity = ...

{-# ANN foo (defSyn "bar") #-}
foo = ...
```

Listing 5.1: *A Clash module that would, when completed, compile to two modules named* topEntity *and* bar*.*

5.2 Defining reconfigurable regions & DPR simulation

This section proposes a way of grouping reconfigurable modules in a design element representative of a reconfigurable region that supports simulation of undergoing DPR.

5.2.1 Definition

A reconfigurable region represents a physical portion of FPGA fabric that can be reconfigured to alternately implement a number of reconfigurable modules. From the static region's point of view, it can be considered a single module with a certain functionality that happens to change over time due to the process of partial reconfiguration. It therefore makes sense to have functions represent reconfigurable regions. Such functions should return something of the same type that the reconfigurable modules that the regions can implement have. Combined with the fact that the Clash compiler requires top-level functions to be monomorphic for them to be compilable, this makes that all reconfigurable modules in a region should have same type.

A custom data type is introduced in Listing 5.2 with the goal of capturing the reconfigurable region related information required to realize a partially reconfigurable design. It encapsulates a vector of three-tuples where each first tuple element is a function implementing a reconfigurable module in the region. The second and third elements of each tuple are the corresponding partial bitstream start address and size. These indicate where a partial reconfiguration controller may retrieve the relevant partial bitstreams as elaborated upon further in the next section, 5.3. Type variable `n` is the type-level natural number representing the number of reconfigurable modules in the region while `f` is the type of the module functions. Their type equality is enforced by the vector being homogeneous.

```haskell
data Region n f
  = Region (Vec n (f, BitstreamAddr, BitstreamSize))
type BitstreamSize = Unsigned 32
type BitstreamAddr = Unsigned 32
```

Listing 5.2: *The custom data type introduced to contain the settings regarding a reconfigurable region and its modules. `BitstreamSize` and `BitstreamAddr` are type synonyms for a 32-bit unsigned number.*

The only other major piece of information left regarding a reconfigurable region is its physical footprint on the FPGA fabric. As this information is not required at the Clash level of abstraction and as there is no standard way to pass such information to an FPGA EDA tool like Vivado, it is not included in Listing 5.2's data type. Designating the region footprint is left as a manual step during the floorplanning phase of the FPGA EDA toolflow.

5.2.2 Implementation

The function introduced to represent a reconfigurable region is presented in Listings 5.3. Its role is twofold: it needs to be able to mimic the act of reconfiguration when simulated, whereas it needs to result in a format usable by EDA tools make the design partially reconfigurable when compiled. This difference in behaviour calls for use of a HDL primitive.

Function compilation To have `region` compile to the HDL sources required by FPGA EDA tools to enable partial reconfiguration, it uses the region definition that is its first argument to extract the function that is the first reconfigurable module. That module function is passed fully applied in a call to `region'`, which has a HDL primitive annotation. This primitive only uses the applied module function, which makes that `region'` is compiled to VHDL as a component that is instantiated

```
region
  :: (KnownNat n, HiddenClockResetEnable dom)
  => (HiddenClockResetEnable dom => Region (n+1) (Signal dom a -> Signal
   ↪  dom b))
  -- ^ The region definition that this function implements
  -> Signal dom (RegionSt (n+1))
  -- ^ The region state denoting some module being active or the region
   ↪  being undergoing reconfiguration. For simulation purposes only.
  -> Signal dom a -> Signal dom b
region r@(Region ((f0,_,_):>_)) pRegSt inp
  = region'
      f0Outp
      (fst3 <$> fromRegion r) -- The reconfigurable modules in the region
      (Reconfiguring undefined)
      pRegSt
      hasClock
      hasReset
      hasEnable
      inp
      (pure undefined)
  where
    f0Outp = f0 inp
```

Listing 5.3: *The function that encapsulates a reconfigurable region.*

with the first reconfigurable module. As all modules in a region have the same type, all of them could be used to instantiate the VHDL component. This is the format required by Xilinx Vivado, Intel Quartus [44, p. 16], and potentially EDA tools of different FPGA vendors to make a design partially reconfigurable.

Function simulation The `region'` function definition of Listing 5.4 defines the DPR simulation behaviour. Simulation only, as how the function is compiled is defined in the function's aforementioned HDL primitive annotation.

To know when to simulate which module being active or PR being in progress, `region'` has a signal of `RegionSt n` as given in Listing 5.5 as argument. As illustrated in Figure 5.1, this status signal is to come from the PR controller, and is used in simulation only. It does not end up in hardware when compiled, as it is not used by the HDL primitive of `region'`.

Function `region'` furthermore has the aforementioned region definition from Listing 5.2 and the region's inputs as arguments. It returns the region's outputs. Those outputs are set to *undefined* during reconfiguration to represent that they can be random during that time, at least for Xilinx FPGAs [27].

Virtually all hardware designs operate on or return non-constant values. The concept of changing values on a wire is captured by values of type `Signal d a` in Clash. For the purpose of simulation, they can be constructed as infinite streams of values using the `(:-)` type constructor, an operator that prepends a value in front of another signal. Clash functions operating on signals describe the function input/output relation over time. As such, when a reconfigurable region has

```haskell
region'
  :: (KnownNat n, KnownDomain dom)
  => Signal dom b
     -- ^ The first module, fully applied, used by the HDL primitive only
  -> (HiddenClockResetEnable dom => Vec n (Signal dom a -> Signal dom b))
     -- ^ The modules in the reconfigurable Region
  -> RegionSt n
     -- ^ The RegionSt during the previous clock cycle
  -> Signal dom (RegionSt n)
     -- ^ The RegionSt during this and all following clock cycles
  -> Clock dom
     -- ^ The reconfigurable region clock input
  -> Reset dom
     -- ^ The reconfigurable region reset input
  -> Enable dom
     -- ^ The reconfigurable region enable input
  -> Signal dom a
     -- ^ The reconfigurable region input
  -> Signal dom b
     -- ^ The reconfigurable region output when the same module as in the
      ↪  previous clock cycle remains active. Undefined otherwise.
  -> Signal dom b
     -- ^ The reconfigurable region output
region' !f0Outp mods prevSt (st :- sts) clk rst en inps outpActs
  = head# outs :- region' f0Outp mods st sts clk rstNxt enNxt inpsNxt
   ↪  (tail# outs)
  where
    rstNxt = tailRst# rst
    enNxt = tailEn# en
    inpsNxt = tail# inps
    outs = case st of
      Reconfiguring x
        -> undefined :- undefined
      Active x
        -> if st == prevSt
             then outpActs
             else exposeClockResetEnable (mods !! x) clk rst en inps
```

Listing 5.4: *The helper function of* `region` *that determines reconfigurable region simulation and compilation.*

undergone PR, the function of the module that is then present needs to be initiated by calling it with the then current values of the input signals. This is implemented at the last line of the `region'` function definition. The module output signal values of the subsequent clock cycles are passed in a recursive call, to be used when the same module remains active (i.e. when the region state does not change). To have the relevant inputs ready for a new module function call after a potential future PR action, the clock and the consequent values (i.e. the tails) of the reset, enable and normal input signals are passed to the recursive call as well.

```haskell
data RegionSt n
  = Reconfiguring (Index n)
  | Active        (Index n)
```

Listing 5.5: *A custom data type used to indicate the PR status of a reconfigurable region.*

Notes on the reconfigurable module type As can be seen, `region` requires the modules in a reconfigurable region to be of the type `Signal dom a -> Signal dom b`, optionally with hidden clock, reset and enable inputs. Ideally, it would have a more general definition that supports module functions of any arity by using a single type variable for the module type.

First, the reconfigurable module type is constrained to `a -> b` due to one of the modules being used in the HDL primitive of `region'` to make it compile properly as elaborated upon above. The Clash compiler requires types used in HDL primitives, such as type variables `a` and `b`, to be representable in hardware, i.e. monomorphic and not functions. (There are exceptions, but those have hand-rolled reductions in the `reduceNonRepPrim` internal compiler transformation.) This limitation implies that `a -> b` needs to represent functions of arity one. This does not limit expressiveness in practice as the arguments of functions of a higher arity can be bundled together in a tuple by uncurrying. Alternatively, multiple `region` functions for reconfigurable regions with module functions of different arities could have been written.

Reconfigurable module types are furthermore constrained to `Signal dom a -> Signal dom b` to accommodate the signal deconstruction used by `region'` to enable simulation of PR designs with sequential circuits as reconfigurable modules. This does not limit expressiveness in synchronous designs as any combinational circuit can be expressed as a sequential one through applicative lifting.

5.3 Partial reconfiguration controller design

This section discusses the design of a partial reconfiguration controller (PRC) finite-state machine. A PRC is vital in designs using partial self-reconfiguration. On receiving a trigger for a specific reconfigurable module and region, it is to initiate and oversee the transfer of the corresponding partial bitstream from a bitstream storage location to an FPGA configuration interface. The PRC is itself controlled through its trigger inputs by a hardware state machine or software running on processor cores. How it may be interconnected with other components is illustrated in Figure 5.1.

PRCs can be implemented in software that runs on a soft- or hard processor core, or in hardware in the reconfigurable logic of the FPGA in question. The latter option was picked for this PRC implementation for no particular reason. The final PRC design is provided in Listing 5.6 and will be discussed below. It supports control of only one reconfigurable region at the moment.

```haskell
controller
  :: ( HiddenClockResetEnable dom, KnownNat n, KnownNat b)
  => Region n f
  -> ( Signal dom BitstreamAddr -> Signal dom (BitVector b) )
     -- ^ Address to bitstream, same clock cycle response is assumed
  -> Signal dom (Trigger n)
  -> ( Signal dom (RegionSt n)
     , Signal dom (Maybe (BitVector b)) -- ^ A partial bitstream
     )
controller r bsMem trig = (regionSt, mBitstream)
 where
  isReconfiguring (Reconfiguring _) = True
  isReconfiguring _ = False

  mBitstream = mux (isReconfiguring <$> regionSt)
                   (Just <$> bsMem rdAddr)
                   (pure Nothing)

  (rdAddr, regionSt) = mealyB controllerTF (ControllerSt (Active 0) 0) trig

  -- | PRC transfer function of mealy machine.
  controllerTF
    :: KnownNat n
    => ControllerSt n -- ^ State
    -> Trigger n -- ^ Input
    -> ( ControllerSt n -- ^ Next state
       , ( BitstreamAddr
         , RegionSt n
         ) -- ^ Outputs
       )
  controllerTF (ControllerSt regionSt counter) trigger
    = (ControllerSt regionSt' counter', (counter, regionSt))
   where
    (_, bsAddrs, bsSizes) = unzip3 $ fromRegion r

    (regionSt', counter') = case (trigger, regionSt, counter) of
      (Nothing, Active x,        _) -> (Active x,              undefined)

      -- Start reconfiguration when receiving a trigger, except when:
      --   - the relevant bitstream size is set to 0, or
      --   - the triggered module is already active.
      (Just tr, Active x,        _) -> if (bsSizes!!tr /= 0) && (tr /= x)
                                       then (Reconfiguring tr, bsAddrs!!tr)
                                       else (Active x,         undefined)

      (_,       Reconfiguring x, c) -> let c' = c + 1
                                       in if c' < (bsAddrs!!x + bsSizes!!x)
                                          then (Reconfiguring x,  c')
                                          else (Active x,         undefined)
```

Listing 5.6: *A Clash hardware design for a partial reconfiguration controller.*

PRC in- and outputs The `controller` function requires a `Region` definition for the bitstream addressing and size information stored within. It furthermore takes a function representative of a bitstream storage location. This function is to return the bitstream data at a given address. A response in the same clock cycle is assumed at this revision of the PRC. Finally, `controller` has a trigger signal as an input argument, `Trigger` is defined in Listing 5.7.

```
type Trigger n = Maybe (Index n)
```

Listing 5.7: *A type synonym for PRC triggers.*

The controller provides feedback through an output signal that shows the status of the reconfigurable region that it controls. This indicates the reconfigurable module that is currently active or being configured. Furthermore, the bitstream that is to be sent to an FPGA configuration interface is returned as wrapped in a `Just` when configuration should be in progress, or `Nothing` otherwise. This is defined at lines 15 through 17 of Listing 5.6.

Mealy machine The `controller` function contains a Mealy machine with as transfer function `controllerTF` as provided at lines 21 through 49. The Mealy machine state is stored in a `ControllerSt` as defined in Listing 5.8. It contains both the region state — which module is active or being configured — and an address counter for requesting a bitstream from a bitstream storage location, from a memory or over a bus.

```
data ControllerSt n
  = ControllerSt
      (RegionSt n)
      BitstreamAddr -- Address counter
    deriving (Show, Eq, Generic, NFDataX)
```

Listing 5.8: *A data type for storing the PRC mealy machine state.*

The next state is determined by the previous state and the PRC trigger input. Line 37 covers no trigger and a reconfigurable module being active: the module remains active and nothing changes. Line 36 covers a trigger coming in and a module being active: in the next state reconfiguration will start with the counter being set at the corresponding start address of the bitstream storage location. This happens *only if* the trigger does not match the module being active, and if the bitstream size for that module is not zero. Line 46 covers reconfiguration being in progress: triggers are ignored and reconfiguration ends when the partial bitstream has been sent in full.

Notes on tracking reconfiguration progress One major design consideration is how to determine when a partial bitstream has been fully sent. As discussed above, the start address and size as provided in a region definition are used to retrieve the bitstream from a storage location and to count until it has been sent in full. This approach has been picked because of its ability to be simulated true to the FPGA implementation and because of its independedness of any one FPGA vendor.

Alternatively, some configuration interfaces (such as the Xilinx Ultrascale ICAPE3) feature output signals indicating whether configuration is done [45, Table 7-8]. Such run-time information may be very useful for reliable operation of a partial reconfiguration controller. Unfortunately, accurate simulation of such signals is not possible, as FPGA vendors have not released accurate models. After all, the process of acquiring such a model that is reasonably fast to simulate is involved as it would need to be a simplified version of the FPGA's design itself. For example, the ICAPE3 simulation element from the vendor-provided UNISIM Vivado simulation library keeps its done and error signals constant. The only realistic fully accurate simulation of partial reconfiguration is one involving hardware-in-the-loop.

5.4 ICAP configuration interface

To perform partial reconfiguration, a partial bitstream needs to be fed to one of the FPGA's configuration interfaces. This section introduces a Clash HDL primitive for one such interface, the Xilinx ICAP, so that it may be easily instantiated in Clash hardware designs.

We work with Xilinx devices as these are extensively used in the PR related research at UT research group CAES[4]. The Xilinx Vivado Design Suite[5] is a proprietary HDL design logic synthesis and analysis tool for Xilinx FPGA devices. Its partial reconfiguration flow has been supported since it's predecessor, the ISE Design Suite. Users wishing to utilize partial reconfiguration on Xilinx FPGA's need to utilize the vendor's synthesis tools to do so. Though documenting the configuration bitstream format – a prerequisite for developing alternative synthesis tools – is an ongoing effort by the open source community in projects such as SymbiFlow[6], Xilinx's tools are the only fully tested and functional tools supporting PR on their devices as of yet.

FPGAs typically have a range of configuration interfaces supporting different combinations of protocols and bus widths. Xilinx User Guide UG470 [29] documents many configuration specifics of their 7 Series FPGAs. The document focusses on the supported *external* configuration interfaces — serial, SPI, BPI, SelectMap and JTAG — but also has a few specifics on the *internal configuration access port* (ICAP) and *processor configuration access port* (PCAP). The former is, as the name suggests, an *internal* interface, that is: accessible from within the FPGA fabric to ease self-reconfiguration. The latter is present only in the Zynq SoC series of Xilinx devices where it is accessible from the on-die hard ARM processor cores accompanying the reconfigurable logic. This research is not limited to Zynq devices

[4]https://www.utwente.nl/en/eemcs/caes/

[5]https://www.xilinx.com/products/design-tools/vivado.html

[6]https://symbiflow.github.io/

and focusses on self-reconfiguration from within the FPGA fabric, and therefore on the ICAP.

The ICAPE2 design element is the ICAP in Xilinx 7 series FPGAs and Zynq-7000 SoCs. Its interface is illustrated in Figure 5.2. The Clash function introduced to instantiate the ICAPE2 is provided in Listing 5.9. To accomplish that instantiation, it has a HDL primitive based on a template from Xilinx. The function's `IcapEn` input is a custom data type introduced in Listing 5.10 that encapsulates the ICAPE2's CSIB active-low enable line and RDWRB input that selects between reading from or writing to mode. The type is annotated with a custom bit representation so that the two bits representing its values can be used by the HDL primitive directly.

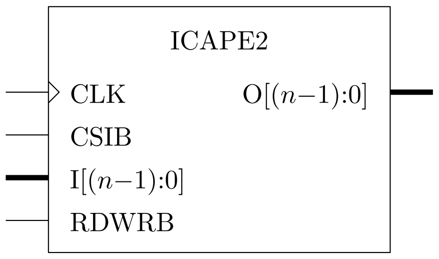

Figure 5.2: *The ICAPE2 interface where n is the data width in bits.*

```
icape2
  :: ( KnownDomain dom
     , (n == 8 || n == 16 || n == 32) ~ 'True
     )
  => Clock dom
      -- ^ 1-bit input: Clock Input
  -> Signal dom IcapEn
      -- ^ 2-bit input:
      --    - MSB is Active-Low ICAP Enable,
      --    - LSB is Read/Write Select input (0=write, 1=read)
  -> Signal dom (BitVector n)
      -- ^ Configuration data input bus
  -> Signal dom (BitVector n)
      -- ^ Configuration data output bus
icape2 !_ !_ !_ =
  pure $ error "This ICAPE2 simulation model does not support reading "
            <> "the FPGA configuration memory from it."
```

Listing 5.9: *Clash ICAPE2 function.*

The ICAPE2 further provides a configurable 8, 16 or 32 bits wide parallel input for providing (partial) bitstreams and an equally wide output for reading from the configuration memory. The latter may be used to verify configuration correctness or to determine the state of user state elements such as registers. Those specific use cases are not specific to nor required for partial reconfiguration, therefore readback simulation in Clash is not implemented by the `icape2` function. Note that this function is one of an uncommon few where one may not be interested in

```haskell
data IcapEn
  = IcapIdle
    -- ^ Inactive
  | IcapRead
    -- ^ Read from the ICAP
  | IcapWrite
    -- ^ Write to the ICAP
  deriving (Show, Eq, Generic, NFDataX)
```

Listing 5.10: *The custom data type used to represent the ICAP mode of operation.*

its return value, but only in its side effect when implemented in hardware (in this case: reconfiguration). Clash sees this as a call to a function where the result is never used, i.e. as one that may be left unevaluated. To force it to be rendered in hardware, one should take care to pass the function's return value to `hwSeqX`, which should in turn be (indirectly) used at the top of a design.

The parallel FPGA configuration interfaces automatically determine the data width in use by observing how the bus width auto detection words present in each configuration bitstream are received [29, p. 80]. Note that bitstream contents are independent of the used configuration interface and its bus width.

An additional function with a different type is provided for easier incorporation of the ICAPE2 in FPGA designs using self-DPR. Listing 5.11 provides function `icape2Write` which only allows writing to it by feeding it a bitstream wrapped in a `Maybe`. Depending on it being a `Nothing` or a `Just`, the enable is toggled automatically.

```haskell
icape2Write
  :: ( KnownDomain dom
     , (n == 8 || n == 16 || n == 32) ~ 'True
     )
  => Clock dom
  -> Signal dom (Maybe (BitVector n))
  -> ()
icape2Write clk mbs = icape2 clk icapEn bs `hwSeqX` ()
 where
  icapEn = (\mbs' -> case mbs' of
             Just _  -> IcapWrite
             Nothing -> IcapIdle
           ) <$> mbs
  bs = fromJustX <$> mbs
```

Listing 5.11: *ICAPE2 instantiation that can only be written to.*

The ICAPE2 expects bitstreams handed to it to have all individual bites reversed [29, p. 83]. Function `bytewiseReverseBV` from Listing 5.12 provides a function for accomplishing this bytewise reversal easily.

```
bytewiseReverseBV
  :: KnownNat l
  => BitVector (l * 8)
  -> BitVector (l * 8)
bytewiseReverseBV bv = v2bv $ concat $ reverse <$> (unconcat d8 $ bv2v bv)
```

Listing 5.12: *Takes a multiple-of-8 bit long bitvector and returns it with the bits in each individual byte reversed.*

5.5 Integrated logic analyzer IP core

When verifying or debugging FPGA designs, a logic analyzer can be very beneficial in triaging problems. It can be used to monitor the internal signals of a design at runtime. Xilinx Vivado is one of the EDA tools that provides such an *integrated logic analyzer* (ILA) IP core [46]. This section introduces a Clash HDL primitive for easy integration of the core in Clash designs.

The ILA IP core operates on detecting that a user-defined trigger condition is met. It then samples the signals connected to its probe inputs and stores them in an on-chip block RAM. From there they are sent to the Vivado analyzer over the FPGA's JTAG interface. How the core should be instantiated in a design depends on the amount of probes and their bit widths it is configured to capture. As an example, for two signals of bit widths 1 and 32, the instantiation in VHDL would look as depicted in listings 5.13 and 5.14.

```
component ila_0
port (
  clk    : in std_logic;
  probe0 : in std_logic_vector(0 downto 0);
  probe1 : in std_logic_vector(31 downto 0)
);
end component;
```

Listing 5.13: *An example ILA VHDL component declaration.*

```
ila_0_inst: ila_0
port map (
  clk    => clk,
  probe0 => probe0,
  probe1 => probe1
);
```

Listing 5.14: *An example ILA VHDL component instantiation.*

To avoid the cumbersome manual integration of the core instantiation in the compiled HDL output from Clash designs, a Clash HDL primitive may be used. Unfortunately, the ILA IP core's component port is not static, but depends on the amount of probes and their bit widths it is configured to support.

Polyvariadic functions can take an indefinite number of arguments that do not have to be of the same type. They can be realized in Haskell through the use of type classes, and sound like a good fit for a Clash ILA primitive. Unfortunately, they are inadequate as HDL primitives in Clash's templating system are static and cannot vary depending on e.g. polyvariadic function arity. Supposedly a new template system in the Clash compiler can support this, but that system is undocumented and barely used as of this writing.

Therefore, *Template Haskell* (TH) [40] was utilized for code generation. It allows for compile-time metaprogramming using *abstract syntax tree* (AST) representations of Haskell code. Such ASTs are built in the quotation monad `Q`. This monad is performed at compile-time when *spliced* to produce regular Haskell programs. See Appendix B for more information on Template Haskell.

Meta function `makeIla :: String -> [Int] -> DecsQ` was written to calculate the right ILA primitive. The first argument determines the name of the ILA function that is spliced in Clash and should match the ILA IP core name as configured in Vivado. The elements in the list that should be provided as the second argument represent the bit widths of the ILA probes. It returns a list of Template Haskell AST declarations wrapped in the `Q` monad as `DecsQ` is a type synonym for `Q [Dec]`. This list of declarations includes the ILA function type signature, function declaration, inline HDL primitive and NOINLINE pragma. The pragma is required to ensure that Clash picks up on an HDL primitive[7].

As an example, `makeIla "ila_0" [1,32]` splices in `ila_0` as defined in Listing 5.15. When used, it compiles to the VHDL component declaration and instantiation from listings 5.13 and 5.14. They are, however, wrapped in a VHDL block statement to work around the fact that Clash primitives do not seem to support putting component declarations in the declaration section (i.e. before the `begin` keyword) of a VHDL architecture statement.

```haskell
ila_0
  :: Clock dom
  -> Signal dom (BitVector 1)
  -> Signal dom (BitVector 32)
  -> ()
ila_0 !_ !_ !_ = ()
```

Listing 5.15: *The Haskell function declaration that results from `makeIla "ila_0" [1,32]`.*

Note that ILA functions are of an uncommon few where one is not interested in its return value — in this case it's even void — but only in its side effect when implemented in hardware. Clash sees this call to a function that always returns void as one that may be left unevaluated. To force it to be rendered in hardware, one should take care to pass the function's return value to `hwSeqX`, which should in turn be (indirectly) used at the top of a design.

[7]Source: `http://hackage.haskell.org/package/clash-prelude-1.2.4/docs/Clash-Annotations-Primitive.html#t:Primitive`

This Clash primitive supports the Xilinx ILA core v6.2 (and most likely other versions) in native mode only. Support for the core's trigger in- and outputs for interconnecting multiple ILAs is absent. It can be compiled to both VHDL and Verilog.

5.6 Putting it all together

In this section, a simple hardware design is worked out to check that all PR-related work from this chapter works properly and together. The general architecture as depicted in Figure 5.1 is used. The design features a single reconfigurable region with two modules: an incrementer and a decrementer. Over- or underflow occurring is the PRC trigger for configuration of the other module. FPGA ILA measurements and simulation traces are laid side by side to verify simulation being representative of the actual FPGA.

Reconfigurable modules and region The first reconfigurable module in this example design is `incrementer` as provided in Listing 5.16. The second module is exactly equivalent, except for the names (`decrementer` instead of `incrementer`, `subOUF` instead of `addOUF`) and one function call at line 10 (`sub` instead of `add`). The two modules respectively increment and decrement a stored signed integer number. The result is represented with one more bit than the input numbers. The two most significant bits are checked for inequality, which would indicate over- or underflow having occurred in the addition or subtraction. The resulting boolean and a truncated version of the addition or subtraction outcome are returned in a tuple.

The over- or underflow check at line 11 happens by packing the addition or subtraction outcome to obtain its `BitVector` representation, then casting that to a regular vector of bits and matching on the first two elements. Those are checked for inequality. A TopEntity annotation at lines 15 through 20 is required to have the modules compile to self-contained HDL sources in the user-selected HDL as explained in Section 5.1.

Listing 5.17 provides the reconfigurable region definition. It captures all reconfigurable region related information required at the Clash level to realize a partially reconfigurable design as elaborated upon in Section 5.2.1. The bitstream starting addresses and sizes were back-annotated after walking through the FPGA EDA tool used.

Configuration interface The partial bitstreams are stored in a block RAM on the FPGA fabric itself as to not have to deal with external memories and their communication. This choice does imply that the partial bitstreams need to be back annotated after an initial run of the FPGA EDA tooling, to then compile and run that tooling again. The block RAM was instantiated as shown in Listing 5.18. A block RAM contents file named `bitstreams.rbt` was manually compiled. It

```haskell
-- | Add two Signed numbers, output the result and whether over- or
-- underflow has occurred in the addition.
addOUF
  :: KnownNat n
  => Signed (n+2)
  -> Signed (n+2)
  -> (Signed (n+2), Bool)
addOUF x y = (truncateB r, ouf)
 where
  r = add x y
  ouf = let (a:>b:>_) = bv2v (pack r) in a /= b
  --Over- or underflow occurred if the two most significant bits aren't equal

{-# NOINLINE incrementer #-}
{-# ANN incrementer
  (Synthesize
    { t_name   = "incrementer"
    , t_inputs = [ PortName "clk", PortName "rst", PortName "en", PortName "inp" ]
    , t_output = PortProduct "" [ PortName "outp", PortName "ouf" ]
    }) #-}
incrementer
  :: ( HiddenClock XilinxSystem
     , HiddenReset XilinxSystem
     , HiddenEnable XilinxSystem
     )
  => Signal XilinxSystem (Signed 3)
  -> Signal XilinxSystem (Signed 3, Bool)
incrementer num = bundle (s', ouf)
 where
  s = register 0 s'
  (s', ouf) = unbundle $ addOUF <$> s <*> num
```

Listing 5.16: *One of the two reconfigurable modules: an accumulator that adds the input value to the current state. Resets to 0.*

```haskell
prIncrement
  :: HiddenClockResetEnable XilinxSystem
  => Region 2 ( Signal XilinxSystem (Signed 3)
              -> Signal XilinxSystem (Signed 3, Bool) )
prIncrement = Region $ (incrementer, 0, 13000) :> (decrementer, 13000,
↪  13000) :> Nil
```

Listing 5.17: *The reconfigurable region definition.*

contains the two partial bitstreams, the first starting at address 0, the second at address 13000. They fit together in 26000 32-bit words, hence the block RAM size.

```
bitstreams
  :: (Enum addr, HiddenClock dom, HiddenEnable dom)
  => Signal dom addr
  -> Signal dom (BitVector 32)
bitstreams rdAddr
  = blockRamFile
      (SNat :: SNat 26000) -- ^ The bitstream size in 32-bit words
      "bitstreams.rbt"     -- ^ Path to file with block RAM contents
      rdAddr               -- ^ The blockram read addresses
      (pure Nothing)       -- ^ Don't ever write
```

Listing 5.18: *Create a block RAM with 26000 32-bit values that is only ever read from.*

The top entity The top entity is provided in Listing 5.19. It has a clock, a reset and an enable signal as inputs and a single bit signal indicating overflow as output. The ICAPE2 is instantiated at line 11 and receives a bitstream from the controller instantiated at line 13. The reconfigurable region input is set to a constant one, so the modules always increment or decrement with one. The region output is determined by an instantiation of `region` using the region definition from Listing 5.17. The region in- and outputs are decoupled during reconfiguration to avoid undefined values from propagating through the reconfigurable module and the static region. Default values are output during that time. As shown at lines 20 through 24, DPR is triggered for the *other* module if and only if the currently active module outputs over- or underflow having occurred.

Line 1 splices in an `ila_0` function as described in Section 5.5. It is instantiated at line 33 with as probe inputs the signals at lines 28 through 32. These inputs are also marked as *traced* for simulation. This was required for them to end up in a simulation trace dump to a Value Change Dump (VCD) file that is used below.

Xilinx Vivado and implementation on an FPGA The top entity was compiled to Verilog and the resulting sources were added to a project in Xilinx Vivado 2019.2. The project was set to target a Digilent Arty 7 development board featuring a Xilinx Artix-7 XC7A35 FPGA. Xilinx ILA IP core in native mode with the number of probes and probe widths as used in the design was added too. The core was configured to add an input pipe stage, which is important to have the ILA measurements match Clash simulation cycle-accurately.

During the implementation step of Vivado, the reconfigurable modules were assigned to a Pblock in the floorplanner. The Pblock has its `RESET_AFTER_RECONFIG` property set to have the reconfigurable modules in a predefined state after reconfiguration and to have it match the Clash design as the `XilinxSystem` domain has reset values enabled.

```haskell
makeIla "ila_0" [1,2,2,3,1]
topEntity
  :: ( "clk" ::: HiddenClock XilinxSystem
     , "rst" ::: HiddenReset XilinxSystem
     , "en"  ::: HiddenEnable XilinxSystem
     )
  => "result" ::: Signal XilinxSystem Bit
topEntity = ila `hwSeqX` icap `hwSeqX` (boolToBit . snd <$> regionOut)
 where
  -- We're not interested in the ICAP output, but need it in a call to `hwSeqX` to stop Clash from optimizing the ICAP primitive away.
  icap = icape2Write hasClock (bytewiseReverseBV <<$>> mBitstream)

  (regionSt, mBitstream) = controller prIncrement bitstreams trig

  -- Always increment or decrement with one.
  regionIn = decoupler 0 <$> regionSt <*> pure 1

  regionOut = decoupler (0, False) <$> regionSt <*> region prIncrement regionSt regionIn

  trig :: Signal XilinxSystem (Trigger 2)
  trig = ( \ouf regSt -> case (ouf, regSt) of
      (True, Active n) -> Just $ complement n
      _                -> Nothing
    ) <$> (snd <$> regionOut) <*> regionSt

  -- We'd like to give the signals nice descriptive names as they end up in the Vivado ILA GUI.
  traceAndName s = nameHint s . traceSignal1 (ssymbolToString s)
  ilaEn = traceAndName (SSymbol @"Enable") $ pack <$> fromEnable hasEnable
  ilaTrig = traceAndName (SSymbol @"Trigger") $ mIndexToOH <$> trig
  ilaRegSt = traceAndName (SSymbol @"RegSt") $ pack <$> regionSt
  ilaCnt = traceAndName (SSymbol @"Count") $ pack . fst <$> regionOut
  ilaOuf = traceAndName (SSymbol @"OverOrUnderflow") $ pack . snd <$> regionOut
  ila = ila_0 hasClock ilaEn ilaTrig ilaRegSt ilaCnt ilaOuf
makeTopEntity 'topEntity
```

Listing 5.19: *The top entity of this example design showcasing DPR.*

After bitstream generation, the partial bitstreams were included in the block RAM in the Clash design. A second round of compilation and bitstream generation later, the new bitstreams were verified to be equivalent to those from the previous round as stored in the block RAM.

The full bitstream was used to configure the Digilent Arty 7 development board, and the Vivado hardware manager was used to arm the ILA, its trigger being the design's enable input signal. The physical switch on the board that is wired to the enable signal was flipped on to have the initial reconfigurable module start accumulating and to trigger the ILA. The resulting measurements were stored as a VCD file.

Waveform comparison The GTKWave software was used to show waveforms of the aforementioned ILA and simulation VCD files. Sections of the waveforms are shown in Figure 5.3, the simulation traces at the top, and the ILA measurements from the design as implemented on the Digilent Arty 7 development board at the bottom. The FPGA implementation measurements were time-shifted to align it with the enable signal assertion at time instance 0 of the simulation.

The "trigger" signal uses one-hot encoding, either the least or the most significant bit being asserted respectively means PR of the incrementer or the decrementer module being triggered. The most significant bit of the "RegSt" signal uses 0 for reconfiguration being in progress, and 1 for a module being active. The least significant bit indicates the module under reconfiguration or being active; 0 for the incrementer module, 1 for the decrementer.

The leftmost section of the waveforms show the initial reconfigurable module, the accumulator, to immediately start accumulating. The counter already overflows after four clock cycles, which triggers PR of the decrementer module. The middle section of the waveforms show the FPGA coming out of reconfiguration at 13004 ps. The continuous subtraction overflows after a couple of clock cycles, which triggers PR of the accumulator module again. This module kicking into action again is illustrated in the rightmost section of the figure. Of course, this alternation of the two modules would continue indefinitely if the FPGA was to be kept powered on.

The waveforms of Figure 5.3 proof that the design elements developed and described in this chapter are suitable for realizing a Clash FPGA design with PR. Furthermore, the waveforms of the ILA measurements and the simulation being equivalent shows that such Clash designs can be simulated cycle accurately and true to how the design would run when implemented on an FPGA. With this approach retaining the ability of full-design Clash simulation while using PR, design errors may be caught more early on.

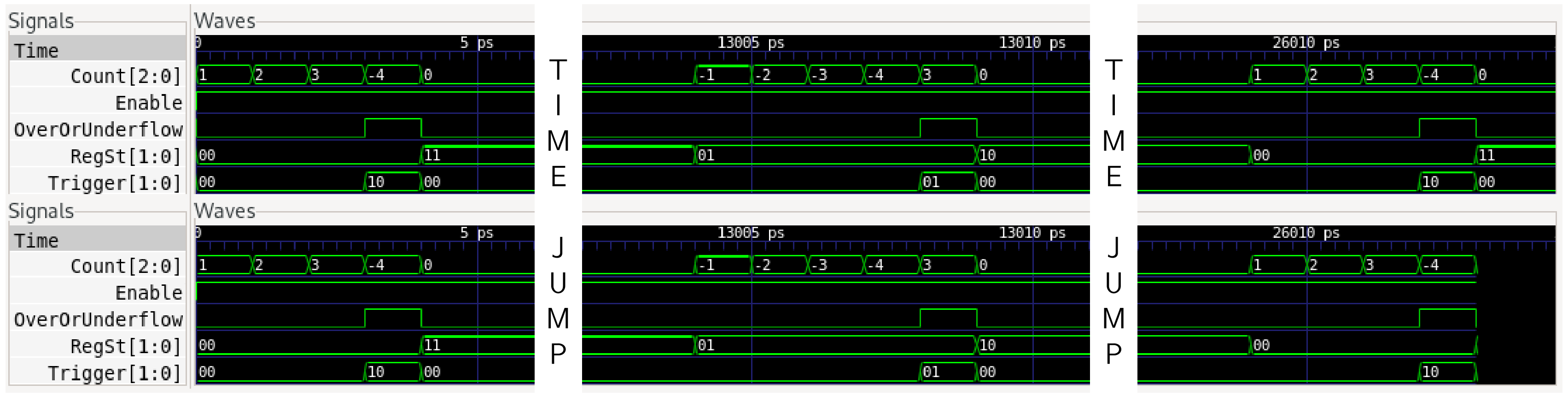

Figure 5.3: *Waveforms of a few selected signals. Top: waveforms from Clash simulation. Bottom: waveforms from measurements on an FPGA through an ILA.*

6 Application on AES

In this chapter, the left fold variant transformation is applied to an implementation of the Advanced Encryption Standard. Implementation diversity may be beneficial to AES given the desired secrecy of the secret key, and its `MixColumns` step can be expressed using left folds. This step is not typically targeted in published side-channel attacks, but an interesting candidate nonetheless.

The pre-existing AES implementation used was acquired from [47]. It supports encryption, which was successfully tested to be operating correctly through simulation of a few sample inputs. The author claims it to be FPGA-proven. Its `MixColumns` step is declared as shown in Listing 6.1.

```haskell
mixColumns :: AESState -> AESState
mixColumns = map mixColumn
    where
    mixColumn :: Vec 4 (BitVector 8) -> Vec 4 (BitVector 8)
    mixColumn col = r0 :> r1 :> r2 :> r3 :> Nil
        where
        doubled = map (pack . gfDouble . unpack) col
        r0 = (doubled !! 0) `xor` (col !! 3) `xor` (col !! 2) `xor`
         ↪  (doubled !! 1) `xor` (col !! 1);
        r1 = (doubled !! 1) `xor` (col !! 0) `xor` (col !! 3) `xor`
         ↪  (doubled !! 2) `xor` (col !! 2);
        r2 = (doubled !! 2) `xor` (col !! 1) `xor` (col !! 0) `xor`
         ↪  (doubled !! 3) `xor` (col !! 3);
        r3 = (doubled !! 3) `xor` (col !! 2) `xor` (col !! 1) `xor`
         ↪  (doubled !! 0) `xor` (col !! 0);
```

Listing 6.1: *The MixColumns step of an AES round. Source: [47]. Please refer to the license at `https://github.com/adamwalker/clash-utils/blob/master/LICENSE`.*

It was rewritten to use `Clash.Prelude.foldl` calls and subjected to the variant transformation concocted from the left fold transformation primitive outlined in Section 4.4. This is shown in Listing 6.2.

With four vectors of length five being folded and with `termsOfSum` from Listing 4.4 returning 15 possible ways of splitting one such vector, a total number of $15^4 = 50625$ functionally equivalent `MixColumns` variants are spliced. They are named `mixColumnsV_0` through `mixColumnsV_50624`.

Unfortunately, there was no opportunity for the experimental verification of these AES `MixColumns` variants by performing power measurements as there was no working test bench available. Thus the extend to which mounting a successful side-channel attack is impeded by these specific variants remains unclear. However, this case study can be used to conclude that this Clash-based approach lends itself well to deriving sheer numbers of variants in an automatic fashion. The variant derivation approach can furthermore be highly mathematical in nature to enable provably correct transformations.

```
$(varDecs
    (mkT tfFoldl)
    [d| mixColumnsV col doubled = let
          r0 = foldl xor zeroBits (doubled!!0 :> col!!3 :> col!!2 :>
           ↪  doubled!!1 :> col!!1 :> Nil)
          r1 = foldl xor zeroBits (doubled!!1 :> col!!0 :> col!!3 :>
           ↪  doubled!!2 :> col!!2 :> Nil)
          r2 = foldl xor zeroBits (doubled!!2 :> col!!1 :> col!!0 :>
           ↪  doubled!!3 :> col!!3 :> Nil)
          r3 = foldl xor zeroBits (doubled!!3 :> col!!2 :> col!!1 :>
           ↪  doubled!!0 :> col!!0 :> Nil)
          in (r0,r1,r2,r3)
    |] )

-- | The MixColumns step of an AES round
mixColumns :: AESState -> AESState
mixColumns = map mixColumn
  where
    mixColumn :: Vec 4 (BitVector 8) -> Vec 4 (BitVector 8)
    mixColumn col = r0 :> r1 :> r2 :> r3 :> Nil
      where
        doubled = map (pack . gfDouble . unpack) col
        (r0,r1,r2,r3) = mixColumnsV_0 col doubled
```

Listing 6.2: *The `MixColumns` step modified for the automatic derivation of variants using the left fold transformation primitive.*

7 Conclusions & recommendations

7.1 Implementation diversity

Conclusions The first two research questions asked how to automatically derive functionally equivalent variants of a given hardware design with the goal of adding implementation diversity as a side-channel attack countermeasure. The proposed solution involves formulating concepts to deriving suitable variants as transformation rules. Such rules operate at an abstract syntax tree representation of a hardware description and they are automatically applied wherever possible by traversing the tree and accumulating any resulting variants at compile time.

As a proof of concept, one concept for transforming left fold higher-order functions was worked out to such a transformation rule. The `MixColumns` step of an AES cipher implementation could be described using such functions and was subjected to the transformation rule. This fully automatic application resulted in over 64000 variants. With those variants being physically different in relation to one another, it stands to reason that they each have different side-channel leakage characteristics; their alternated usage would introduce some temporal jitter.

In conclusion, variants of a given hardware description can be automatically derived using the novel approach outlined in this thesis. This does assume that the desired approach to obtaining variants can be formulated as a transformation rule. The strongly mathematical formalism of Clash enabled provably correct transformations that were previously unexplored in the context of implementation diversity. There was no opportunity for experimental verification of the example AES `MixColumns` variants, thus the extend to which mounting a successful side-channel attack is impeded by the left fold transformation rule specifically remains unclear. Nonetheless, this work may aid in the (semi)automatic addition of implementation diversity to a hardware design as a side-channel or fault attack countermeasure, particularly when used with variant derivation approaches that have been proven to be beneficial.

Recommendations Future research of the novel compile-time facilities for AST traversal and variant accumulation proposed in this thesis will need to show whether they are sufficiently versatile and efficient in aiding automated variant derivation. This could be done in combination with the left fold transformation rule, or with other rules that are found to be more promising or proven to be beneficial in combating side-channel attacks.

For some variant derivation approaches, a notable optimization could be realized by not necessarily considering every variant as entirely self-contained. For example, the AES `MixColumns` variants described in this thesis only differ in how several operations are wired together. By retaining the overlapping logic, potentially huge savings in the physical footprint of variants could be realized.

Finally, the hard- or software required for the continuous pseudo-random exchange of variants was not explored in this thesis. Perhaps a reusable approach to that could be developed, if not existing already.

7.2 Partial dynamic reconfiguration

Conclusions Research questions four and five ask how FPGA designs featuring partial dynamic reconfiguration can be simulated and compiled by the Clash compiler. The proposed solution revolves around a design element representative of a reconfigurable region. Through the use of a HDL primitive, different behaviour for compilation and simulation could be defined. This design element is special in that it can simulate the process of undergoing reconfiguration in the Clash simulator. Like on a real FPGA, modules can be exchanged, and the signals going into the static region carry undefined values during that process. When compiled, it produces the sources needed by FPGA EDA tools to enable partial reconfiguration.

Other core components needed to realize an FPGA design featuring partial dynamic self-reconfiguration in Clash were identified and realized. A novel partial reconfiguration controller was designed, and Clash HDL primitives for instantiating the Xilinx ICAP configuration interface and the Xilinx integrated logic analyser IP core were written.

To verify the correct operation of all components, an example design with one reconfigurable region and two reconfigurable modules was written. It was successfully implemented on a Xilinx Atix 7 FPGA. Waveforms acquired from both Clash simulation and from the FPGA through an integrated logic analyser were shown to be equivalent. This shows that simulation can be cycle-accurate and true to what is happening on an actual FPGA.

All in all, the successful implementation and simulation have shown the proposed solutions to be working. They can help one to easily attain a Clash-based FPGA design flow featuring partial reconfiguration while retaining the ability of functional verification through Clash simulation.

Recommendations Despite these conclusions, there is room for improvement of some of the proposed design elements. For example, the partial reconfiguration controller could be extended to support bitstream storage locations that take some number of clock cycles to provide their response. The currently assumed same clock cycle response works out when storing bitstreams in block RAM, but adding compatibility with other storage devices can be beneficial.

Furthermore, the partial reconfiguration controller could incorporate detection and handling of potential reconfiguration failures to increase robustness. Some FPGAs feature configuration interfaces that provide feedback on whether configuration is done or on whether an error has occurred. Alternatively, the bitstream could be checked for the presence of words that any bitstream should contain, a synchronization word for example.

Finally, Clash compilation currently only yields the HDL sources needed by FPGA EDA tools to enable partial reconfiguration. It may be desirable to automatically set up parts of the EDA tool project based on what is defined in the Clash design. For example, the reconfigurable regions, their corresponding reconfigurable modules and perhaps the region's physical footprint could already be set.

References

[1] YongBin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptol. ePrint Arch.*, 2005:388, 2005.

[2] Robert Richardson and CSI Director. Csi computer crime and security survey. *Computer security institute*, 1:1–30, 2008.

[3] Daniel Ziener. Security in embedded hardware. 2019.

[4] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[5] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The em side—channel(s). In *International workshop on cryptographic hardware and embedded systems*, pages 29–45. Springer, 2002.

[6] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[7] Nele Mentens, Benedikt Gierlichs, and Ingrid Verbauwhede. Power and fault analysis resistance in hardware through dynamic reconfiguration. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 346–362. Springer, 2008.

[8] Benjamin Hettwer, Johannes Petersen, Stefan Gehrer, Heike Neumann, and Tim Güneysu. Securing cryptographic circuits by exploiting implementation diversity and partial reconfiguration on fpgas. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 260–263. IEEE, 2019.

[9] Ivan Bow, Nahome Bete, Fareena Saqib, Wenjie Che, Chintan Patel, Ryan Robucci, Calvin Chan, and Jim Plusquellic. Side-channel power resistance for encryption algorithms using implementation diversity. *Cryptography*, 4(2):13, 2020.

[10] Stephen M Steve Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology: This paper reflects on how moore's law has driven the design of fpgas through three epochs: the age of invention, the age of expansion, and the age of accumulation. *IEEE Solid-State Circuits Magazine*, 10(2):16–29, 2018.

[11] Umer Farooq, Zied Marrakchi, and Habib Mehrez. *Tree-based heterogeneous FPGA architectures: application specific exploration and optimization*. Springer Science & Business Media, 2012.

[12] Cindy Kao. Benefits of partial reconfiguration. *Xcell journal*, 55:65–67, 2005.

[13] Pascal Sasdrich, Amir Moradi, Oliver Mischke, and Tim Güneysu. Achieving side-channel protection with dynamic logic reconfiguration on modern fpgas. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 130–136. IEEE, 2015.

[14] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 219–235. Springer, 2013.

[15] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 105–114. IEEE, 2011.

[16] Sergei P Skorobogatov and Ross J Anderson. Optical fault induction attacks. In *International workshop on cryptographic hardware and embedded systems*, pages 2–12. Springer, 2002.

[17] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of aes. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 7–15. IEEE, 2012.

[18] NIST. Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197, 2001.

[19] Sıddıka Berna Örs, Elisabeth Oswald, and Bart Preneel. Power-analysis attacks on an fpga–first experimental results. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 35–50. Springer, 2003.

[20] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *International workshop on cryptographic hardware and embedded systems*, pages 292–302. Springer, 1999.

[21] R Jacob Baker. *CMOS: circuit design, layout, and simulation*. John Wiley & Sons, 2019.

[22] Abaddon1337. Cmos inverter. `https://commons.wikimedia.org/wiki/File:CMOS_inverter.svg`. Accessed: 17th November 2020. License: `https://creativecommons.org/licenses/by-sa/3.0/legalcode`.

[23] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure Integrated Circuits and Systems*, pages 27–42. Springer, 2010.

[24] Stephen Brown and Jonathan Rose. Architecture of fpgas and cplds: A tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, 1996.

[25] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.

[26] Henry Wong, Vaughn Betz, and Jonathan Rose. Comparing fpga vs. custom cmos and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 5–14. ACM, 2011.

[27] Xilinx Inc. *Vivado Design Suite User Guide - Dynamic Function eXchange (UG909)*, v2019.2 edition, January 15, 2020.

[28] Stephen M Trimberger and Jason J Moore. Fpga security: Motivations, features, and applications. *Proceedings of the IEEE*, 102(8):1248–1265, 2014.

[29] Xilinx Inc. *7 Series FPGAs Configuration User Guide (UG470)*, v1.13.1 edition, August 20, 2018.

[30] Emi Eto. Difference-based partial reconfiguration. *XAPP290 (v2. 0) December*, 3, 2007.

[31] Clash. `https://clash-lang.org`. Accessed: 17th November 2020.

[32] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. Cλash: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 714–721. IEEE, 2010.

[33] Haskell language. `https://haskell.org`. Accessed: 17th November 2020.

[34] Nahome Bete, Fareena Saqib, Chintan Patel, Ryan Robucci, and Jim Plusquellic. Side-channel power resistance for encryption algorithms using dynamic partial reconfiguration (spread). In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019.

[35] Intel. *Intel® Quartus® Prime Pro Edition User Guide: Partial Reconfiguration (UG-20136)*, June 22, 2020.

[36] Lingkan Gong and Oliver Diessel. Resim: A reusable library for rtl simulation of dynamic partial reconfiguration. In *2011 International Conference on Field-Programmable Technology*, pages 1–8. IEEE, 2011.

[37] Jan Kuper, Lutz Schubert, Kilian Kempf, Colin Glass, Daniel Rubio Bonilla, and Manuel Carro. Program transformations in the polca project. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 882–887. IEEE, 2016.

[38] Rinse Wester. *A transformation-based approach to hardware design using higher-order functions.* PhD thesis, University of Twente, Enschede, Netherlands, 2015.

[39] Rinse Wester and Jan Kuper. A space/time tradeoff methodology using higher-order functions. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–2. IEEE, 2013.

[40] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, 2002.

[41] Xilinx Inc. *Vivado Design Suite User Guide - Synthesis (UG901)*, v2019.2 edition, January 27, 2020.

[42] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003.

[43] Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 49–60, 2007.

[44] Intel. *Intel® Quartus® Prime Standard Edition User Guide: Partial Reconfiguration (UG-20136)*, September 24, 2018.

[45] Xilinx Inc. *UltraScale Architecture Configuration User Guide (UG570)*, v1.13 edition, July 28, 2020.

[46] Xilinx Inc. *Integrated Logic Analyzer LogiCORE IP Product Guide (PG172)*, v6.2 edition, October 5, 2016.

[47] Adam Walker. A straightforward, unoptimised aes implementation. `https://github.com/adamwalker/clash-utils/blob/master/src/Clash/Crypto/AES.hs`. Accessed: 6th June 2020.

A An introduction to Clash

Clash [31, 32] is a functional HDL that uses the syntax and semantics of the functional programming language Haskell [33]. This appendix aims to concisely introduce some main language concepts and conventions, particularly those most useful for reading this thesis. I imagine it can strike one as being somewhat rudimental, but I hope it at least provides useful pointers for further research when concepts are unclear. Please refer to `https://hoogle.haskell.org/` for searching in many Haskell libraries.

Haskell being a *functional* programming language suggests that meaning is conveyed by evaluating expressions as opposed to sequential execution of statements as is the case with imperative programming languages. Haskell is *pure*: expressions are *referentially transparent*. This implies that:

- Variables are immutable. A variable can only be *declared* to be (or: *bound* to) a certain value and cannot be assigned a different value later.
- Expressions are side-effect free; they cannot alter some global state. Repeated evaluation with the same inputs will always have the same result.

Declarations use the "=" keyword with the identifier name at the left hand side. They can optionally be accompanied by an explicit *type signature*: "`::`" reads as "has type".

```
x :: Int
x = 8
```

Concrete types (such as `Int`) always start with a capital letter whereas variables (such as `x`) always start with a lower case one. Function declarations look as follows:

```
plus :: Int -> Int -> Int
plus a b = a + b
```

We can read the type of function `plus` as "`Int` to (`Int` to `Int`)"; given two values of type `Int`, it returns a value that is their sum and also of type `Int`. The function can only be given values of type `Int` as Haskell programs are statically typed: all types are checked at compile time, and compilation fails until any and all type mismatches are resolved. Function application is indicated by whitespaces: `plus 1 2` evaluates to `3`.

Because of support for *partial function application*, functions can also be applied partially: `plus 1` has only one of the two arguments of `plus` applied, yet it yields a valid new function of type `Int -> Int` that adds `1` to any given integer value. This already hints at the fact that functions are *first-class*: they can be passed around and assigned just like any other value:

```haskell
increment :: Int -> Int
increment = plus 1
```

Functions that take a function as an argument or return a function as part of their result are referred to as *higher-order functions*. An example is `foldl` as illustrated in Section 4.2.

Of course there are more types than `Int`. In fact, one can define custom data types. For example, a boolean data type `Bool` can be defined as shown below. A value of this type can be either `True` or `False`. Common types and other definitions like these are also provided by the GHC and Clash Prelude modules.

```haskell
data Bool = True | False
```

Haskell's type system knows *ad hoc polymorphism* and *parametric polymorphism*. Types can contain *type variables*, such as "a" in the examples below. In contrast to *concrete types* such as `Int` or `Bool`, they start with a lower case letter. Variables of such types can take any value of any concrete type *depending on the context* in which they are evaluated. For example, `iden True` evaluates to `True` and `iden 1` evaluates to `1`.

```haskell
iden :: a -> a
iden x = x
```

Haskell is *lazy*: anything is only ever evaluated when needed. This allows us to define, for example, our own if-then-else statement as illustrated with function `ifElse` below. `ifElse True 1 2` evaluates to `1` and `ifElse False 1 2` evaluates to `2`. As can be seen, the function takes a boolean value as its first argument, which is first *pattern matched* to `True`. If the match fails, it falls through to the line below where the boolean value is matched to `False`. To know whether `x` or `y` must be returned as the result, the boolean value must be evaluated. In fact, most evaluation is driven by pattern matching: the values that `x` and `y` are *bound to* are not evaluated unless and until they are needed at some point.

```haskell
ifElse :: Bool -> a -> a -> a
ifElse True  x _ = x
ifElse False _ y = y
```

Functions like `iden` and `isElse` have a somewhat limited usability; variables of an unconstrained type can only be passed around. Type variables can be constrained, for example requiring them to be an instance of a specific *typeclass*:

```haskell
isEqual :: Eq a => a -> a -> a
isEqual x y = x == y
```

Haskell typeclasses allow one to define behaviour irrespective of one single type. For example, the `Eq` a typeclass constraint above enables the use of the (==) equality operator. The caller of `isEqual` can use any type for "a", as long as it is an instance of `Eq`, which means as much as the behaviour of (==) having been defined for it. Of course, use of (==) should be preferred over declaring and using `isEqual`.

Data types can also be polymorphic and contain type variables:

```
data Maybe a = Just a | Nothing
```

Tuples and lists can be used to combine multiple values. Tuples contain a fixed number of them, and they can be of different types. For example: `(1,'a',False) :: (Int,Char,Bool)`. Lists can be appended to or split, but only contain values of a single type. For example: `["these","are","strings"] :: [String]`, which is syntactic sugar for `"these":"are":"strings":[]`.

With Haskell lists being single-linked, they do not lend themselves well to being implemented in hardware. Clash therefore offers vectors that have their length encoded in their types as an alternative. This allows the Clash compiler to deduce their static length and thus how large such a data structure is in hardware at compile time. For example: `0:>1:>Nil :: Vec 2 Bit`.

Convention is to indicate lists and vectors like xs, the plural of x, which represents elements x_0 up to x_n where $n = (\texttt{length}\ xs) - 1$.

The application operator (\$) can sometimes be used to omit parenthesis, and the function composition operator `(.)` can be used to compose functions.

B An introduction to Template Haskell

Template Haskell (TH) [40] is an extension to the GHC compiler. It allows for compile-time metaprogramming by constructing or manipulating *abstract syntax tree* (AST) representations of Haskell code. The TH AST is represented using an ordinary Haskell data type (ADT).

When aiming to construct some Haskell source code, one can construct its AST representation in the quotation monad `Q` using regular Haskell functions and *splice* the result to instantiate the represented code at the desired location. Splicing happens through use of splicing brackets `$(` and `)` and is performed at early compile-time. Conversely, *quotation* (or: *Oxford*) *brackets* `[x|` and `|]` convert from source code to its TH AST representation. These operators being each other's inverse makes that `y` = `$( [x| y |] )` holds for any *expression* `y` when `x` is `e` or empty, any *declaration* `y` when `x` is `d`, any *type* `y` when `x` is `t` and any *pattern* `y` when `x` is `p`.

For example, consider the following Clash expression `not True`. Quotation yields its TH AST representation in the `Exp` data type: `runQ [| not True |]` evaluates to `AppE (VarE GHC.Classes.not) (ConE GHC.Types.True)`. This represents a function application (`AppE`) of the function variable (`VarE`) `not` to the data constructor (`ConE`) True.

In the above example, `GHC.Classes.not` and `GHC.Types.True` are just pretty prints of the names (of type `Name`) for the respective function and data constructor. Such names can be constructed by prepending an apostrophe to a variable in scope, for example like `'not` and `'True`. This allows us to construct the original expression by splicing as follows: `$(returnQ $ AppE (VarE 'not) (ConE 'True))`.

C Auxiliary functions on vectors in the TH AST

This section exposes a couple of functions that were used in the left fold transformation primitive in Section 4.4.2. They operate on the Template Haskell representation of Clash vectors.

These functions are declared in module `THOps2`, which imports `Clash.Prelude hiding (Exp)` and `Language.Haskell.TH` while having GHC's `ViewPatterns` language extension enabled.

```haskell
isCons :: Exp -> Bool
isCons (ConE ((==) 'Cons -> True)) = True
isCons (ConE ((==) '(:>) -> True)) = True
isCons (ConE ((==) "Cons" . nameBase -> True)) = True
isCons (ConE ((==) ":>" . nameBase -> True)) = True
isCons _ = False
```

Listing C.1: *Checks whether an `Exp` represents `Clash.Prelude.Cons` or `Clash.Prelude.(:>)`.* NB*: Matches on* any *function named `Cons` or `(:>)` to accomodate for TH ASTs acquired via the `Language.Haskell.Meta.Parse` parse functions.*

```haskell
isNil :: Exp -> Bool
isNil (ConE ((==) 'Nil -> True)) = True
isNil (ConE ((==) "Nil" . nameBase -> True)) = True
isNil _ = False
```

Listing C.2: *Checks whether an `Exp` represents `Clash.Prelude.Nil`.* NB*: Matches on* any *function named `Nil` to accomodate for TH ASTs acquired via the `Language.Haskell.Meta.Parse` parse functions.*

```haskell
lengthTH :: Exp -> Int
lengthTH (ParensE xs) = lengthTH xs
lengthTH (isNil -> True) = 0
lengthTH (AppE (AppE (isCons -> True) _) xs) = 1 + lengthTH xs
lengthTH (InfixE (Just _) (isCons -> True) (Just xs)) = 1 + lengthTH xs
lengthTH v@(UInfixE _ (isCons -> True) _) =
  error $ "lengthTH: cannot reliably interpret `Vec`s with `UInfixE` (unknown "
       <> "fixity). Either define it without infix operators or use quotation "
       <> "brackets to acquire the TH AST as opposed to the parse functions "
       <> "from the `Language.Haskell.Meta.Parse` library. The offending `Vec`"
       <> ":\n" <> show v
lengthTH x = error $ "lengthTH: could not interpret as a Vec: " <> show x
```

Listing C.3: *Returns the length of an `Exp` that represents a `Vec`.*

```haskell
viewVTH :: Exp -> (Exp, Exp)
viewVTH (ParensE xs) = viewVTH xs
viewVTH (AppE (AppE (isCons -> True) x) xs) = (x, xs)
viewVTH (InfixE (Just x) (isCons -> True) (Just xs)) = (x, xs)
viewVTH v@(UInfixE _ (isCons -> True) _) =
  error $ "viewVTH: cannot reliably interpret `Vec`s with `UInfixE` (unknown "
       <> "fixity). Either define it without infix operators or use quotation "
       <> "brackets to acquire the TH AST as opposed to the parse functions "
       <> "from the `Language.Haskell.Meta.Parse` library. The offending `Vec`"
       <> ":\n" <> show v
viewVTH (isNil -> True) = error "viewVTH: vector is empty"
viewVTH x = error $ "viewVTH: could not interpret as a Vec: " <> show x
```

Listing C.4: *Returns the pair of the head and tail of an `Exp` representing a `Vec`. Similar to `Data.List.HT.viewL`. Think of the type as `Vec n a -> (a, Vec (n-1) a)`.*

```haskell
consTH :: Exp -> Exp -> Exp
consTH x xs = AppE (AppE (ConE '(:>)) x) xs
```

Listing C.5

```haskell
nilTH :: Exp
nilTH = ConE 'Nil
```

Listing C.6

```haskell
splitAtTH :: Int -> Exp -> (Exp, Exp)
splitAtTH n (ParensE xs) = splitAtTH n xs
splitAtTH 1 xs = (x `consTH` nilTH, xs')
  where
    (x, xs') = viewVTH xs
splitAtTH n xs = (x `consTH` x', xs'')
  where
    (x,  xs')  = viewVTH xs
    (x', xs'') = splitAtTH (n-1) xs'
```

Listing C.7: *Think of the type as `m -> Vec (m + n) a -> (Vec m a, Vec n a)`.*

```haskell
splitPlacesTH :: [Int] -> Exp -> [Exp]
splitPlacesTH [] _ = []
splitPlacesTH (x:xs) vs =
  let (v, vs') = splitAtTH x vs
  in  v : splitPlacesTH xs vs'
```

Listing C.8: *Largely similar to `Data.List.Split.splitPlaces`.*

```haskell
listToVecTH' :: [Exp] -> Exp
listToVecTH' [] = nilTH
listToVecTH' (x:xs) = x `consTH` listToVecTH' xs
```

Listing C.9: *Go from a list of elements to the TH AST representation of a `Vec` of those elements. Think of the type as `[a] -> Vec n a`.*