Configurable Central Clock and Reset Controller

Alexander Tegnvallius Boklund, Ture Strömberg
al7346te-s@student.lu.se, tu6544st-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Liang Liu
Mattias Sönnerup, Ericsson

Examiner: Pietro Andreani

June 8, 2026

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

With growing complexity of developing Application-Specific Integrated Circuit (ASIC), comes a higher development time, in both designing and verifying Register Transfer Level (RTL) code. Configurability can reduce the time spent on re-designing, and instead replace with simple reconfiguration. The configurable design then typically need initial verification effort.

This thesis seeks to investigate the possibilities of making the Clock and Reset Controller (CCR) block more configurable, both in terms of run-time and compile-time configurability. The configurability makes the interface and logic of modules more predictable which more easily enable reusability in the verification framework. However, as concluded in the thesis, using SystemVerilog language features for configurability can reduce the readability of the RTL code. Code generation can be used to generate more readable RTL files, but can be harder to implement.

The thesis investigates different run-time Finite State Machine (FSM) architectures suitable for CCR, and choses one for implementation. The results show that the design sets constraints on the functionalities of the FSM and can come with increased area and power, but can save from expensive re-tapeouts in case of faulty implemented logic.

Clock division and clock muxing are critical features used throughout the CCR. The thesis investigates current clock division and clock muxing designs and proposes new implementations, that can be configured for in the CCR. The proposed clock mux implementation has less area, power, critical path and faster transition time, than using traditional 2 input glitch free multiplexers, according to our measurements. However, more testing would need to be done.

One technique to integrate third-party Intellectual Propertys (IPs) is to design a wrapper module. A unified Phase-Locked Loop (PLL) wrapper interface was designed, that can integrate a variety of PLLs. The thesis also proposes a general software automation flow, where code generation is used to integrate a PLL wrapper, through defining pin connections and configurations in a spreadsheet.

Popular science summary

In modern electronic systems, it is common to see the inclusion of one or many ASICs, which are "chips" developed for specific tasks. Inside the ASIC, there are numerous logical modules called IP, which handles some functionality of the ASIC. This thesis explores one such IP at Ericsson called CCR. Clocks are essential to the ASIC and allows for determinism, robustness, synchronisation and data pipelining. Reset signals ensure that the ASIC go to a deterministic state, which is particularly useful when booting up the ASIC. The CCR ensures that other IPs receive stable clocks and reset.

The IPs in an ASIC are designed using hardware description languages. These languages allows the designer to describe how the IP should function, using RTL code. One popular RTL coding language called SystemVerilog includes features to design configurable code. This means that the code can adapt to changes in its specification, without writing new code to describe the new functionality. Configurability can avoid redesign of the IP, which can reduce the development time and reduce human-made errors. This thesis explores the possibility of making the CCR IP more configurable.

The results show that making a configurable IP comes with challenges and trade-offs. One such trade-off is that the readability of the RTL code can be worse when making it configurable. Another challenge is making the design configurable while the ASIC is running, as this can increase the area and power consumption.

The thesis can be used by ASIC RTL design engineers seeking to learn about techniques used and the trade-offs of designing configurable RTL code, such as using software scripts to generate RTL code. The thesis also proposes new implementations that can be used when constructing designs similar to the CCR.

Acknowledgments

Firstly, we would like to thank everyone in the Ericsson team for help and support during the thesis. Special thanks to our Ericsson supervisor Mattias for his guidance and encouragement to explore creative solutions during our thesis. We also want to thank our university supervisor Liang for help with structuring the thesis and for his part in this fascinating master program.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial intelligence.
ALU	Arithmetic logic unit.
AON	Always-On.
APB	Advanced Peripheral Bus.
ASIC	Application-Specific Integrated Circuit.
CCR	Clock and Reset Controller.
CDC	Clock Domain Crossing.
DFT	Design for testing.
FIFO	First In, First Out.
FPGA	Field-programmable gate array.
FSM	Finite State Machine.
IC	Integrated circuit.
IP	Intellectual Property.
LCR	Local Clock and Reset.
OTP	One-Time Programmable.
PLL	Phase-Locked Loop.
POR	Power-On Reset.
RTL	Register Transfer Level.
SoC	System-on-Chip.
SPI	Serial Peripheral Interface.
UVM	Universal Verification Methodology.

Chapter 1

Introduction

1.1 Background

Modern System-on-Chips (SoCs) require a robust and efficient way of distributing clocks and resets throughout the chip. With an increasing number of IPs and SoC size, a structured approach is to centralize the clock and reset management into one IP, often called CCR. The IP controls and coordinates multiple PLLs, which generates the clocks that are used throughout the ASIC. The IP controls the order of resets on the chip through reset sequencing, which ensures correct system startup.

1.2 Problem & motivation

Currently, the CCR in this Ericsson IP development team is redesigned for each ASIC project to meet specific requirements, such as the number of PLLs, the number of clock observation ports, the reset sequence order and PLL vendor differences. Configurability of the CCR can reduce the time to adapt to changes in requirements, without requiring re-design of the hardware logic.

The majority of the time of ASIC development is during verification. Making a module configurable could reduce verification effort when adapting the test framework to a new ASIC project, by enabling reusability in the test framework, such as functional coverage checks and testcases.

1.3 Aim and scope

This thesis analyses multiple CCRs from different ASIC projects at Ericsson, to find differences in the designs, and then aims to make the CCR architecture more

configurable. Advantages and drawbacks of different ways to achieve configurability will be explored and discussed, with configurability options such as SystemVerilog parameters and code generation. The thesis aims to evaluate between using run-time configurability and compile-time configurability and when and why either is used. The thesis explores ways to make the CCR architecture reusable, with focus on making the various submodules reusable. The thesis will explore how Python programming language can be used to achieve configurability, in automatically generated PLL wrapper, and setting parameters for the run-time FSM.

The thesis will mainly focus on RTL architecture as design methodology. Other than doing power and area synthesis, the thesis will not focus further on back-end implementation. Software drivers for the CCR are not implemented and SoC integration is mainly considered in the PLL wrapper part of the thesis. Verification, although a big and important part of the work in making the CCR configurable, will only be briefly analysed.

1.4 Previous work

The previous work on CCRs mainly comes from internal documents and code from Ericsson. Finding examples from outside Ericsson proved to be quite difficult as it tends to be done confidentially at companies. Previous work on specific parts of the CCR has been found outside Ericsson (for example on clock muxing), but not for the whole block. Previous work on configurability and how to design configurable designs has been found and used to help in the design process. One example of this is the article [1]. This source proposed using design patterns, which is usually found in software development instead of using component-based reuse, which is usually done in hardware design.

1.5 Contributions

- Analyse various CCR architectures and propose a new configurable CCR architecture, with configurable CCR sub-blocks. Evaluate the possibilities and limitations of the proposed configurable architecture.
- Examine the differences between various PLLs architectures and create a unified PLL wrapper interface to be used in the CCR architecture. Explore ways to automate the design, verification, and interconnect of the PLL wrapper using a single source of truth, such as an Excel sheet.
- Examine different implementations of glitch-free clock multiplexers and clock dividers and provide configurability for selecting the type. Evaluate benefits and drawbacks of different implementations using area, power, and functionality as evaluation metrics.
- Examine different types of run-time FSM architectures with trade-offs in area, power, and configurability. Evaluate if and how existing and future

CCR architectures can utilize a run-time FSM, and develop a user-friendly method for configuring the FSM.

Chapter 2

Theory

This chapter will go through relevant theoretical background needed to understand the CCR architecture. The chapter will include a short introduction to PLLs, clocking and reset concepts, and techniques when writing configurable RTL code.

2.1 PLL architecture and digital interface

PLL is typically a mixed-signal component used in clocked circuitry, found in Integrated circuits (ICs). PLL design is a complex topic that contains many things that are not relevant to the thesis, thus only the basics will be presented.

For many applications, it is relevant to use a PLL as a frequency synthesizer, meaning it can generate one or many output frequencies from one or many input frequencies. Figure 2.1, shows a PLL used as a frequency synthesizer. A more complete explanation of PLL and frequency synthesis can be found in [2].

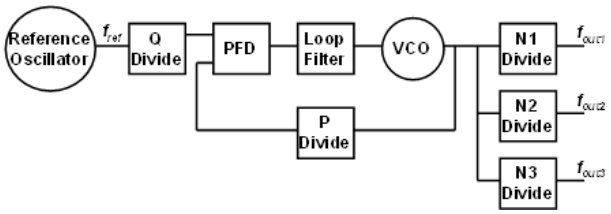

Figure 2.1: PLL architecture with multiple output dividers, taken from [3].

The digital interface of the PLL heavily depends on the type of PLL architectural design. Sources [4] and [5] shows two examples of a PLL digital interfaces. Important common PLL signals are: One or more input reference clock, one or more output clock signals, input PLL reset or PLL enable, used to activate the PLL, and output PLL lock, which indicates that the output frequency is stable.

The PLLs typically include configuration ports for configuring the PLL, with the divider values being an example.

2.2 Clocking architectures and techniques

An ASIC or SoC that needs sequential logic will most likely use some sort of clock as it has been proven over time to be safe and reliable [6]. SoCs may also use multiple clocks, create new clocks through clock division, multiplex between different clocks and gate some part of the chip from a clock. All of these topics will be covered in this section.

2.2.1 Clocking characteristics

Metastability can occur in a digital synchronous system when setup and hold time violations are violated, as seen in Figure 2.2. Metastability is when the output of the flip-flop is unknown [6].

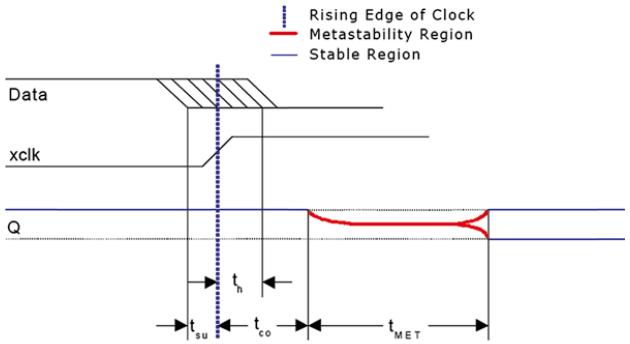

Figure 2.2: The *data* signal changed to close to the $xclk$ clock edge, resulting in a metastable state, taken from [6].

Clock skew and jitter are two characteristics of clock signals. Clock skew is when a clock signal has a delay between two different points on a chip. Clock jitter, on the other hand, is when the clock period will vary over time, usually randomly [6].

2.2.2 Clock domains

A clock domain is a part of a chip that runs on a clock with a specific frequency and phase [6]. A chip might have multiple separate clock domains that use different frequencies and have phase differences between each other. Sending data between

two clock domains is referred to as a Clock Domain Crossing (CDC). Sending data between two clock domains without a synchronizer is very risky as the arrival of the data might violate setup or hold time and cause metastability problems. To solve this a synchronizer circuit is needed, this is usually done by using two or more flip-flops, as can be seen in Figure 2.3. When the first flip-flop takes a sample of the data, it might become metastable, as the second one will sample a whole clock cycle later, the output is quite likely to have become stable and thus resolved the issue. Adding more flip-flops can reduce the risk of metastability even further. The cost of the synchronizer is latency and the area of the flip-flops. For multiple bit CDCs and crossing from faster to slower clock domains, more complex synchronization strategies may need to be used.

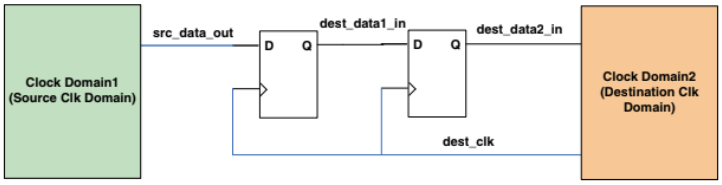

Figure 2.3: A clock domain crossing using two flip-flops, taken from [6].

2.2.3 Clock dividers

Clock division is used to generate a slower clock from a faster clock [6], the opposite of what a PLL is usually used for. There are multiple ways to do clock division. To divide by a power of two, a ripple counter can be used as can be seen in Figure 2.4. There is a connection through an inverter from Q to D for each flip-flop that is not shown in the figure. A flip-flop in the ripple counter simply inverts its value on clock pulses and sends that output to the next flip-flops clock input. The output and input of the ripple counter will have the following relationship:

$$f_{out} = \frac{f_{in}}{2^{numberOfFlipFlops}} \tag{2.1}$$

The positive aspect of the ripple counter is that it uses very little hardware and can therefore be used when low power and area is needed. The negative aspect of using a ripple counter is that it causes clock skew between each new clock in the pipeline. This builds up in each stage and thus becomes more and more skewed.

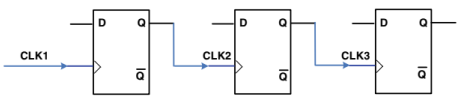

Figure 2.4: A ripple counter, taken from [6].

If a division by an even integer that is not a power of two is desired, then a counter can be used, where the generated clock signal flips every time the counter reaches the desired value divided by two. For example, let's say division by 10 is necessary. Then what you would do is have a counter that counts from zero to 4 and then flip a value stored in a flip-flop as the counter goes back to zero. The clock signal is equal to the value of this flip-flop.

If a division by an odd integer is needed then it becomes a bit more complicated how to do that and still get a 50% duty cycle. In order to achieve the 50% duty cycle, create two clock signals that are half the desired frequency and put them 90°out of phase from each other. Then to generate the desired frequency, simply perform an XOR on these two signals. A more detailed description of how to generate the phase shifted signals can be seen in [6].

2.2.4 Clock multiplexing

Multiplexing between separate clock signals can be useful when it is needed or desired to run a part of the ASIC at different frequencies, at different points in time. Another use case would be when there is need to observe different clocks. Using a normal multiplexer for clocks can cause glitches when switching between them [7]. What is referred to as a glitch is when the output clock of the mux has irregular high/low periods compared to the clocks being selected between. This could for example happen if the selected clock is high and then the select signal flips while the other clock is low causing the output to glitch as the high period of the clock is too short.

To avoid this issue, a glitch free mux could be used. A commonly used approach can be seen in Figure 2.5. The way the circuit works in simple terms is that when the select signal flips, it first deselects the current output and then selects the new signal. This is done by having the flip-flops and the feedback to the other side when the clock has been deselected. The reason why there are two flip-flops on each side and not just one is to avoid metastability problems. However, this leads to more latency between the select signal and the output clock changing. This architecture also has the problem that if one of the clocks stops ticking while it is selected then you can't switch to the other one. If the clocks that are being selected between are generated from the same reference clock, then both will be active at the same time and this won't be a problem. For cases where the clocks have completely different sources and one or both can disappear, timers can be used to check if one of the clocks are dead, and if that's the case, reset the flip-flops on that side. The timers might use one of the clocks (maybe it is certain that it won't disappear), use a third clock or perhaps have timers that use the other sides clock as is done in [7].

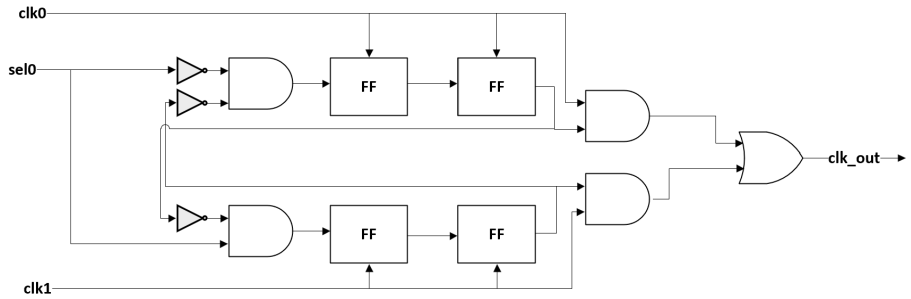

Figure 2.5: Glitch free clock mux for two clocks.

2.2.5 Clock gating

Clock gating is the process of gating a clock to a part of a chip, in other words, not sending the clock to that part of the chip [6]. This can be used in order to reduce power consumption by not sending the clock to parts of the chip that are not actively being used. Gated logic is usually implemented by using an AND gate where one input is the clock and the other the enable signal as can be seen in Figure 2.6. One potential problem with this design is that if the enable is changed while the clock is high, then it could cause glitching on the line. One way to avoid this is to ensure that the gating logic only changes the enable when the clock is low. A perhaps safer way to protect against glitching is to use latch based clock gating as can be seen in Figure 2.7. The latch lets through the enable when the clock is low and stores it while the clock is high, this helps protect against glitches as the clock can't go down prematurely.

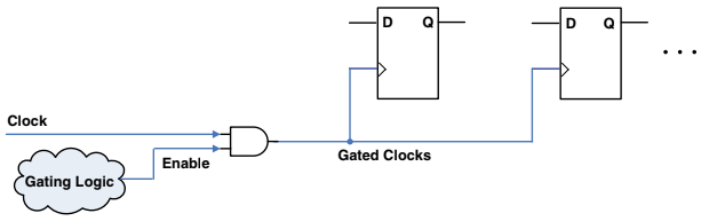

Figure 2.6: Depicts latch free clock gating, taken from [6].

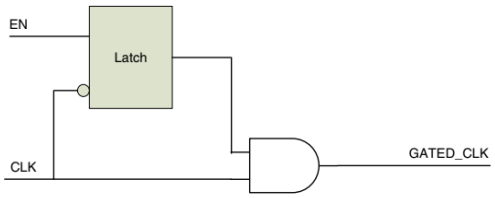

Figure 2.7: Latch based clock gating, taken from [6].

2.3 Reset architectures and techniques

In a SoC, reset is a vital signal used for setting flip-flops to a determined state. Without reset, the flip-flops would not start with a determined state, and unknown signals would propagate through the design. A good design strategy should minimize the probability of unknown states. Reset can be asynchronous, synchronous, or a mix of the two [6].

2.3.1 Synchronous and asynchronous reset

Synchronous resets are resets of the flip-flops that are done on active edges of the clock. Synchronous resets ensure that resets are done in a controlled and predictable manner and can reduce the risk of unpredictable behaviour, such as metastability. However, if the clock signal is not synchronized effectively with the reset, issues such as clock skew, setup and hold violations can occur [6].

Asynchronous resets are reset that are applied to the flip-flops without being synchronous with the clock edges. This is useful when the system demands that reset happens immediately without waiting for clock edges. However, asynchronous reset is prone to potential timing hazards and metastability. Asynchronous reset is potentially easier to implement than synchronous reset due to not needing logic for synchronisation [6].

Asynchronous reset with synchronous release is a common way to handle the reset on SoC, which combines the advantages of synchronous and asynchronous reset. The system has an immediate asynchronous reset of the flip-flops and has a controlled and predictable release of reset. When designing this type of reset it is important to ensure that the synchronous release is made properly and that it does not cause metastability. These can be described as two potential problems. First is violation of reset recovery time, meaning the time between reset de-assertion and clock edge is too low. Second is that the reset removal is the propagation delays on the SoC causing some flip-flops to exit reset before others [6].

The solutions to the problems described is to use a reset synchronizer, seen in Figure 2.8. The reset synchronizer has two flip-flops that drives the $masterrst\_n$ when rst_n is asserted. When rst_n is de-asserted, the first flip-flop will latch the

logical 1 at the D input. The second flip-flop then latches this signal after another clock cycle. The second flip-flop is needed to remove the metastability that might appear in the first flip-flop. Metastability due to recovery time violations cannot appear in the second flip-flop. This is because the output and input of the input is both low the reset is de-asserted [6].

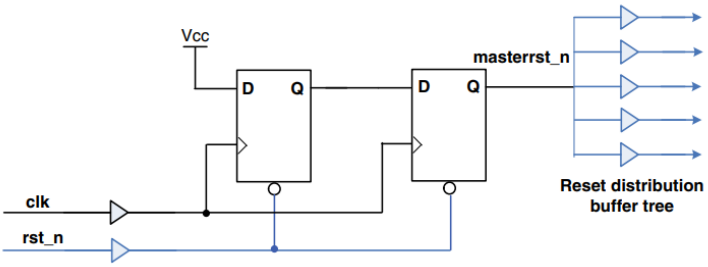

Figure 2.8: Circuit showing a reset synchronizer, taken from [6].

This design of reset synchronizer in Figure 2.8 can be susceptible to glitches on the rst_n line. Small glitches may cause the whole system to reset. One solution is to implement a digital delay, seen in Figure 2.9, which filters out glitches.

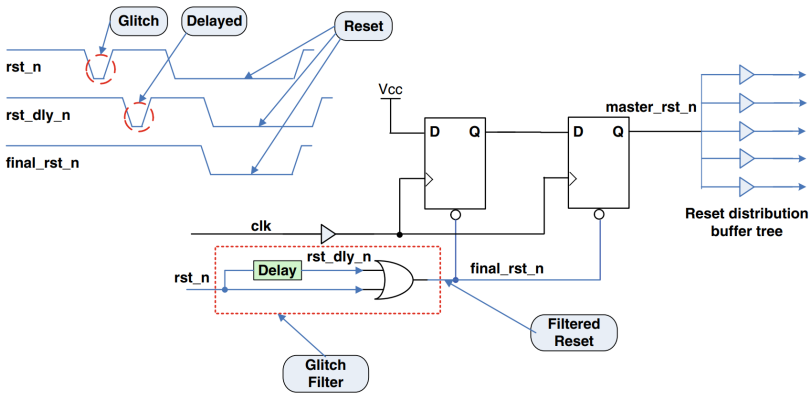

Figure 2.9: Circuit showing reset synchronizer with glitch filter, taken from [6].

2.3.2 Reset types

There are different reasons why resets are done on a SoC, thus the reset controller may treat them differently. When the device is booting, a Power-On Reset (POR) is conducted. The POR is a circuitry often embedded into the IC that generates the reset signal when it receives power [8]. Other reasons for reset are commonly

known as system resets. These could be of two different types: reset to the whole system, also known as cold reset, or resets to specific parts of the IC, also known as warm resets [9].

2.3.3 Reset sequences

At Ericsson resets on large ASICs are often divided into reset domains. One reset domain is a group of logic that share the same reset signal and reset conditions. A reset sequence, often implemented as FSM, is a specific order in which reset signals to reset domains are asserted and de-asserted. In large systems, this is needed because the reset of a whole system can cause strain on the power supply to instantaneously deliver power to all parts of the system at the same time. A reset sequence spreads out these loads, which causes the resets to be smoother in the system [10].

Reset sequences also give a more controlled way to do a reset to the SoC, if resets are needed in a specific order, which can be useful for example: apply PLL enable and then wait for PLL lock, and after that going to state that resets the memory, and so on. The clock needed to drive the FSM needs to be stable at the time of reset. For this, a reference clock coming from an oscillator is often used [10].

2.4 Configurability in RTL design

A configurable IP is one that has options that affect the IP in various ways. Configurability can be done at run-time or at compile-time. The first subsection will go over SystemVerilog compile-time techniques, which can be used to support variable port widths, array sizes, enabling and disabling ports, modules and signals. It can also be used to include or not include logic. The last subsection will go through run-time configurability.

2.4.1 Compile-time configurability

Generate construct can be used to configure a design during compile time. This can be in the form of instantiating a certain amount of modules and based on conditional statements do different routing or instantiation of modules [11].

Module parameters are values set when instantiating a module and can not be changed at run-time. This type of parameter can be used for variable port and array width, values used by logic, or the inclusion and exclusion of logic together with generate blocks. Module parameters allow for different configurations when instantiating modules [11].

Compiler directives allow for text substitution in code at compile time. The

‘define macro can be used together with ‘ifdef and ‘ifndef to optionally include ports and logic, among other things. Compiler directives do not easily allow for different configurations when instantiating the same module, and if this is needed it is possible define and undefine the macros [11].

Code generation with software tools can be used when the above mentioned techniques are not sufficient, which might be suitable for applications needing a lot of configurability. The SystemVerilog constructs listed above might give RTL code that is hard to read, code generation can give RTL code that does not contain the syntax for configurability, making it easier to read. Code generation can be used to generate ports, logic inside modules and create new modules [12].

2.4.2 Run-time configurability

Run-time configurability is the ability for a system to change its logic when the system is running. This can have many use cases, such as dealing with problems with the ASIC after tape-out, such as correcting problems related to ageing or reconfiguring functional units (e.g., Arithmetic logic units (ALUs), multipliers, dividers) [13]. Run-time configurability can be implemented as a set of control or status registers, which can be controlled with software, by writing and reading to the registers. Ericsson uses Advanced Peripheral Bus (APB) communication protocol as the software controlled interface to and from the registers.

Chapter 3

Background and requirements

In this chapter, the general architecture of a CCR is described. These functionalities have been observed in old designs, but is observed to have been implemented in slightly different ways for specific requirements for each project. The sources used to describe the CCR come from Ericsson, including design specifications, code analysis, and discussions with CCR experts, with the workflow described in 4.1.2. The requirements for the configurable CCR will be presented after the background, discussing the focus on the design in the thesis.

3.1 CCR place in system

The CCR is initialized once per ASIC. It is responsible for boot-up sequences, system resets, PLL initialisation and clock delivery on the ASIC. The CCR is connected to multiple Local Clock and Resets (LCRs) seen in Figure 3.1. The system reset can perform warm resets to certain parts of the ASIC called reset groups. The CCR receives one or many reference clocks coming off-chip.

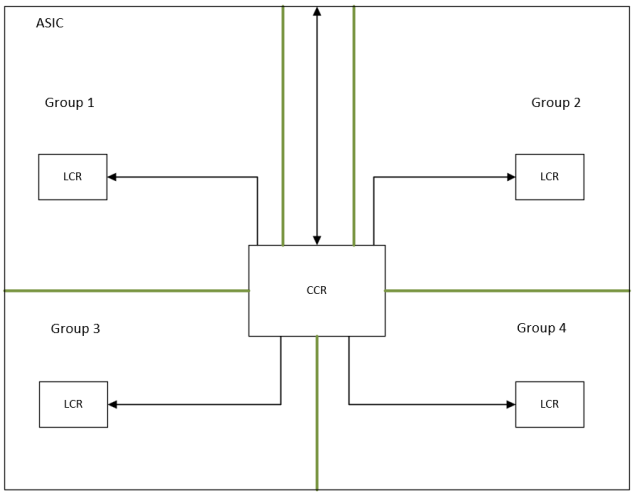

Figure 3.1: Showing CCR place in system, with reset groups, LCR connections and signals to/from the ASIC.

LCR is a module responsible for clocks and resets on a section of the ASIC. The LCR takes the main clock from a PLL, delivered by CCR, and performs clock division on it to generate multiple clocks of separate frequencies. It also takes a reset signal from the CCR and synchronizes it on de-assertion. It also handles clock gating by having ports that can be used to clock gate specific clocks. There is at least one LCR per reset group.

OTP controller The One-Time Programmable (OTP) controller is responsible for programming of the OTP memory, which is a non-volatile memory that can only be written to once. The CCR block can read from the OTP controller to configure certain settings after creating the chip, but only once.

3.2 PLL usage in CCR

As described in the Theory section, a PLL can be used to generate one or multiple clocks from a reference clock. A CCR architecture can have multiple PLLs to generate separate clocks based on separate reference clocks, among these it has one called system PLL that creates a clock that is sent to the regional LCRs. The PLLs in the system are used and driven in two main ways. The first is driven by the CCR when the ASIC is booted, which is what drives the system PLL. Second is driven by software, when the ASIC is already booted. The module driving the PLL is responsible for: delivering a reference clock, driving PLL enable and reading PLL lock, among other things.

3.3 CCR functionality

This section gives an overview of the functionality of the CCR. Generally the headers in the subsections are RTL modules instantiated by CCR IP, with the exception being: reset glitch filter, clock observation and slow clock generation.

Reset glitch filter is used to filter out glitches on external reset input pads. This is done by putting the reset through delay buffers and then putting the delayed and non delayed input into an OR gate. This is the same method shown in figure 2.9. This means that for an external reset to be detected, it must be held for a certain amount of time. Thus, glitches on the pad are filtered out as they will tend to be too short to go through the filter.

Clock observation is a feature in the CCR where it has an output pad that is used for observing internal clocks. This can be clocks in different parts of the ASIC as well as clocks in the CCR. It can also perform clock division on the observed clock to make it slower. What clock to observe and what to divide it with can be decided through software by writing to registers.

Internal LCR is a LCR used internally by the CCR. This LCR has a generated clock from PLL as input, and from this it divides it to generate local clocks, which are used locally within the CCR. The internal LCR is the same RTL IP as the LCR, with the difference being the internal LCR doesn't use all the features of the LCR.

LCR sync generation is a module that generates a pulse that is used to synchronize divided clocks in the LCRs to the CCR main clock that is generated from the system PLL. This is done in order so that all LCR clocks are synchronized for all reset groups.

Slow clock generation is a module that generates so called slow clocks that are based on the reference clocks directly (not the PLL clock) by using clock division. These clocks can both be used internally for certain CCR blocks as well as be outputs from the CCR to other parts of the ASIC.

APB register file contains a collection of registers that can be accessed from software through the APB protocol. These registers are used by software to write and read to modules during run-time. Example usage include setting configurations for PLLs and selecting what clock to observe from the clock observation pad.

Synchronizations for CDC are used internally in the CCR, when separate blocks use different clocks. The reasons for this differ, but one example is the reference clock domain used in reset FSM and the APB clock domain in CCR register file. These CDC need to be handled with synchronizers. An example of a CDC is when a PLL status signal is connected to an APB register. The reason why this is a CDC is because the PLL runs on its clock and the APB register runs on the APB clock.

Sysref capture is a module that monitors an external pad to the ASIC that is called sysref, which stands for system reference. This is a signal that is used to measure the timing difference between the internal clocks generated from the PLL and the reference clock. This is done by generating a snapshot waveform of the reference clock, sysref and an internal signal called sysref_int. Sysref_int is generated from the sysref signal by first synchronizing it with the reference clock and then the PLL clock. This snapshot is then stored in the APB registers and can be read from software to evaluate the timing.

Reset request handler checks if software has sent a system reset request or POR request, meaning that software tells the CCR to perform a reset. Software does these requests by writing to a 16 bit wide register, and when this matches the specific 16 token for that reset request, reset request handler will send a pulse to reset FSM, telling it to perform a reset. The 16 bit register with token matching is used to protect against accidental resets. The reset request handler also has an input for external reset, which triggers system reset pulse to reset FSM, when asserted. When a reset is requested by software or external reset pad, the reset request handler updates registers that stores information about what caused the latest system reset. The reset source and cause registers are clear on read. Because of this, some instances of the module have two register pairs for reset cause registers. Then two sources can read the one register each of the register pair.

The reset request handler can be one of two types, either primary or secondary reset device. If the secondary device wants to do a reset, it sends a reset request to the primary device. The primary device can act on this request and can issue a reset to the secondary device. When it is a secondary device, the reset request handler monitors local reset sources on the ASIC, like timeout of watchdog timers, leading to a reset request. Registers are used to tell what was the source of the local reset request. If the device is a primary reset device, it acts on the secondary devices reset requests through a reset request input vector. In primary mode, the device has a feature called external reset gating. The feature is used to block external resets from registering when the reset FSM has raised the reset imminent signal, which is a signal sent throughout the ASIC telling all modules to prepare for a reset.

Reset sequencer is a part of the CCR that handles most of the system reset. This is done in the form of resetting all the reset groups one by one. It also gives the capability for software to disable individual reset groups, so that they don't get reset during this process. Non enabled reset groups still take a clock cycle to check if it is enabled or not. It can also be set how much time should be between each reset assertion.

Reset FSM is a block that handles all the different tasks that need to be handled during the reset off the ASIC. This can be things such as loading settings from OTP, performing memory repair, locking the PLLs, asserting PLL enable, deasserting internal POR, etc. There are two types of resets that it performs, POR and system reset. POR is asserted when the ASIC is booting up, but can also be activated by software during run-time. System reset is mostly performed by

the reset sequencer, but activated from the reset FSM. There is also handshaking between the two blocks to ensure that the reset FSM only goes to the next state when the reset sequencer is done. When a software POR is asserted it also performs a system reset where all the reset groups are reset, not just the enabled ones.

3.4 Configurable CCR requirements

In this chapter the requirements for the project will be presented. This will be in the form of higher level targeted features and goals for implementation. As will be apparent, the goal is to make the CCR as configurable as possible in order to make the job of design and verification easier and less time consuming. This will be both run time configurability, but also compile time which is often chosen due to APB registers not being available at boot time.

Configurability is the ability for a design to change its behaviour in accordance to changes in requirements. A design that is more configurable thus has more possibility for potential changes in the design. The changes of the configurable design is defined through parameters. Examples of different constructs to achieve configurability are found in section 2.4.

3.4.1 Performance requirements

Requirements on area and power

As the CCR is a module only instantiated once and is a very small part of the ASIC the requirements on area and power are not very strict. However, some implementations done in the thesis can be used outside of the CCR such as clock dividers, clock muxes and reset FSM. Thus the performance of these IPs should then be investigated, with area and power being important metrics. The compile-time configurability should not lead to more area and power compared to previous designs.

Requirement on less development time

Using the configurable CCR should reduce the time spent on design and verification for new CCR designs. This is the main point of making a configurable design rather than writing brand new RTL code for every new project. If it takes more work and time to configure the design, then there is no reason to use it and the developers might as well write a new design. This requirement is for all designed modules. To quantify this requirement would be very difficult, as different engineers will take different amounts of time to do the same design. Thus, the analysis is more based on design decisions that should lead to more or less development time, not

necessarily how much as that could not be measured. Arguments for why some decisions will reduce development time, increase it or not affect it will be done on a case by case basis.

3.4.2 Functional requirements

PLL wrapper

Design a PLL wrapper that can be used to integrate PLLs from current PLL vendors, and future ones. Currently, projects use PLLs from different vendors with different interfaces, which means that new RTL code must be written to drive the new pins. With a general wrapper it could potentially be made easier as the designer would just have to connect the PLL to the ports in the wrapper and then the rest of the design should still work, which can reduce the time and complexity of integrating PLLs in design. The wrapper could also reduce verification effort, as the verification infrastructure surrounding the PLL wrapper can remain the same between projects. Multiple wrappers should be instantiated in an efficient and user friendly way, further reducing time integrating and complexity.

What makes a good wrapper? This is a question that is hard to answer. Source [14], mentions the intent of a wrapper is: "Wrapper allows adapting an interface and behaviour of the IP component to the context of a given application", and that "Many different wrappers can be applied to the same component, as well as the same wrapper can be applied to the different IPs". This means that a good wrapper reduces the time spend on integrating differences of one type of IP, which is done by converting from one interface to another. Source [14] also signifies the importance of abstracting hardware designs, and that a wrapper module could aid in this. A good wrapper needs to then be both reusable and provide a good abstract interface. The problem with this definition is that it is hard for a designer to quantify these requirement, and that these requirements depend on personal preference.

Configurable CCR architecture

Propose a CCR architecture suitable for configurability. The new configurable CCR architecture should have a block hierarchy that enables the CCR top module to have a fixed interface. This will save time in verification, since the verification framework always targets the same interface, which means that testcases and functional cover points can be reused. The top module is configured through module parameters and in some cases software registers for run-time configurability.

The modules inside the configurable CCR should be made configurable to meet the requirement of a configurable CCR architecture. Reset request handler should be able to configure for the different reset request handler specifications presented in section 3.3, as well as make configurability for future design changes. Reset

glitch filter and LCR sync generation are simple modules, but their possibilities for configurability should be explored. Clock division, clock observation and reset FSM requirements are more extensive, and this will be presented in sections below.

Clock division

Be able to generate slower clocks that are based on faster input clocks, this should be in the form of specifying a divider for said clock. In previous Ericsson projects, clock division has been handwritten for each individual case. The point of making a configurable block is that it becomes easier to implement, reduce risk of bugs, and potentially help with verification as will be discussed in the discussion part of this thesis.

Clock observation

Be able to select what clock to observe on the observation pad. Should also be able to decide what to divide said clock with. Options for clock muxing will also be explored. Similarly, this has also been handwritten for each new project. Thus making one configurable design could help in the same ways as for the clock division block. Adding options for clock muxing is a more exploratory part of the thesis. It is something that could perhaps be used in the future if avoiding glitches (and doing so quickly and safely) for the observation clock is required.

Reset FSM

The reset FSM should be made run-time configurable, and programmed in a user friendly way. The advantage of using a run-time configurable FSM is that it can correct for design errors not found during bring-up of the ASIC. Since the CCR is critical to the ASIC, this can be the difference between the ASIC being able to boot or not. Examples of things that could be changed during run-time are: state timeouts, the order of asserting reset groups, state order, etc.

APB clock, commonly used for configuring run-time parameters, is not stable when running parts of the reset sequence. Instead, The OTP controller is enabled in an early stage of the FSM, and can thus be used to configure the logic of the states after the OTP controller is enabled. This programming can only be done once per ASIC, but still opens up the possibility to save an ASIC after tape-out.

Chapter 4

Design and method

This chapter will present the thesis workflow and the implemented hardware and software designs. It will also go through the design choices made and then present block diagrams or descriptions of the designs. This chapter presents all that has been implemented and will not show synthesis results, discussions or proposed but not implemented implementations (with proposed architecture as exception), which will be presented in chapter 5.

4.1 Workflow

This chapter explains the general methodology used in the thesis. Technical details will not be presented in this section. The workflow of PLL wrapper is similar to the reset of the CCR and will thus not be elaborated on in this section.

4.1.1 Literature study

The first step of the thesis was to obtain a good theoretical background on the topic. To find sources, these search engines were used; google scholar, Lund University database, google search and Artificial intelligence (AI) chatbots, such as ChatGPT and Microsoft Copilot. The background in the topic was needed to understand the details of current CCR designs at Ericsson. From this, the relevant information was summarized in chapter 2. The plan was also to find other similar architectures to CCR from other sources. These types of design was however difficult to find, perhaps because complex CCR designs are found within large complex ASICs, which is usually confidential.

4.1.2 Ericsson CCR analysis

The thesis analysed three CCR design at Ericsson to analyse, where one was older, one recent and one in development. Firstly the focus was to understand the recent CCR architecture, since this was a stable IP with finished development. The analysis was done by reading Ericsson design document, reading the RTL code and talking to Ericsson experts. The goal was to understand the CCR architecture with all the design choices and the interconnections of the modules inside CCR. After this, the recent CCR iteration was compared to the other two iterations of CCR, comparing the differences in the architecture and internal modules. The comparison was done in order to map the differences in the designs, and from that, together with experts at Ericsson, analysis was done on the future of the IP.

4.1.3 CCR IP design

When designing the CCR IP, a lot of planning was done before the RTL coding began. Since the blocks inside the CCR depend on each-other, the first action was to plan the design of all blocks, in order to avoid redesigning, and to find architecture easy to scale with future changes. This planning was done by first writing the interfaces of all the blocks, and later do block diagrams of interconnections of the blocks. After this, the CCR RTL blocks were designed in SystemVerilog and verified by doing direct verification. Direct verification was used because it requires little time to set up. It has the drawback of being less reliable then using constraint random verification, or formal verification. Formal verification was not considered during the thesis but could have been a good way to verify the designs. Script-based code generation was used in some places in the thesis to ensure correctness and predictability, while LLM-assisted code generation was not considered.

At the time of doing the thesis, the CCR block was in active development at Ericsson. The consequences of this was that some modules would need constant redesign to not become obsolete. To avoid this, focus where put on what general features were stable in the CCR designs and what features are probable to remain the same in the future. The modules or features that were developed are: clock observability, clock division, reset FSM, reset request handler, LCR sync generation and reset glitch filter. Modules not design are: reset sequencer and sysref capture. Reset sequencer changed late in development and requires more time and effort to make configurable. Sysref capture is a complex module that has remained the same in all previous CCRs, so it was not the focus of configurability. Due to these reason a top module was not designed, and instead the configurable CCR blocks can be used in the future when designing a CCR.

Power and area data were collected using the Ericsson synthesis tool targeting a specific ASIC. The synthesis results are presented in points were taken by setting different parameters for the analysed IPs. The graphs were done using numpy library in Python.

4.2 Proposed CCR architecture

To achieve a fixed top module, a new architecture for the configurable CCR modules is proposed but not implemented, with a block diagram seen in Figure 4.1. It was not implemented since all the modules in the CCR were not implemented and that the top module would likely become outdated version of the CCR, with more discussion about this is found in 4.1.3.

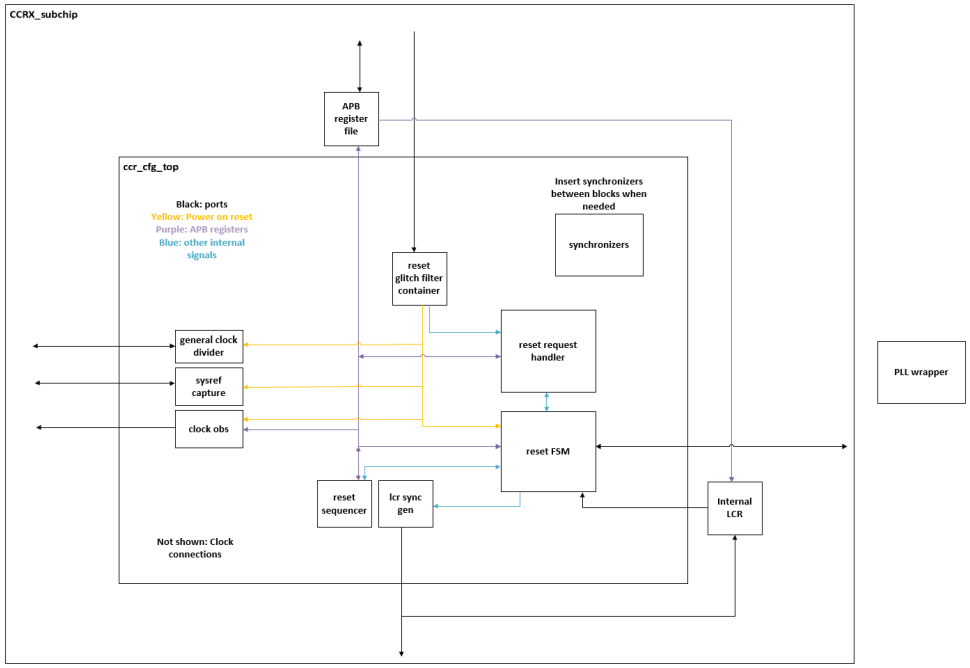

Figure 4.1: Proposed module placement for configurable CCR.

The main difference between the proposed architecture to previous designs is that the APB registers and internal LCR is moved outside of the CCR block. In current designs, the APB registers contains data for blocks other than CCR. If those blocks change their APB interface, the top interface of the CCR block needs to be changed. Putting the APB outside of the CCR, enables the top interface to have the same APB signals but with configurable width. It was observed that the interface of the internal LCR changed between projects, but required functionality for the interface remained the same. Similarly to APB register, moving the internal LCR outside of the top enables a fixed top.

4.3 Simple CCR configurable modules

This section covers configurable module implementation for reset glitch filter, LCR sync gen, and reset request handler.

4.3.1 Configurable reset glitch filter

In order to filter out noise on the reset pad, the reset signal is sent through an amount of delay buffers, and then both the delayed and the non delayed reset

signals are put through an OR gate. There are two types of resets that are sent from the pads: POR and external resets. External resets cause a system reset. There were two aspects that were identified as potentially being configurable: how many external resets that are used and the delay for the filters (same delay for the POR). A reset glitch filter can be seen in Figure 4.2 where the amount of delay buffers can be changed between different chips at compile-time. Adding more complicated configurations and potential runtime configurability is discussed in section 5.2.1.

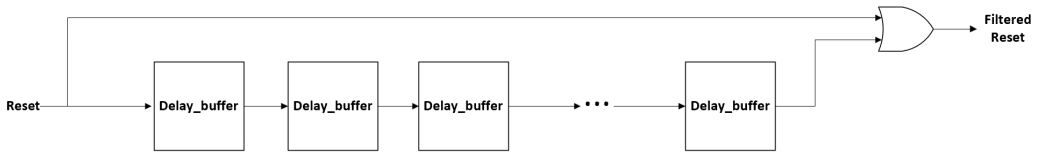

Figure 4.2: Reset glitch filter with variable number of delay buffers.

4.3.2 Configurable LCR sync generation

The sync generation works by having a counter and sending a pulse every time the maximum value of the counter is reached. It was identified that it could be made configurable how many clock cycles there are between each pulse. This is configured during compile time. While the amount of clock cycles between each pulse has been constant for the designs that were analysed, it could be changed in the future which is why it was added as an option.

4.3.3 Reset request handler

The reset request handler was made configurable from the design description in 3.3. Figure 4.3 shows the implemented design. In the figure octagons indicate functionalities in reset_ctrl that is not implemented as it's own RTL module. The orange arrows are the signals that connect directly to and from reset_request_-handler, such as input software reset requests and output register values. All modules in the figure use the same clock and reset.

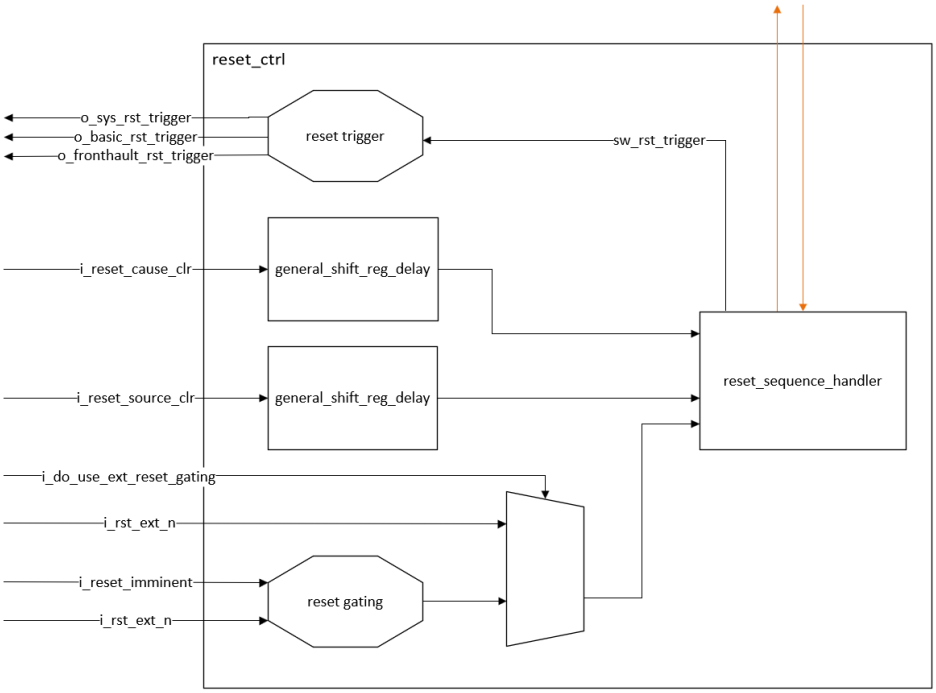

Figure 4.3: Showing the logic of reset_ctrl and reset_request_-handler.

The reset request handler functionality was split into two modules: reset_request_handler, which is a parametrised reusable model, and reset_ctrl, a module that instantiates the reset_request_handler and inserts project specific logic around it. This was done to make the model more reusable, similar to what is proposed in source [1]. In this way, the reset_request_handler needs initial verification and is then a stable module set with parameters. reset_ctrl exists to keep reset_request_handler constant and not to contain specific logic inside it such as delay flip-flops. This divide will hopefully save time in design in that the reusability though setting parameters saves in redesign. It is also likely to save time in verification in that the module ports of reset_ctrl will remain mostly consistent between redesigns. Alternative design considerations was to use compiler directives instead of module parameters, read about in section 5.2.6. Another alternative was to keep the design as one module which is discussed in section 5.2.1.

Table 4.1 shows what parameters can be statically configured for reset_request_handler. The reset gating feature was decided to be a run-time configurable feature in reset_ctrl, by having a mux to select the gating logic or bypass. This could be done since the reset gating is used when the CCR already has enabled clock to software. This created little area and power overhead, but as a critical feature, having it compile-time configurable risks it being set incorrectly. Additional features were moved from original reset_request_handler to reset_ctrl: delay of the register clear signals, and conditions for system reset trigger.

Table 4.1: Table of Reset request handler module parameters.

Parameter name	Allowed value	Description
`p_nbr_of_cpus`	>0	Set how many registers are used for reset cause and reset source. The generated registers are copies.
`p_nbr_of_ext_rst_pads`	>=0	Sets the number of external reset. Also sets the number of output external reset cause registers.
`p_nbr_of_rst_sources`	>=0	Sets how many sources are used. Also sets the number of output source registers.
`p_nbr_of_warm_rst`	>=0	Sets how many warm requests are used. Also sets the number of warm reset cause registers used.
`p_nbr_of_sw_rst_cmd`	>=0	Sets how many reset commands can be sent by software.
`p_width_of_token`	>=0	Sets the width of tokens used in software reset commands.
`p_command_tokens`	–	A matrix of token entries. Width of `p_width_of_token` and length `p_nbr_of_sw_rst_cmd`

4.4 Configurable clock division

It was identified that in some CCRs the reference clocks were divided to generate slow clocks that could both be used in the CCR as well as in other IPs that might need to start earlier then the ones that use the PLL clock. The decision was made to centralize this into one block. This block can take a variable number of clocks as inputs and then output a variable number of output clocks. For every output clock, it can be selected what input clock it is derived from and what to divide it with. This was achieved by having a general block that can perform any clock division (for integers) and generate a 50% duty cycle, this block is called clk_div_50dc_wrapper. For even integers, a counter that counts for half the specified value and then flips its clock output is used. For odd integers, the method specified in section 2.2.3 was used. If a division of 1 is specified, then the output clock is simply directly connected to the chosen input clock. One of these blocks is instantiated for each output and connected to the corresponding input clock. Internal resets are generated and synchronized for each input clock. Each division block is then connected to the reset that is synchronized with the input clock that is used for that block. An example of how the clk_div_50dc_wrapper can be connected, as well as how the resets are connected to the blocks, can be seen in the upper half of Figure 4.4. In this example, the upper three clock outputs are not dividing with a power of 2 and two of them are connected to the same input clock.

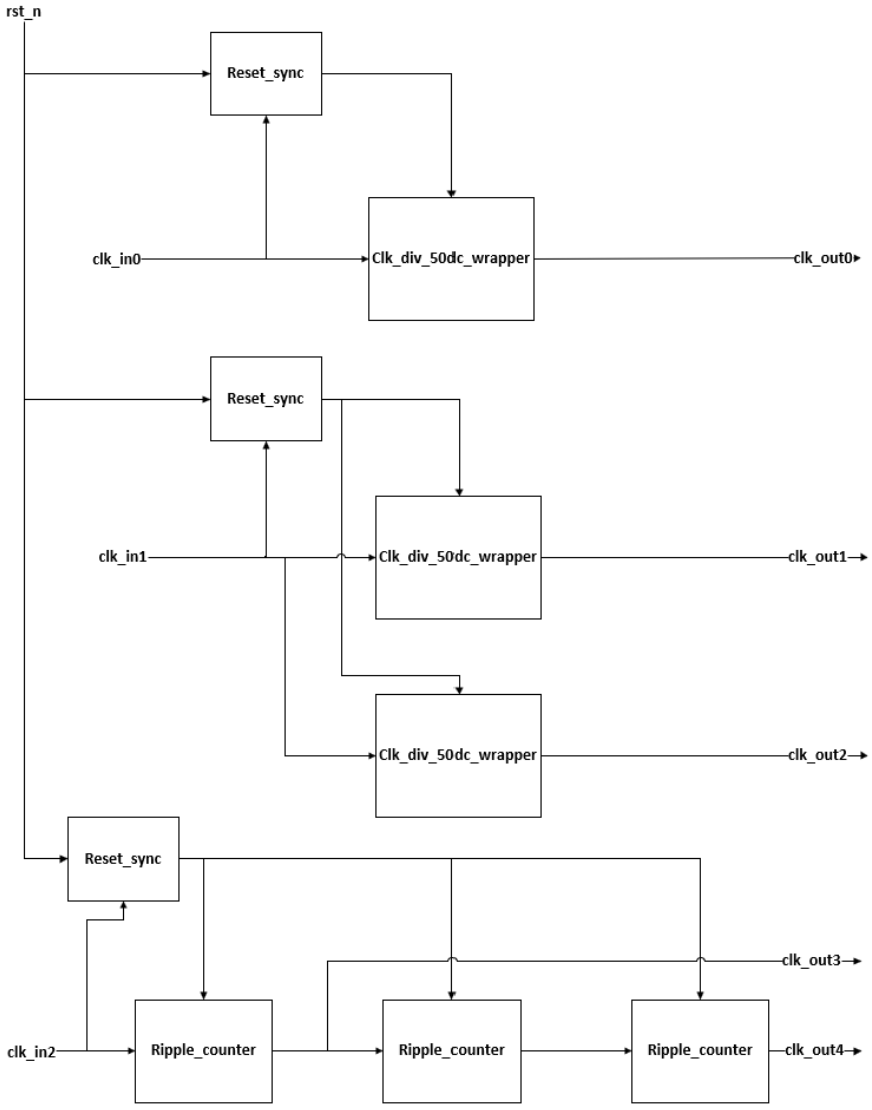

Figure 4.4: An example of the configurable clock division block for some configurations.

This implementation is functionally correct, but in many use cases might be a bit wasteful in terms of area and power usage. Something that was often done in past CCRs was to divide the same clock with multiple power of twos. In other words 2, 4, 8, etc. This can be done using a pipeline of ripple counters. As one of the requirements is that area and power should remain about the same, an optimization was done that makes it so that if a division of a power of two is selected, then a ripple counter pipeline is instantiated. An example is to have two output clocks where one is divided by 8, $div8$, and the other is $div2$. Then a pipeline of three stages will be instantiated, where $div2$ is connected to the output of the first ripple counter flip-flop and $div8$ will be connected to the third, this example can be seen in the lower half of Figure 4.4. This will thus save area in comparison to using the more general counters and lead to the power and area being closer to what it has been in hand written CCRs in the past. The optimization could be done automatically because of the decision to centralize clock division in one block. It can also help with verification as is described in section 5.2.2. It is also discussed how this kind of approach can be applied more generally to other configurable designs.

4.5 Configurable clock observation

4.5.1 Top level

For clock observation, it was made configurable the number of clocks that can be observed, the number of options for what to divide the chosen observed clock with, what integers the clock can be divided with and the number of clock observation outputs. Each observation output has its own control signals so that separate outputs can observe and divide independently from each other. In Figure 4.5 the block diagram of the clock observation block can be seen. The clock observation is divided into three separate blocks, two of which are the same type. The general clock divider is the same one that was presented in section 4.4. The clock_mux_-multi is a clock mux that can take in any number of clocks and then select between them. The clocks that can be observed go into a clock mux. Then the selected clock goes into the divider that performs all the specified divisions. The divided clock outputs are then put into another clock mux that is used to select which clock division to perform. This block thus gives out the clock observation output.

Another approach would have been to perform the division for each input clock and then put them into one big clock mux, this would also solve some of the complications with this structure when glitch free clock muxing is required (as will be shown below). However, it would massively increase the area and power as significantly more clock dividers would be needed, and thus this approach was not chosen. Another approach would be to design a runtime configurable clock division block to use instead of the general clock divider and second clock mux. This could greatly increase the number of divisions that could be performed. However, ensuring glitch free transitions when switching between different divisions would

become more complicated. It was thus decided to not go with this approach as it would take more time, time that could be spent on other parts of the thesis.

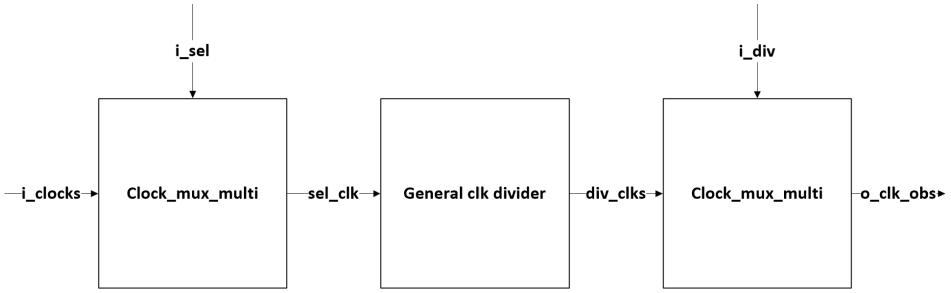

Figure 4.5: Top view of the clock observation block for one output clock.

4.5.2 Glitch free clock muxing and how muxes are connected

It was added as an option whether to use normal muxes or glitch free clock muxes. Glitch free muxes are usually made to take in two clocks and then select between them. In order to select between more than two clocks, muxes need to be connected together, at least for a conventional glitch free clock mux. An example of how a group of muxes can be connected is shown in Figure 4.6, in this case for ten input clocks. In order for it to be configurable how many clocks can be selected between, there needed to be a way to automatically build a tree of muxes. A discovered idea was to define $number\ of\ connection\ layers = clog2(number\ of\ clocks) + 1$. $clog2$ calculates the logarithm of two and then rounds up. The first layer is connected to the input clocks. For the rest of the layers, a signal is connected to a clock mux where the input ports are connected to signals from the previous layer. A signal k will be connected to the signals $2k$ and $2k + 1$ in the previous layer through the mux. The only exception for this is if k is the last signal in a layer and the previous layer had an uneven amount of signals. If this is the case, then k is directly connected to $2k$ instead of going through a clock mux. The number of signals needed for each layer is calculated recursively. For the first layer, the number of signals is set equal to the number of input clocks, for the next layer, it is equal to the previous one divided by two, and the next layers are set in the same way. The division by two in this case is rounded up, for example: 5 divided by 2 is equal to 2.5, so it is rounded up to 3.

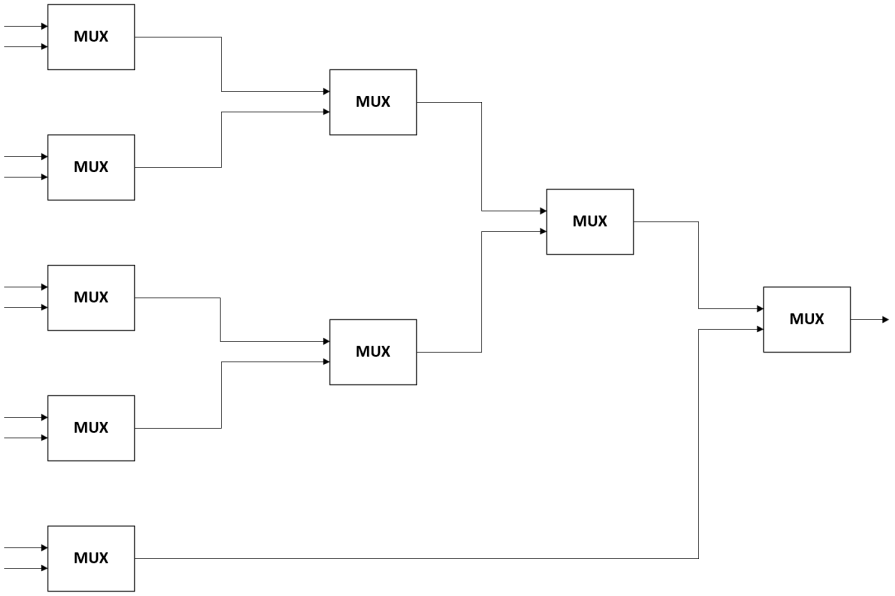

Figure 4.6: An example of what a tree of muxes looks like when there are ten input clocks.

4.5.3 Glitch free transitions when dividing the clock

The above implementation for the clock_mux_multi will result in glitch free transitions for that block if configured for it. However, for the whole clock observation block there was still one case where the clock output could contain a glitch. This could happen if a divided clock was chosen (division of more than 1) for the output and then i_sel changed. This is because clock division is implemented with counters and the input clock to the divider might disappear while the divided clock is high, resulting in the observed clocks high period being stretched as the divider can't count without a clock. To fix this, the observation clock should be turned off when the clock goes low, the clock divider should be reset, the first clock mux should change the clock being observed and then the clock divider and observation output should be enabled again. The way this was implemented is presented in Figure 4.7. The select signal is first synchronized to the negative edge of the observation clock. It is then compared with a select signal synced with the positive edge of the selected clock from the first clock mux. The input to this synchronizer comes from Sync_neg. If they are different, then the clock divider will be reset and the output clock will be disabled. At the same time the select signal to the mux has now changed which results in the clock output changing. The second synchronizer will result in the clock divider and observation clock output being enabled when the clock has changed. All this results in the transition between different clocks being glitch free even when the clock is being divided.

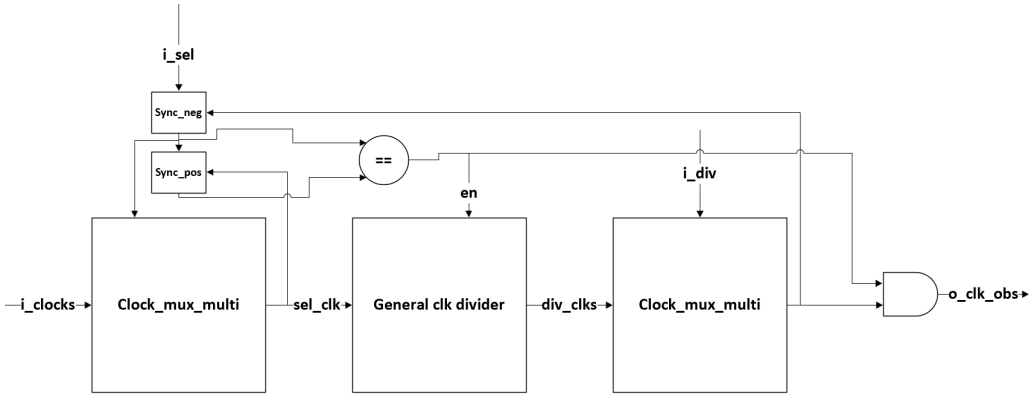

Figure 4.7: The top view of the clock observation block when it is configured to be glitch free.

4.5.4 Multi bit CDC

Most likely, the select signal connected to the first synchronizer in Figure 4.7 (Sync_neg) comes from a separate clock domain. Using a two flip-flop synchronizer for each bit here might cause a problem as the bits might arrive at different times (after either two or one clock cycles) [15]. The main reason this would cause a problem is because the clock used for synchronization (the observation clock) disappears when the select signal changes. This in turn means that the wrong clock could be selected before switching to the right one. This leads to it being crucial that all the bits for the select signal arrive at the same time. For this, the mux recirculation technique was used. In Figure 4.8 it is shown how mux recirculation is done conventionally. Assuming that the bits being transferred are held stable, then as the enable signal is synchronized, it will ensure that all the bits are changed at the same time. Since software does not send an enable signal, this circuit could not be used without modifying so that the circuit generates its own enable signal. The implementation that was made is presented in Figure 4.9. When it is detected that the data from the source domain is different then the data in the destination domain (in other words the data has changed) the enable signal is set high. Otherwise the circuit is the same as a standard recirculation mux. This implementation is done with the assumption that the source data won't change very frequently. As it will be used for the select signal for which clock to observe it won't change very often as there needs to be enough time to observe the newly selected clock.

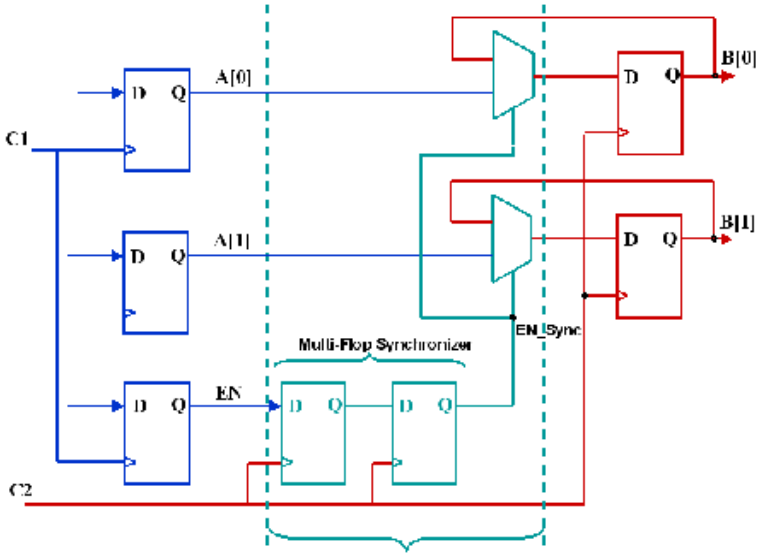

Figure 4.8: Mux recirculation technique for two bits, taken from [15].

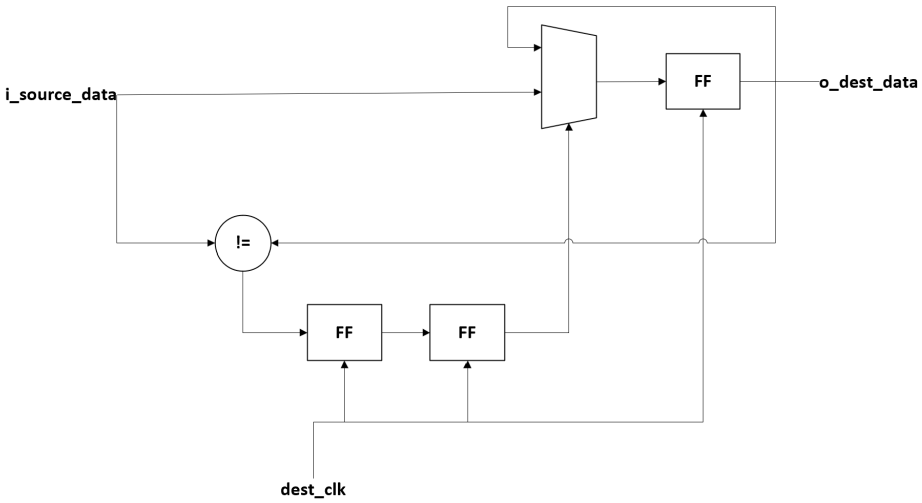

Figure 4.9: Circuit for CDC when multiple bits of a signal are changed.

4.5.5 Low latency clock mux implementation

As stated before, clock muxes for 2 clocks are usually connected as a tree in order to be able to switch between more than two clocks. However, this can in some cases cause very slow transitions between clocks. This can for example happen if there are very slow clocks in the mux network. The proposed solution was to design a clock mux that uses one-hot encoding instead of natural encoding. In Figure 4.10 a conventional clock mux network is presented as an example were four clocks can be selected between. In Figure 4.11 a one-hot encoded clock mux for four clock signals can be seen. The one-hot clock mux will result in a transition between two clocks at most taking two clock cycles of the current clock and two cycles of the clock that is being switched to. It will not be affected if there is slower clocks that can be switched to. The reason it can result in faster transitions is because of the one-hot encoding, as it will ensure that only one clock is enabled at a time, which means that the gated clocks can all be connected to the same OR gate.

In Figure 4.12 a transition between two clocks is presented for the conventional clock mux tree, in figure 4.13 the same transition is presented when using the one-hot mux instead. From these figures it is clear that the one-hot mux has a faster transition. Another thing about this design is that it uses fewer flip-flops than the conventional clock mux network, for the example in the figures it uses 8 flip-flops compared to 12 for the conventional method. This could then lead to this implementation using less area and power. The critical path for the clock will also be shorter for this design. For the conventional design it will be: $t = (t_{and} + t_{or}) \cdot clog2(number\_of\_clocks)$, and for the low latency one it will be: $t = t_{and} + t_{or} \cdot clog2(number\_of\_clocks)$. The select signal from software will be given in natural encoding so it needs to be converted to one-hot encoding as can be seen in Figure 4.14. More about the strengths and potential weaknesses of this design can be read about in sections 5.1.2 and 5.2.3.

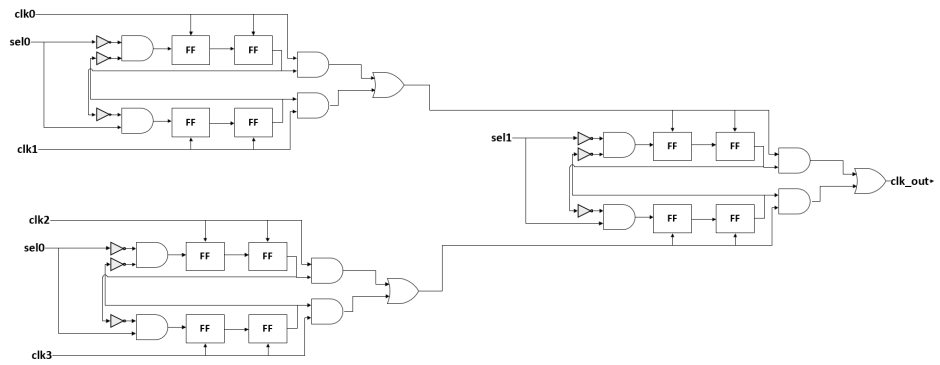

Figure 4.10: Conventional glitch free muxes for four clock signals.

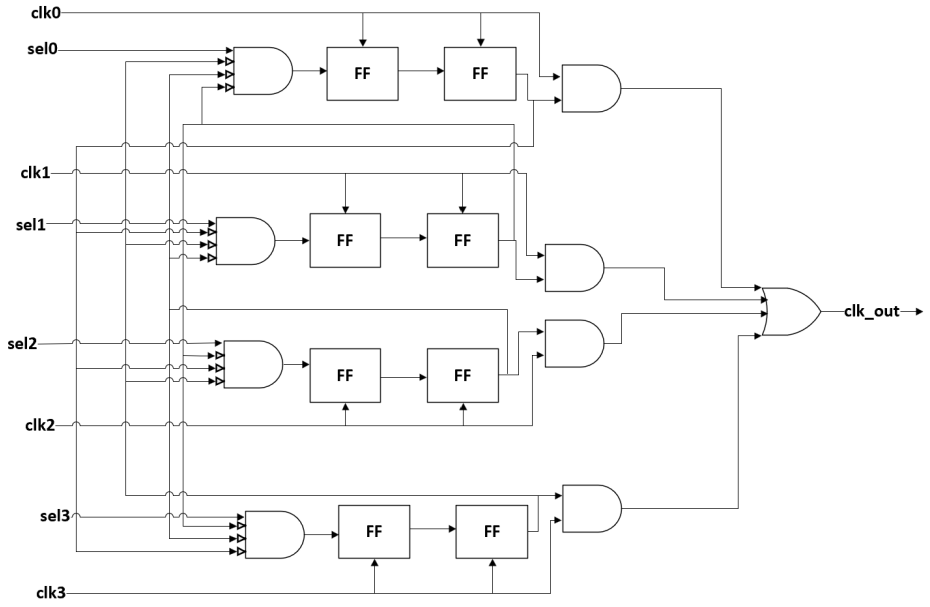

Figure 4.11: One-hot encoded glitch free clock mux.

Figure 4.12: Waveform for transition between two clocks for conventional clock mux tree.

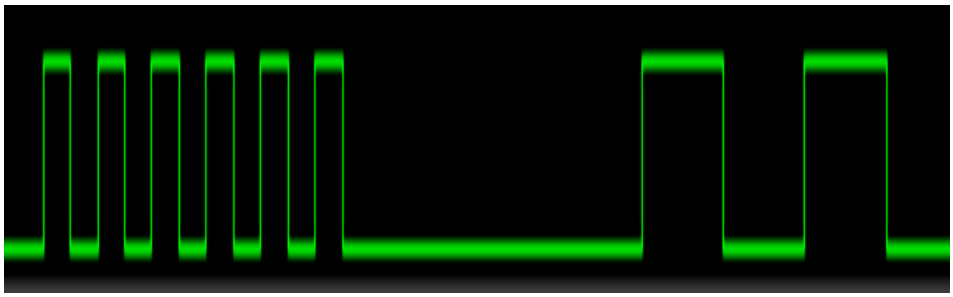

Figure 4.13: Waveform for transition between two clocks for the one-hot mux.

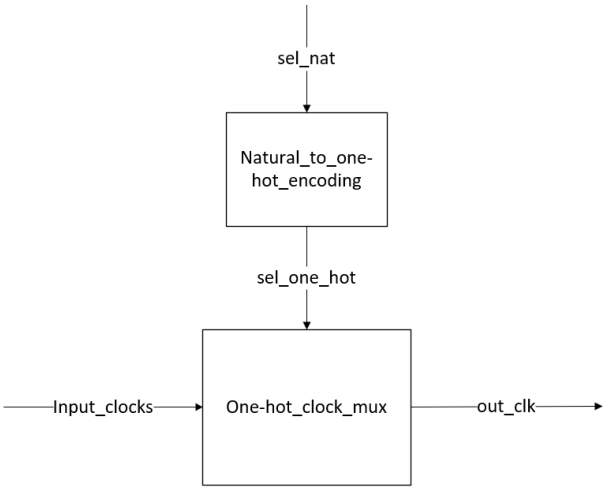

Figure 4.14: The top for the low latency clock mux.

4.5.6 Counter for if the selected clock disappears

For the first clock mux in the design, there might be cases where the clock that is selected could disappear. Therefore, it makes sense to have it as an option for the clock observation to handle this case. As mentioned in section 2.2.4 there are different ways to handle this case. For this use case it makes sense to use a counter that runs on the reference clock used for the system PLL. The reason this clock was chosen is because without it the ASIC won't function anyway, so it is very unlikely to disappear. To accomplish this feature a block called clk_gone_-counter, two muxes and an AND gate was added to the clock observation as can

be seen in Figure 4.15. The muxes are there so that the clk_gone_counter can directly connect i_sel to the signals connected to the muxes in the figure and the AND gate is used so that the general clock divider can be reset. The reset from the clk_gone_counter is also connected to the first clock mux so that it can be reset and thus change what clock is selected. It is not shown in the figure but the select signals (i_sel, sync_neg_sel and sync_pos_sel) are all connected to the clk_gone_counter. This block is implemented with a FSM that is depicted in Figure 4.16. What is done in each state is explained below. Discussion about ways to make the counter more robust using multiple clocks is discussed in section 5.2.3. These ideas were not implemented, as it would have taken more time and probably been unnecessary for the CCR.

COUNT

In this state counter(s) are used to detect when the selected clock from the first clock mux has stopped. The detection can be done for when the clock is low for too long, when the clock is high for too long, or both as can be chosen with parameters. The counters are always counting towards the max value and once reached stays at that max value, this value is chosen with a parameter. The low counter is reset when the selected clock goes high and the high counter is reset when the selected clock goes low. The counter is then synchronously enabled when the selected clock stops resetting it by using a reset synchronizer. When one of the counters has reached the maximum value and the select signal changes the state changes to RESET_NOW.

RESET_NOW

In this state rst_clk_gone_n is asserted in order to reset the first clock mux and the clock divider. Wait_for_sel_same is also asserted in order to control the muxes so that i_sel is connected to the signals from the muxes that can be seen in Figure 4.15. This is done so that i_sel is connected to the clock mux and so that the equal check (==) does not go low. Even though the clock mux can't switch clock while being reset, it is still important that its select port is held with the new select value so that when it goes out of reset the clock mux can't get the wrong value because of the CDC. After one clock cycle in this state it changes to WAIT_FOR_SEL_SAME.

WAIT_FOR_SEL_SAME

In this state the wait_for_sel_same is held high until i_sel, sync_neg_sel and sync_pos_sel are all the same value. When this has happened the clock mux has successfully switched to the new clock and thus the FSM can change the state back to COUNT.

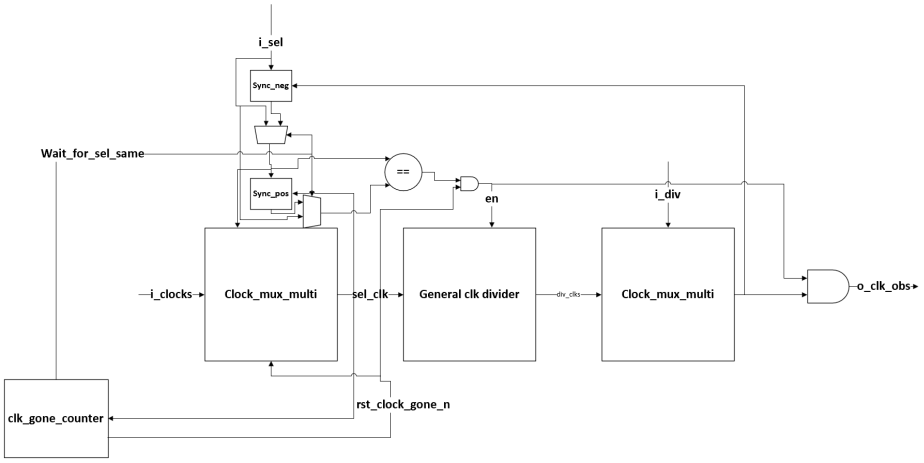

Figure 4.15: Top view for clock observation when a counter is used.

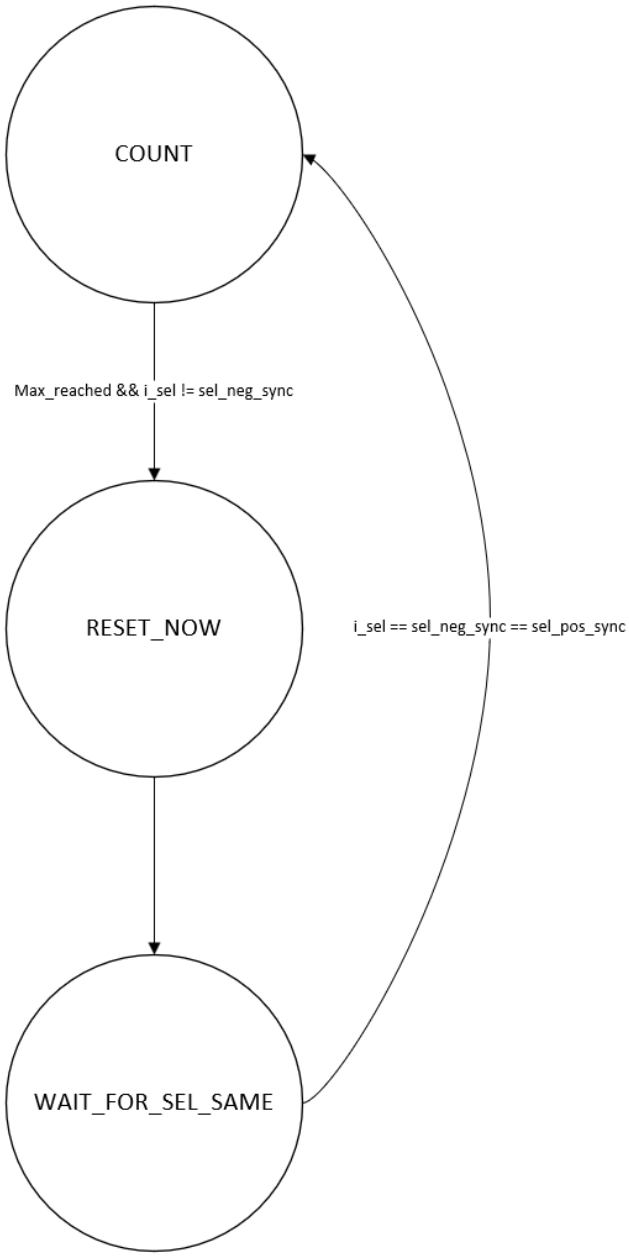

Figure 4.16: The FSM for the clk_gone_counter.

4.6 Run-time configurable FSM

4.6.1 Table based approach

One approach to program a run-time configurable FSM is to have a table where each row represents a state and each column selects some option for that state. The approach presented in this section takes the columns from such a table as inputs to decide how the FSM should behave. This design went through four iterations, each having either expanded capability or ease of use. The different iterations are presented below. Other approaches were also considered. One of these was to use an instruction based architecture that would take a certain amount of instructions for each state in order to describe what happens in said state. Another idea was to use an RISC-V processor and program it using a higher level language. The strength and weaknesses of these approaches, as well as why the table idea was prioritized, can be read about in section 5.2.5.

Base version

The table used to configure the FSM can be seen in table 4.2. The first column sets what the next state should be. The guard selects sets what guard to evaluate before going to the next state, the guards are connected to the run-time FSM from outside the module. This is so that any signal or logical statement can be connected to the guard and thus evaluated by the FSM. The output select is used to decide which output registers to set when the guard is true. The last column says what value to set the register to. This version has a lot of limitations. Firstly, the state machine can only go to one state from the current one. Secondly, only one guard can be evaluated for each state. Third, only one output can be set per state. This results in this implementation probably not being very useful for most applications. How this was implemented can be seen in Figure 4.17. The muxes with only one arrow as input on the long side select among all the signals that are part of that vector. The current state register is used to pick which row to read from the table. The guard is then used to check if the state and chosen output register should change value.

Table 4.2: Table for the base version of the run-time FSM.

Next state	Guard select	Output select	Guarded output

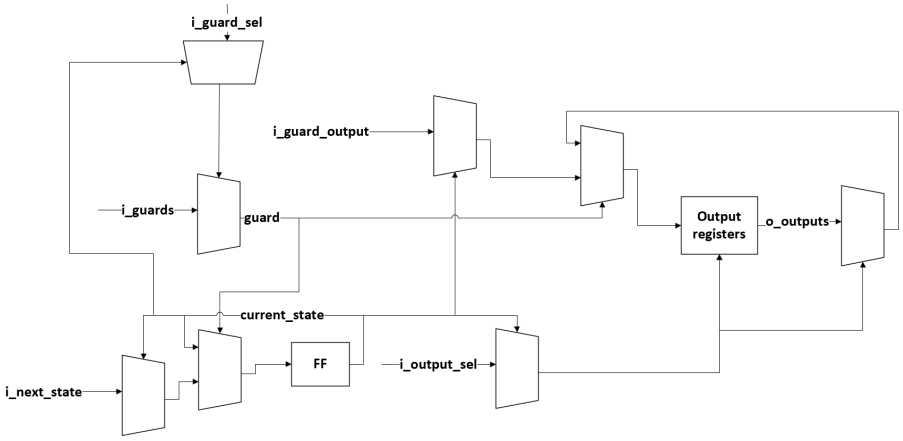

Figure 4.17: Implementation for the table based run-time FSM for the base version.

Conditional FSM

This version takes the previous version and splits column "Next state", "Output select" and "Guarded output" into two. In this version the guard is evaluated and then based on if it was true or false the state machine will use one or the other column. This allows for a lot more complex state machines to be implemented. If it is desired for the state machine to stay in the same state until the guard is true, then the next state for the column that is used when the guard is false should be the same as the current state. This implementation can be seen in Figure 4.18. The difference between this design and the one for the base version is that the guard is now used to select between if or else inputs for the next state, the output select and the output.

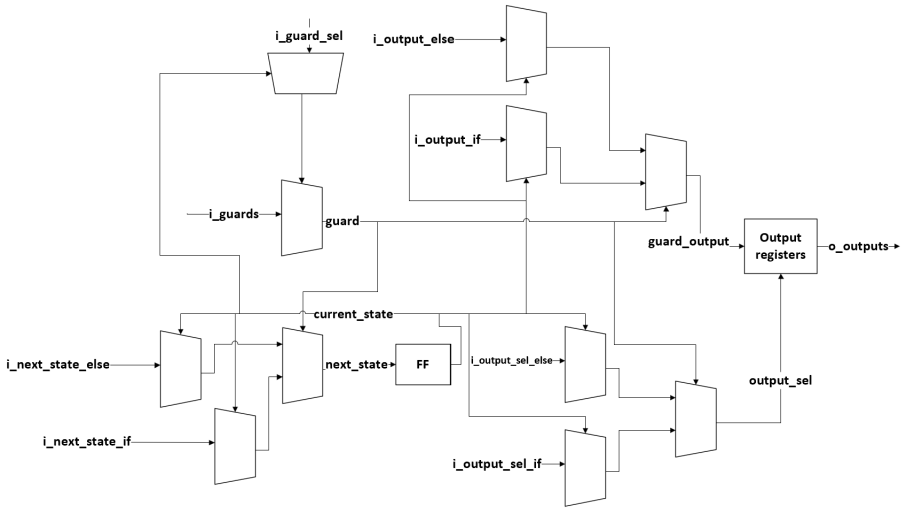

Figure 4.18: Implementation for the table based run-time FSM for the conditional version.

Multi-output FSM

One problem with the previous versions is that only one output register can be set in each state. This means that if multiple bits need to be set at the same time (for some state machine) then the previous two versions won't be sufficient to implement the state machine. Thus in this version it was added that multiple bits can be set at the same time as can be seen in table 4.3. An output mask is used to say which output registers to set, the output column then says what value to set the output registers too. The output and output mask are also split in two so that the outputs can be set based on the guard. The implementation can be seen in Figure 4.19. The mux and flip flop for the output (shown furthest to the right) is a bitwise operation, meaning that each output registers input is connected to a mux that is controlled by the corresponding mask bit. If the mask bit is low then the register is equal to its current value, if the mask bit is high then it is equal to the corresponding output bit.

Table 4.3: Table for the multi-output FSM.

Next state	Guard select	Output mask	Output

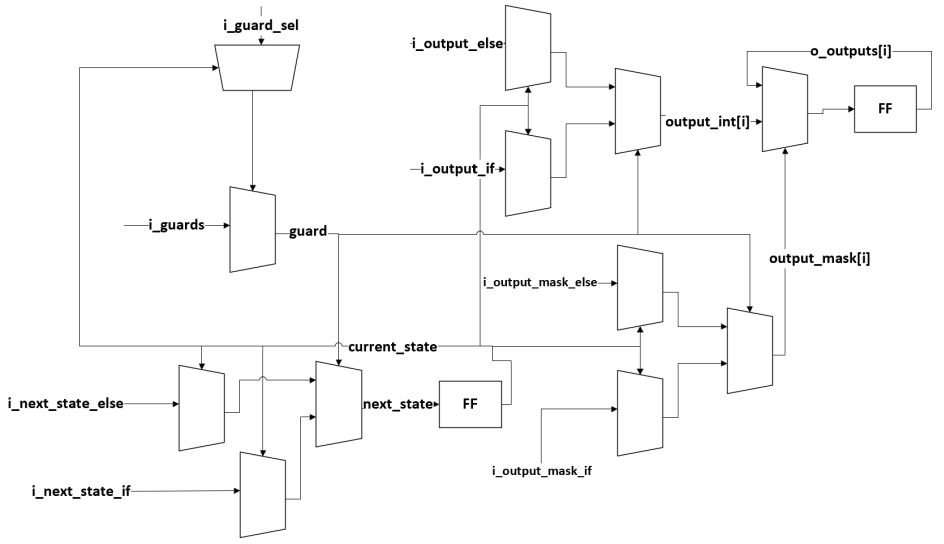

Figure 4.19: Implementation for the multi-output FSM.

Multi-output FSM with counter

A common operation that is done in the reset FSM is to stay in a state, then count to some value and then change state. The same effect can be achieved in the previous versions, but it will use a lot of states. To make it easier to program this module another column was added, as can be seen in table 4.4. This column is used to set a counter value. The counter is incremented and when the value is reached the guard is evaluated, the counter is also reset at this point to be used in the next state. If no counter is needed in a state then the counter value should be set to zero. The implementation for this design can be viewed in Figure 4.20. A counter was added that takes a counter value and outputs a signal that says when that value has been reached. This signal was then used to say if the state should change by using a mux. It was also connected to an AND gate that will cause the output mask to only be valid when the counter has reached its value.

Table 4.4: Table for multi-output FSM with counter.

Next state	Guard select	Output mask	Output	Counter value

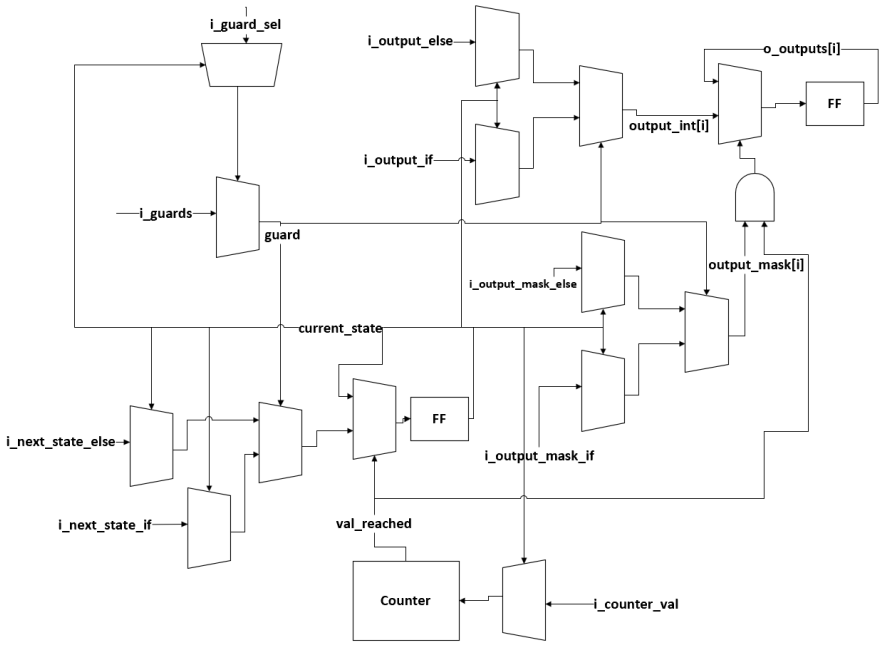

Figure 4.20: Implementation for multi-output FSM with counter.

4.6.2 FSM compiler

To set the FSM during run-time, many bits would need to be set. To make this easier a text based interface (language) was constructed, as well as a compiler that can turn the text into bits for the table. A script that can make a wrapper for the FSM was also made, the reason for doing this was to be able to have signal names and connect the guards, making it easier to integrate the run-time FSM. The scripts where made for the multi-output FSM with counter. The structure of the language can be seen in Figure 4.21. In the first part of the file, signals are declared and parameters are set. The two parameters, number of states and counter width, need to always be set. Then there are four types of signals that can be declared: input, output, reg and guard. The width of the signal can be specified for all types except for the guard (always one). The output signals will use the name written in the text file for the wrapper, and will then be connected to bits from the output vector from the FSM core. The reg type is a form of output, but it can be used to set the guard unlike normal outputs. The input type will be connected into the wrapper and then used for setting the guards. The specified guards will then be connected to the FSM core. Unlike the other types, the guard type needs to be set equal to a statement. This statement can be any valid SystemVerilog statement as long as it uses names of input and reg signals that are part of the wrapper. It can also be constant expressions, for example:

$guard = 1$. Signal deceleration can only be set during compile time, it can not be changed during run-time.

What happens in a state is described inside the *beginstate* to *endstate* construct. The first state that is written will be the state that the machine enters after reset. The first thing that is set about the state is the name, this name can then be used when other states need to go to this state. The counter value for the state can also be written on the same row as *beginstate*. If no counter is needed then that field can be left blank, as the counter value for that state will be set to zero. A state should always have an if and else statement. For the if statement, a guard should be specified. Inside the if and else statement, any output and reg signals can be set. For multi-bit signals, individual or an interval of bits can be selected to be set. The next state is set by writing "next_state", this must be written inside both the if and else block.

```
<start_signal_declaration>
parameter number_of_states = <value>,
parameter counter_width = <value>
<signal type> <signal name> [<signal width>] [= <statement based on input signals if the type was guard>]
.
.
.
<end_signal_declaration>

beginstate <state name>: [<counter value>]
 if <guard name>
  <signal name> = <constant value that is the same width>
  .
  .
  .
  next_state = <name of next state>
 else

  <signal name> = <constant value that is the same width>
  .
  .
  .
  next_state = <name of next state>
endstate
.
.
.
```

Figure 4.21: Template of the language used to program the FSM.

4.7 PLL wrapper

In following section it is concluded that the PLL interface can be grouped into three categories: PLL configurations, PLL clocks and PLL control signals. This section will first discuss the PLL analysis to motivate a PLL wrapper, then present the PLL wrapper grouped by the three interface categories and design choices

made in them. The block diagram of the wrapper is seen in Figure 4.22. The chapter will then present an automation flow for integrating the PLL wrapper. All The configuration in this chapter is compile-time, with exception being run-time configurable APB vector ports. As will be read in this section, the interfaces were grouped into their sources rather than the alternative considered which was to group by functionality, more on this in section 5.2.4.

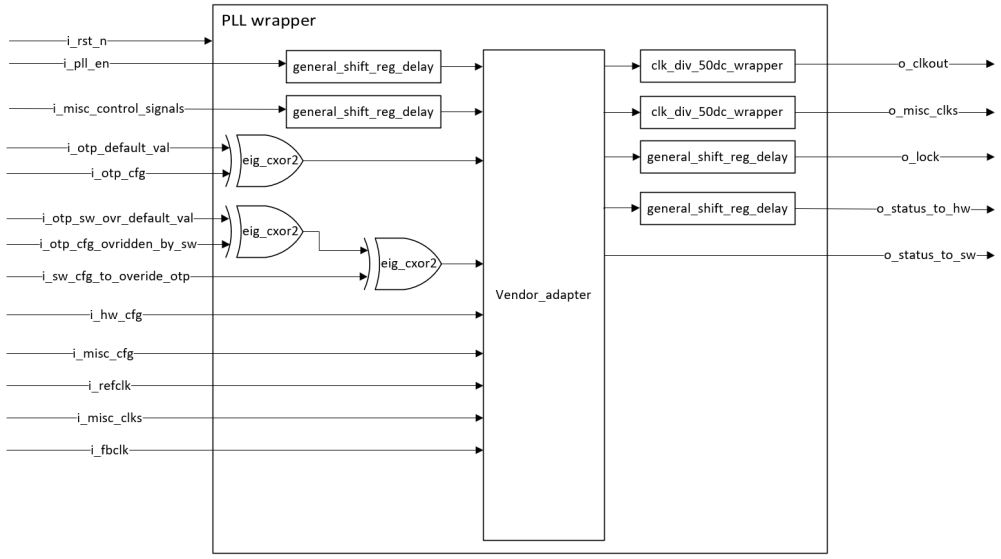

Figure 4.22: PLL wrapper design. Rectangles indicate a RTL module. i_rst_n is sent to general_shift_ref_delay and clk_div_50dc_wrapper.

4.7.1 PLL analysis

To construct a general PLL wrapper, analysis was performed on 4 different PLLs used at Ericsson. Additionally, since the PLL wrapper intends to map to all future PLL vendors, Ericsson's PLLs were compared to Alteras PLL design used in the Agilex 5 Field-programmable gate array (FPGA), [16]. It was seen that the PLLs share the same basic feature ports, which are one or more reference clock, PLL enable, PLL lock and one or more PLL output clocks. Some PLLs included a clock feedback port, used to balance clock skew. There were many signals that differed between the PLLs, which will be covered with the sections below.

4.7.2 PLL configurations

There were many features and signals that were unique to each PLL. Many of these can be seen as input configuration or output status, with input loop frequency divide being such an example. If all of these different signals would be included as ports, that are optionally used, the resulting wrapper would have numerous ports, making it cumbersome to use and hard to understand. Instead, the configurations were put in a vector, which is then uniquely set by each project, representing the specific PLL instance. This was also done to the many unique PLL outputs, bundling them into a status vector.

In the current projects at Ericsson, configurations were observed to come from different sources. Instead of grouping the configuration by functionality, they were categorized into what source it comes from. This predictability of the source of signals enables the wrapper to perform override logic on the sources, that is perform logical XOR between signals. The grouping of configuration signals is reasonable since the different configuration ports, except the hardware defined ones, are controlled by software, that can write to the specific fields of the vector, using the project specific vector representation. The identified configuration types from previous projects are, firstly, hardware defined configurations, meaning they are set at compile time. The second is hardware-defined values that can be overridden by OTP memory. The third is from APB registers, controlled at run-time by software. Further, a fourth type was added, not based on previous projects but to expand the wrapper functionality for the future, which is software APB registers that overrides OTP memory. The override logic was implemented by generating XOR blocks called eig_xor2. If the previous configuration source ports does not fit a configuration source, a fifth general source, called miscellaneous vector, was added. One such example of a source that did not fit into the other sources was a project using Serial Peripheral Interface (SPI) registers to set configurations.

The hardware defined values for hardware configurations and OTP default values were decided to come from input ports, instead of parameters. The resulting logic for the PLL wrapper will be the same, but the benefit being the wrapper is more flexible, since that port can also instead be driven from elsewhere.

4.7.3 PLL clocks

The wrapper were designed with vector ports for reference clocks and PLL generated output clocks. In addition to this it has miscellaneous clock vectors that are uniquely set for each project. This was done because the used clocks differs too much for a common interfaces to be built. The wrapper had an input miscellaneous for other clocks such as SCAN_CLOCK, DIFFERENTIAL_CLOCK and BYPASS_CLOCK, and an output miscellaneous vector for clocks such as DLL_OUTPUTS and DIFFERENTIAL_VCO_OUTPUT.

In an ASIC project, it was observed that the division options provided by the

PLL were not sufficient to produce the required frequency, which was solved by external clock division. The feature of external division was added as an feature in the wrapper to the output clock vector, with the divide values being set by vector parameters setting the divide value. The division was implemented by the clk_div_50dv_wrapper, see section 4.4 for details.

4.7.4 PLL control signals

The control signals presented in this section are those coming from and going to the CCR block. These were the PLL enable and PLL lock signals. In addition to this a miscellaneous control signal vector and a miscellaneous status to hardware vector was added.

In some previous projects, PLL enable signal had external shift registers delaying the assertion of the signal. This functionality was added to the wrapper, and expanded to the other PLL control signals mentioned previously. This delay enables flexibility in the design in for example delaying PLL enable if the delay in CCR_FSM is not sufficient. The delay on PLL lock can be useful since the clock divide mentioned previously takes some clock cycles to provide correct output.

4.7.5 PLL integration

The wrapper interface is connected to the PLL vendor pins through instantiating a vendor adapter file, seen in Figure 4.22. The vendor adapter file contains the pin mapping for all PLL instances in a project. Using a parameter, it selects between them with if statements inside generate block. This separation was done in order to keep the PLL wrapper file consistent between projects and have a project specific vendor_adapter file.

When instantiating the PLL wrapper, parameters are set for each PLL wrapper instance, using a package file for each PLL. This package then tailors the PLL wrapper to that instance, through delay values, divide values, override logic and which pin-mapping to use.

4.7.6 PLL instance table

The wrapper instantiation should be described in a specification. This is because different teams need information about the PLL, not only the design team. The verification team needs to connect and verify the wrapper and integration needs it to connect the wrapper. Figure 4.23, shows the proposed tabular structure for integrating the most recent PLL used at Ericsson. The ignore column tells the script to skip row if "Y" is present. The default value column creates parameters in the package file with those values, used in integration. Alternative considered was to have the configurations inside a word document, since this is where Ericsson

writes their design specification. Excel was chosen for easier implementation of software scripts.

Ignore	Child pin name	Parent connection	Default value
	PIN_1	i_hw_cfg[7:0]	b01111011
	PIN_2	i_hw_cfg[12:8]	b00100
	PIN_3	i_hw_cfg[21:13]	b000100000
	PIN_4	i_hw_cfg[45:22]	h000000
	PIN_5	i_hw_cfg[47:46]	b10
	PIN_6	i_hw_cfg[55:48]	b00000101
	PIN_7	i_hw_cfg[56]	b1
	PIN_8	i_pllen	
	PIN_9	i_pllen	
	PIN_10	i_pllen	
	PIN_11	i_pllen	
	PIN_12	o_lock	
	PIN_13	i_pllen	
	PIN_14	i_refclock[0]	
	PIN_15	o_clkout[0]	
	PIN_16	i_dft_cfg[0]	
	PIN_17	i_dft_cfg[1]	
	PIN_18	i_dft_cfg[2]	
	PIN_19	i_dft_cfg[3]	
	PIN_20	i_dft_cfg[4]	
	PIN_21	i_dft_cfg[5]	
	PIN_22	i_dft_cfg[6]	
	PIN_23	i_pllen	

Figure 4.23: Describing the connection between the PLL (child) and the port in the PLL wrapper (parent).

4.7.7 PLL automation script

To make the integration task easier and to modify the wrapper in one single source document, a Python script was developed to read the PLL instance table and generate one vendor_adapter file per project and one PLL_package per PLL, seen in Figure 4.24. In the vendor_adapter, the script creates the SystemVerilog module port definition based on the PLL wrapper module instantiation. The pin-mapping were inserted based on connections described in the PLL instance table. The names in the if and generate logic and PLL instantiation names were based on a config sheet which described the PLL module name and instance name.

The created package files contains parameter values for the width of the vector signals described in earlier sections, and default values for the hardware assigned values. When integrating the PLL wrapper the package file is used to assign parameter values when instantiating the wrapper.

The script was created to save time in integrating the PLL into the PLL wrapper, and minimize human made errors, and can be used in future CCR projects. The wrapper script was made general so that it can be used in various ASIC SystemVerilog projects, by using the excel sheet and the script.

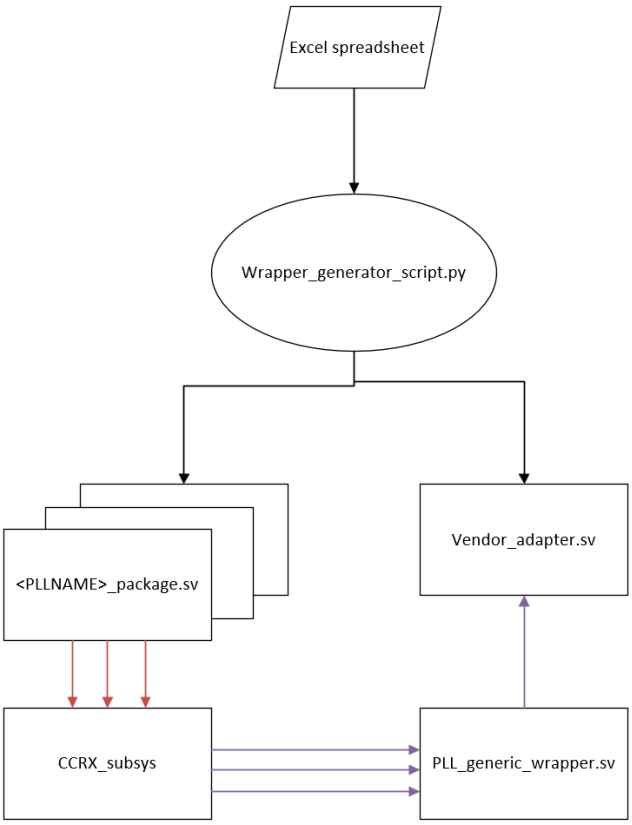

Figure 4.24: PLL automation flow. purple arrow: module instantiation, orange arrow: parameter extraction, black arrow: software data flow.

Chapter 5

Results and discussion

This chapter will contain power and area results and discussion for clock division, clock mux and reset FSM. It also provides analysis on the configurability for all of the developed IPs. It then has a discussion about another CCR architecture and a proposal for a future CCR module placement.

5.1 Area and power

The following section presents the power and area synthesis results. The method for extracting area and power is presented in 4.1.3.

5.1.1 Clock division

The area and power for this design can be viewed in table 5.1. The parameters that are not in the table have constant values for the two measured settings. The parameters that had constant values: $p\_nbr\_of\_input\_clocks = 1$, $p\_nbr\_of\_output\_clocks = 5$, $p\_nbr\_of\_div\_bits = 6$ and $p\_input\_clocks\_used = \{0, 0, 0, 0, 0\}$. The area and power is shown for two cases: one where the clock is only divided by a power of two, and one where none of the dividers are a power of two. As can be seen, when dividing only with numbers that are a power of two, the area and power is reduced quite significantly. This is because of the optimization that the SystemVerilog file will perform on the hardware in this case as explained in section 4.4.

Table 5.1: Clock divisions area and power results for two separate parameter configurations. Abbreviations used in table are **TO**: total area, **CO**: combinational area and **FF**: flip-flop area.

p_div_for_clock	TO (μm^2)	CO (μm^2)	FF (μm^2)	power(μW)
{2,4,8,16,32}	1.59	0.12	1.47	1.82
{3,5,7,15,31}	7.01	1.66	5.35	5.01

5.1.2 Clock observation

The synthesis result for the clock observation block can be seen in table 5.2. The parameters that are not in the table have constant values for all the measured settings. The constant parameters: $p\_nbr\_of\_input\_clocks = 16, p\_nbr\_of\_obs\_clocks = 1$, $p\_nbr\_of\_div\_options = 5$ and $p\_available\_divs = \{1, 2, 4, 8, 16\}$. These were kept constant as seeing how the other options change the area and power seemed like the interesting part. The biggest jump in area and power is when the clock observation is using glitch free transitions. The reason for this is that the multiplexers now use both more flip flops and logic. When changing from conventional two input clock muxes to the low latency one, the area and power decreases. This is because it uses fewer flip flops than the conventional method. When the clk_gone_counter is used, the area and power obviously increases as it will add more logic and flip flops to the design. If only the high or low counter would be used (instead of both), the increase would be less.

Table 5.2: Clock mux area and power for parameter configurations. **GF:** p_use_glitch_free, **LL:** p_use_LL_mux, **LC:** p_low_counter_enabled, **HC:** p_high_counter_enabled, **MC:** p_max_count, **TO**: total area, **CO**: combinational area, **FF**: flip-flop area.

GF	LL	LC	HC	MC	TO (μm^2)	CO (μm^2)	FF (μm^2)	power(μW)
0	0	0	0	0	2.22	0.97	1.25	1.29
1	0	0	0	0	19.69	2.27	17.31	11.1
1	1	0	0	0	14.2	2.85	11.24	8.80
1	1	1	1	32	19.53	4.4	14.81	15.6

5.1.3 Runtime FSM

The synthesis results below were extracted for the Multi output FSM with counter.

Area

In order to do an accurate analysis of this block, synthesis was performed on it for different parameter values. The parameters: p_nbr_of_states, p_nbr_of_guards, p_nbr_of_outputs and p_counter_width. Points with the values: 5,10,20,40,80 for each parameter were part of the data set. Using the least square method, the four-dimensional polynomial in equation 5.1 was found. The variable x is equal to p_nbr_of_states, y to p_nbr_of_guards, z to p_nbr_of_outputs, and w to p_counter_width. The reason this was tried was to see which parameters that has the most impact on the area. However, it was very difficult to see this from the polynomial. The correlation between each parameter and the area was checked to get a number that can be compared between each of the parameters. As can be seen in table 5.3 , p_nbr_of_states had the highest correlation with the area, followed by p_nbr_of_outputs, p_counter_width and lastly p_nbr_of_guards. Assuming a linear relationship, this means that the number of states has the most impact, which makes sense as it will increase the amount of rows for each input column. Something that was also done was to go between 5-80 and increment with 5 for all the parameters at the same, this was done in order to get the graph that can be seen in Figure 5.1. The graph was done in order to give a visual representation of how the area grows when the parameter values are increased, as it is a four dimensional function, it needed to be limited to one dimension to make it easy to plot.

$$\begin{aligned} area = &- 0.665 + 0.234w + 0.234z + 0.021zw + 0.233y + 0.016yw + 0.018yz \\ &- 0.016yzw + 0.234x + 0.023xw + 0.025xz + 0.018xzw + 0.020xy \\ &+ 2.34 \times 10^{-6} xyzw \end{aligned} \tag{5.1}$$

Table 5.3: Showing the correlation of module parameters and resulting area.

Parameter	Correlation
p_number_of_states	0.711
p_number_of_guards	0.632
p_number_of_outputs	0.686
p_counter_width	0.668

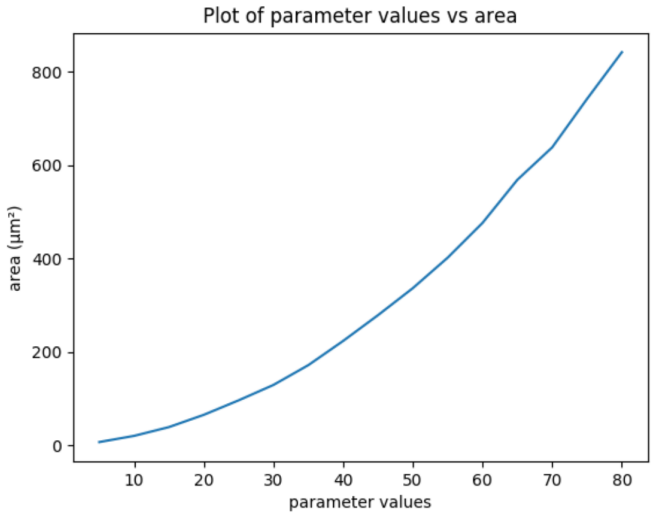

Figure 5.1: Showing the area as a function of FSM parameter values, which are all set to the same value.

Power

The power data was collected at the same time as the area and thus the parameter values are the same, see more in section the area section above. Once again, a polynomial was found in equation 5.2 that describes the power as a function of the parameters. In table 5.4, the correlation between the parameters and the power is shown. Unlike for the area, the number of states and the number of outputs seems to have the same level of impact on the power. Otherwise, the order is the same as for the area. Why it is this way is hard to say, can only speculate that maybe the type of cells that are used when it increases uses more power in relation to how much the area increases. In Figure 5.2, the power is shown for different parameter values in the same way as for the area. As can be seen, the values for 65-70 are almost the same and look very out of place compared to the rest of the points. This is most likely due to some bug in the synthesis tool used, what exactly was not found. It shows a very similar relationship between the parameter value and the power as between the parameters and the area. This is not surprising, as when the area increases the number of flip flops and logic cells increase and therefore more power will be needed to run all the circuitry.

$$\begin{aligned} power =& 2.988 + 0.051w + 0.052z + 0.041zw + 0.051y + 0.034yw \\ &+ 0.039yz - 0.016yzw + 0.052x + 0.041xw + 0.046xz \\ &+ 0.019xzw + 0.039xy - 0.016xyw + 0.009xyz + 3.600 \times 10^{-5}xyzw \end{aligned} \tag{5.2}$$

Table 5.4: Showing the correlation of module parameters and resulting power consumption.

Parameter	Correlation
p_number_of_states	0.706
p_number_of_guards	0.627
p_number_of_outputs	0.706
p_counter_width	0.650

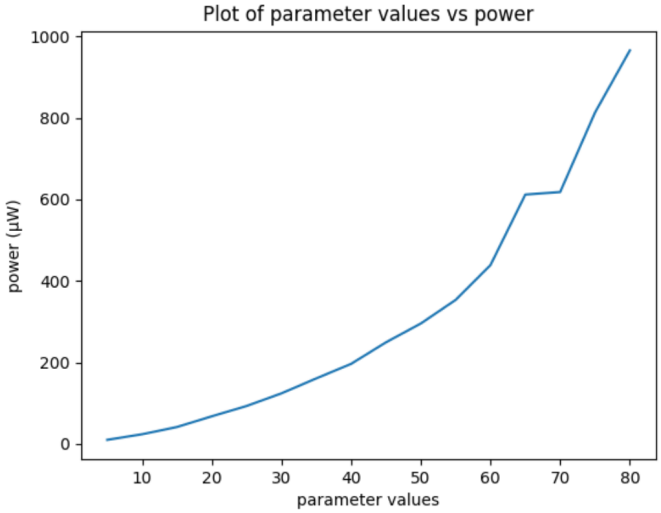

Figure 5.2: Showing the total power consumption as a function of FSM parameter values, which are set to the same value.

Handwritten FSM vs run-time

Another analysis was performed. In table 5.5 below, the area and power for the three different cases are presented. The cases were between a handwritten reset FSM from an old CCR, a run-time FSM that has the capacity to be configured for the handwritten one, and a run-time FSM with constant values for the configuration inputs (it can't be configured during run-time any more). One thing to point out is that the comparison is not completely one to one. The handwritten one contains some functionality (such as synchronizers) that will add some area and power. However, these parts are not that big in comparison to what is the same for all three versions. The run-time FSM can also not implement every state exactly like the handwritten version and thus those states were split into multiple ones.

As can be seen, the run-time FSM significantly increases the area and power compared to the handwritten one. This is not surprising, as the run-time FSM

must be able to handle a much larger possible number of cases, and thus requires more area and power. The RTP (run-time FSM with compile time parameters) unsurprisingly uses less area and power than the run-time FSM as the synthesis tool is able to use constant propagation to optimize the result. However, what was surprising was that the RTP uses less area and power than the handwritten FSM. The reason for this is partially because the RTP does not have all the functionality that exist in the handwritten one. However, it does not completely explain the discrepancy. One big reason that seems to result in less area and power is that the RTP reuses the same counter with different values taken from a look up table, unlike the handwritten one that has individual counters for separate cases. This can be seen by the fact that both RT and RTP uses a lot less flip flops (but RT uses a lot more combinational logic because of all the multiplexers). The use of specific counters in the handwritten FSM makes it more readable which helps when debugging. Surprisingly, this led to the conclusion that the language for the run-time FSM can be used to get readable code (text based code instead of bits for a table) while getting more optimized implementations, at least when it comes to reusing counters.

Table 5.5: Showing the comparison between: HW: handwritten FSM, RT: run-time FSM, RTP: run-time FSM with compile-time parameters.

	HW	RT	RTP
Total area (μm^2)	38.46	113.59	15.22
Combinational area (μm^2)	7.80	102.77	4.95
Flip-flop area (μm^2)	29.60	10.51	9.95
power (μW)	56.07	125.7	25.72

5.2 Configurability analysis and discussion

This section presents analysis and discussions on the configurability of all implementations in relation to the requirement on less development time found in section 3.4.1. It also presents extensions to the implementations or alternative design implementations.

5.2.1 Simple CCR configurable modules

Reset request handler

Since the reset request handler heavily depend on the rest of the CCR architecture, It was difficult find what features could be made configurable and if it would suffice for feature use. Configurability of a design can be done if clear patterns exist. If the design often differs from these patterns, it is hard to modify the configurable

code to include the new logic. The proposed implementation focused on how to maximize configurability based on previous designs. The proposed separation into a core configurable module with project specific wrapper, created a structure that is good for configuration and modification similar to previous designs. However, the separation made the reset request handler more unclear in terms of what is done in the core, and what is done in the wrapper, with a mix of features in both. The configurability also made the code significantly harder to read, in the form of more general names and signals being merged into general vectors.

The resulting module shows a trade-off between configuring a design for different features and how easy to add new logic that is outside of the possibilities of the current configurability. The time spent on making the design configurable to this degree is deemed not worth, considering the small gain in terms of configurability and the drawbacks listed above.

Future recommendation is not to use the proposed wrapper solution, and instead have one file containing all logic. What could be kept as configurable is; parameters for vector width, The run-time configurable reset gating to make the module more flexible at run-time, variable amount of reset sources and variable amount of external reset signals. Architecture specific things like the different reset commands, might be better to directly redesign between projects, instead of making changes in the proposed wrapper. Configurability should be added where it is easy to add or necessary for the design, which will lead to a mix of configurability and specific RTL code.

Reset glitch filter

The reset glitch filter is a very small, but vital block for the CCR. It can handle multiple resets, but there is only one parameter for the number of delay blocks. It would be conceivable to add the feature that each reset delay can be set individually. However, this seems like an unnecessary feature as all resets would most likely need the same amount of delay. Furthermore, it would make it more complicated to set and thus more tedious to integrate. Another thing that could be added to the filter is a run-time configurable delay. This could be used if it turned out that the delay chosen was unnecessarily long and thus leads to a slower startup. However, boot-up occurs very rarely and thus it would probably not be worth the extra complexity. As a result, the conclusion can be drawn that the current implementation is probably good enough. The current implementation could potentially save a small amount of development time by not having to place all filters for all resets by hand. However, it would most likely not save a lot of time, but might as well be used now that it is developed.

5.2.2 Clock division

The clock division block can be used to generate slow clocks by using clock division. Unlike a lot of blocks written in SystemVerilog code, it does not have a very set structure. The reason for this is because the block is very configurable. A potential problem with a configurable design is that depending on how the code is written, overhead could cause the design to be less optimal than one that is written by hand. The reason that one written by hand could be better is because an engineer could optimize it for the specific case that the hardware being developed for.

However, as this block is configurable during compile time, it can be optimized as described in section 4.4. This principle could be applied in general to highly configurable blocks. The way this is done is by having parameters that the engineer sets, then code that is able to analyse the set parameters and based on that create a design that is more similar to a hand written one. In SystemVerilog code, it can be implemented by using functions that set local parameters that are then used in a generate block to generate more optimal hardware. One drawback of this approach is that the code becomes harder to understand, another is that SystemVerilog is not necessarily the easiest language to implement complex analysis of parameters. Another approach is to use a programming language (e.g Python) to generate a SystemVerilog file based on parameters. This would solve the issue of readability and make it easier to write code that performs optimizations. One negative aspect would be that an extra step in using this block would be necessary as the script would need to be run first to generate the code. In that sense, having one SystemVerilog file is easier to use. While simply running a script is not very difficult, it needs to be clear where it is and how to use it.

The question becomes, how useful is this block? On its own, it probably does not save that much development time as this is quite a small but important feature of the CCR. However, now that it exists it might as well be used as it would make the job of implementing the generation of slow clocks faster. It might also reduce the risk of bugs and make it easier to create reusable Universal Verification Methodology (UVM) components as well as test cases. These could be configurable with the same parameters that are used for the design. What the development of this block shows is that making a configurable design will take more time (especially with optimization), but can be useful for something that is used often. As clock division is only done twice (generating slow clocks and clock observation) per chip, it might not reduce development time much as it is a very small percentage of the work. If it is a feature that will be used a lot with different attributes, it might make more sense to put in the time to create one configurable block with optimizations. How much the configurable block would help with verification is hard to say as it would need to be tested to be conclusive.

5.2.3 Clock observation

Clock observation is a part of this thesis that became bigger then it was first thought to be. The reason for this is that options for the clock mux was explored. The clock mux part of section 4.5 is thus more exploratory than other sections. It might not necessarily be needed for most projects (as it has not been needed before), but if that changes the options are already there. The work on the clock mux could also potentially be used for other parts of the chip if needed. The one-hot encoded clock mux is an idea that could be tested more thoroughly to see how high clock frequencies it can be used for. One potential benefit of using the more conventional two input clock muxes is that they can use special cells that are made specifically for that circuit. This might lead to a more clean clock signal than using the one-hot mux that does not have a special made cell. A potential fix to this problem is to create one big cell for the one-hot mux that can take in some max number of input clocks. However, it would most likely take more time to develop one large cell in comparison to one small cell that is used multiple times. This would also mean that developing such a cell would probably be more costly. However, this is all assuming that such a cell would be required to get a good enough signal for clock observation. This would have to be determined in further studies.

The clk_gone_counter would be useful in cases where a clock that can be observed may disappear. Thus for projects where glitch free clock muxing is used, it will probably be needed as a precaution. One thing that could be changed in the future is that the counter could use more than one clock to count and check if the selected clock is stuck. A more robust idea would be to have one clk_gone_counter for each input clock and if any of them detects that the selected clock is gone then it performs the sequence of events specified in section 4.5.6. One potential problem with this is that for the counters to detect if the clock is gone at about the same time, the engineer must set the counter value for each clock frequency. Another approach is to have the one counter value for all counters, but set it based on the fastest clock. This would ensure that the death of the clock is not detected prematurely. However, it would mean that the slower clocks would take longer to detect the loss of the clock, but this would only have an effect if the fastest clock disappeared.

The multi bit CDC block is very simple, but could be used in similar situations where a register can change asynchronous to the running clock. It would work best for situations where said register does not change very often as it takes 2-3 clock cycles for the value to change. For situations where a multi bit signal is changing values very frequently, an asynchronous First In, First Out (FIFO) should probably be used instead.

Similarly to clock division, the question becomes: how useful is this block? If the new features are used (multiple observation clocks, glitch free transitions, low latency clock mux, and handling of when the selected clock dies), then the work that has been done will have been useful. If they are not used, it will still be simpler to use this block as just a few parameters need to be changed and then

it is ready to be used. Similarly to clock division block, how much this will save development time is questionable. For verification, the same thing can be done for this block as for clock division: create general UVM components and test cases that use the same parameters as the block to configure.

5.2.4 PLL wrapper

This section will discuss the PLL wrapper implementation and the proposed PLL automation flow, in terms of quality, reusability and ease of use. It will also propose an extension to the PLL automation flow.

PLL wrapper analysis

The ports in the PLL wrapper was chosen to be flexible for future use. When considering the ports, the sources of the signals were largely influencing. Example of this is that the configurations were put into vectors, based on category, making the interface more readable and easier to connect, and can makes for easier reusability in the UVM framework, since the ports remain the same between projects.

The challenge with vectors is that the content and order of the vectors might change between projects, which can break reusability in verification and integration. A solution for the clock vectors is to have the same bit representation of the vector with the different types of clocks set at a specific index. For example, the signal SCAN_CLOCK can always be the index 0 of input miscellaneous clocks and OFFSETCALCLK always on index 1, and if only OFFSETFCALCLK is used, the size of the vector is still 2 and index 0 is left unused. The configuration ports are harder to keep consistent like the above proposed solution, due to variance in width of the configuration signals between the PLLs. This might not be a big problem since the configurations mainly comes from software controlled registers, the vector representation exists in software, where it is easier to maintain.

There were considerations to group the ports into the type of interface it belonged to. An example of this is the scan interface which is included in some PLLs for Design for testing (DFT). The benefit is that it could be easier to integrate the wrapper. The downside is that it would be hard to combine with the solution for override logic, and doing so could lead to an increase in ports and reduced readability and possibly flexibility, and was thus not chosen.

According to experts at Ericsson a good wrapper can transform a task requiring a designer into one that only requires integration and little knowledge about details of the module. The wrapper designed in this thesis hides details about override logic, delay logic and divide logic, into the wrapper which is then controlled with parameters, making the job of controlling this easy for integration. This logic implemented in the wrapper, while not being that complex, might save from errors that can originate from copying and pasting code, which is currently done. Having

one wrapper interface has the benefit of making the verification more reusable, since the framework needs to test one module instead of several ones.

The drawback of the PLL wrapper is that it adds one additional step in implementing a PLL. Instead of directly connecting the signals to the PLL and quickly make logic around it, the designer now has to put all signals into arrays and set the values of the parameters. This process can be error prone and add extra design time. To accommodate for this drawback, the PLL wrapper automation flow was made.

PLL wrapper automation flow

The automation flow does not currently generate parameter values for delays or divide logic, and currently has to be manually inserted into the packages. The PLL instance table can be extended to include this information and then the script can be extended to generate these parameters.

The PLL wrapper automation flow was designed to be a general design pattern, which can be applied to other RTL projects which has a similar problem of integrating various IPs with a unified wrapper. When other projects use the wrapper flow then ideally only the wrapper module needs to be changed in order to fit that specific project. The implemented automation flow makes the job of integrating the PLL wrapper together with the PLL easier. The integrator does not need to set parameter values and make connections in RTL, and instead only needs to modify connections and default values in an excel table, reducing the complexity of integrating the PLL. For an experienced RTL designer the time saved can be negligible, due to the simplicity of the logic inside the wrapper. The proposed extensions below can however lead to saved time.

The automation flow can be expanded to generate connectivity assertions in a svh file and also make modifications into the UVM framework, based on the integration excel table, see Figure 5.3. The main thing to consider in the UVM framework are modifications in testcases and functional coverage, which depends on the connections to the PLL. The automation flow can also be expanded to automatically create Ericsson register tables, which are tables used for automatically generate APB register files. These modification makes it easier to integrate the wrapper, reduces the modifications needed in verification and reduces errors in the design process. Expanding the automation flow to this could potentially create a structured way of seamlessly creating wrapper to a design. The designer needs only to create a task specific SystemVerilog wrapper, with the integration being handled by the automation flow running on the excel file.

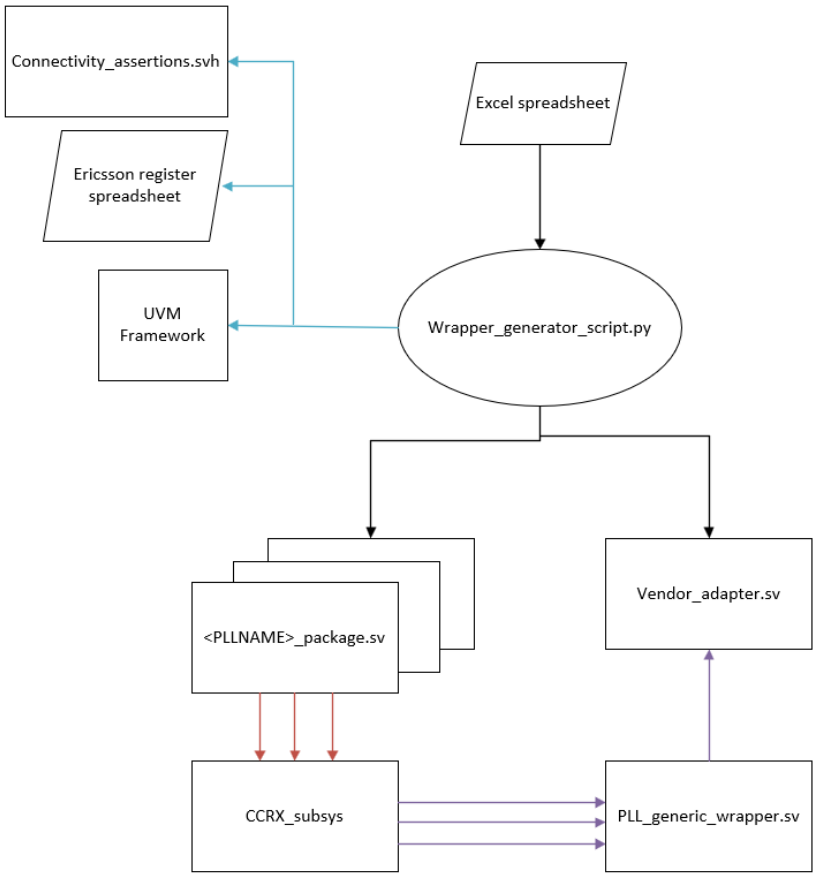

Figure 5.3: Proposed extension, in blue arrows, to PLL automation flow.

5.2.5 Run-time configurable FSM

Table based approach

The table based approach presented in section 4.6.1 has some limitations that could be a problem for some applications. The most obvious one is that it can only evaluate one guard per state. This can be a problem if there is need for a state with more complex evaluation. This could for example be in the form that if the first guard is true, the machine should evaluate another guard and based on that guard do different things. For the current implementation, the state would have to be split into two separate states and thus be evaluated during two different clock cycles. For a lot of implementations performing the logic in one or two clock cycles might not matter, whilst for others it might make all the difference. Adding the capability to do it in one clock cycle would require adding more columns to the table and making the design more complex. In fact, adding more features to this design would in general not be very straightforward (compared to other ideas that will be discussed later), especially for someone that did not design the existing design. Another issue is a fundamental problem with the programming format being a table: any new feature will add new columns that will result in more registers as it needs to be able to be programmed for each state. It thus becomes questionable whether it is worth it to add more features, especially if only a few states will use it as every state will still add registers. One way to fix this problem is to connect only some of the configuration inputs for some of the states to registers and connect others to constant values. Another approach to remedy these problems is to make another architecture, as will be discussed below.

Instruction based FSM

The reason this idea is presented here is because it was not implemented and tested in code and thus falls under future work. The approach presented here is to make a design that have a set amount of instructions per state that says what should be done in that state. There are probably a lot of ways to implement such a design, each with their own strengths and weaknesses. The benefits of this kind of architecture is that adding more features could be made quite simple by implementing new instructions. It could also make it easier to implement more complicated guard logic as the instructions should effect each other in such a way that some instructions should decide if other instructions are valid.

One way to implement this would be to have a block that can execute one instruction and then connect multiple of these blocks in a line as can be seen in Figure 5.4. One of these blocks would set an output vector, next state and a control vector. The output vector would be sent to the next block which would then do the same thing. If multiple blocks want to set the same bits then the one latest on the line would ultimately set the final output. A block that is earlier would also send control signals to all the blocks after it on the line. This could

be used to implement if statements by having that a block evaluates a statement and then based on that disable some of the blocks that come after. A disabled block would let the output vector pass through without change. This design also has some negative aspects. Firstly, the critical path would increase linearly as the amount of instructions per state increased, this can be seen by the fact that the output vector goes into and out of every block. This results in the critical path not scaling very well and thus perhaps not being that useful for complicated state machines that might need a lot of instructions per state to work. Another architecture could probably be developed to fix the critical path issue. But that lies outside the scope of this thesis. The second negative aspect is that it could be a bit complicated to program as the programmer would basically have to learn a new assembly language.

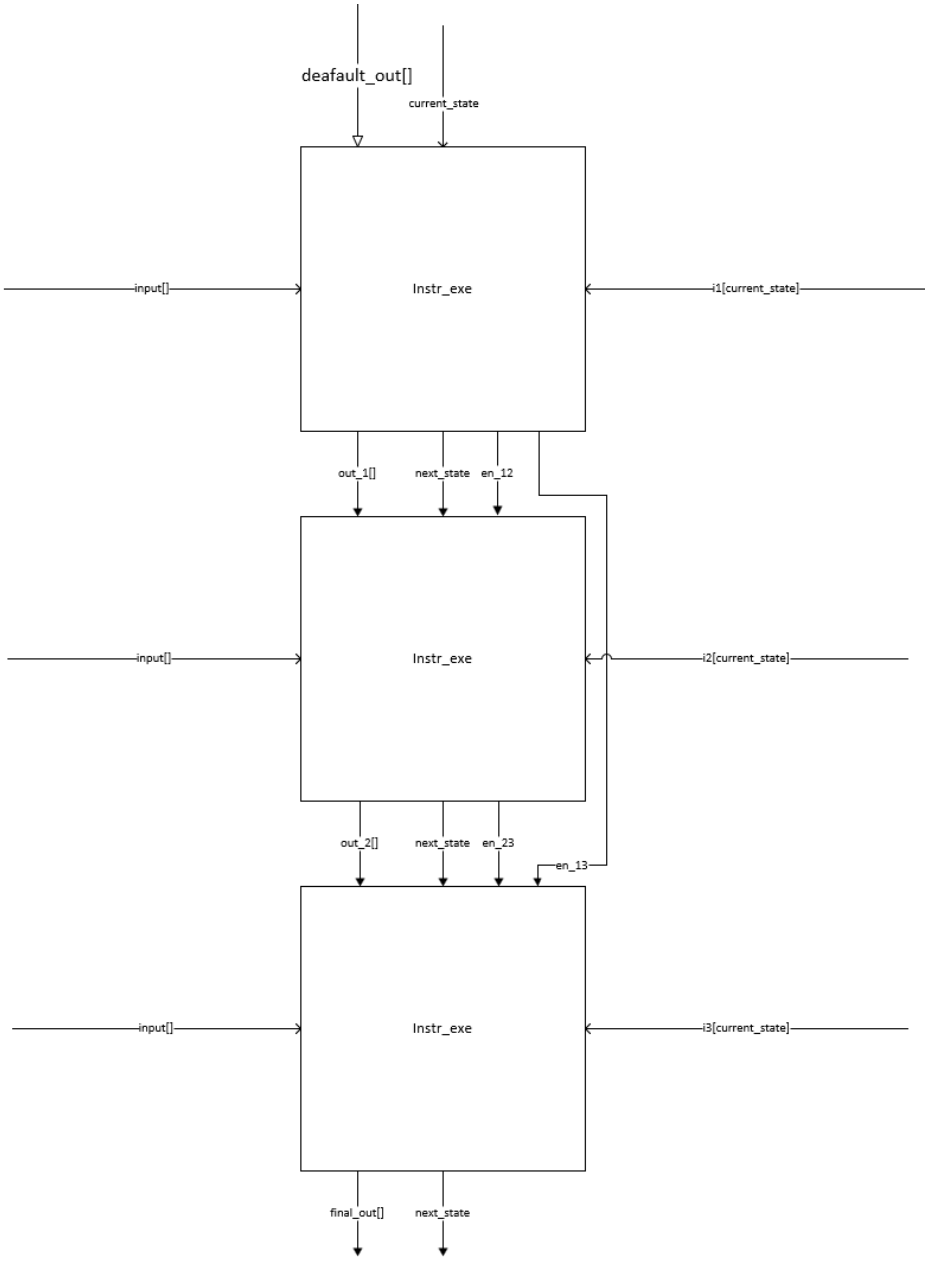

Figure 5.4: One design for instruction based FSM where the blocks executing the instructions are connected on a line, this example uses 3 instructions per state.

RISC-V processor approach

This approach would use an RISC-V processor to perform the logic of the FSM. This implementation would include a processor and then registers that the processor could write to that set the output signals for the FSM. As high throughput and clock frequency would most likely not be needed, the processor would not have to be pipelined. The calculations that the processor would need to perform is not particularly complex and thus only the base integer instruction set would be needed. These two things would probably cause the processor to not use a lot of area. One potential problem with this approach is that it could be very slow to set multiple bits in a state if they all have unique addresses. One solution to this would be to place bits together. This could however be a problem if only some of the bits at an address need to be set. A solution to this could be to split the signal that the processor is writing into two fields, a mask and the data. For example: if the processor is writing 32 bits then 16 of the bits would be the mask and the other 16 would be the bit values, this means that at each address there would be 16 bits stored. This implementation would enable the processor to have the capability to set individual bits as well as set multiple bits at the same time. This means that the placement of output bits could have an impact on how many cycles it takes to implement one state. The inputs to the FSM would also be placed in the address map, but read only.

One interesting aspect of this solution is that a higher level programming language (like C) could be used to program the FSM, as existing compilers could be used. This is in contrast to the previously mentioned ideas where programming the FSM could be quite tedious. However, one potential problem with this idea is that it becomes less precise. With the table based approach one row will be evaluated during a clock cycle, whereas with the processor approach it is not necessarily obvious how many clock cycles any one thing will take when written in a higher level language. One way to remedy this issue would be to write the code in assembly, but that would take away the main benefit of using this idea. One other benefit of using this approach is that it is a tried and tested instruction set that thus is known to work. This would be in contrast to the idea from the last section that would need a brand new instruction set to fit the architecture and thus require a lot more testing.

Why the table based approach was prioritized

There were a few different reasons why the table based approach was prioritized over the other two other ideas. The first and main reason is that it is a simpler design and thus easier to fit into the thesis as a lot of other blocks took some time to make. Its simplicity also makes it less likely to cause bugs during development. Another reason is that the table structure is very intuitive and thus perhaps easier to program than an assembly language. The time saved by making a simpler design also made sure that there could be enough time to work on a more user friendly interface in the form of a text based one as described in section 4.6.2.

Discussion about the FSM compiler and language

The language described in section 4.6.2 is a very limited one that needs to be written in a very specific way. The positive thing about this is that the compiler becomes easier to make. However, it can become quite tedious to program certain things that is easily done in SystemVerilog code. A simple example is when setting an output bit equal to an input bit. A whole state is needed to perform this action. The state needs to have a guard equal to the input bit, when the guard is evaluated it will set the output equal to one if it was true, and zero if it was false. In SystemVerilog code it could have been done with one line. The reason for this is mostly because of how the run-time FSM was implemented, but perhaps a special case could have been done in the language so that a new state would be inserted if an output was set equal to the input. Another limitation with this language is that it can only use one dimensional signals. Multidimensional signals is something that could be added if there is any need for it. The most ideal solution would probably be to build a compiler that can turn SystemVerilog code into this table representation, but that would take a significant more amount of work and time. It would probably take more work than it would be worth. However, as the language stand right now, it is a big improvement over trying to configure the FSM directly bit by bit.

5.2.6 Compiler directives vs module parameters

This thesis decided to mainly use module parameters instead of compiler directives. When a module can be instantiated multiple times, it is necessary to use module parameters to be able to have different configurations. If the module is only instantiated once, compiler directives can be used as effectively. The advantages of using them are that ports can easily be eliminated from the design which can reduce the amount of synthesis warnings, and that it hides non driven ports in simulations. The drawbacks are that some designers find the code to be harder to read and the compiler directives more clunky to use. This comes down to personal preference, and both can be used to achieve good configurability. The thesis seldom mixed the two techniques, in order to keep consistency.

5.3 Other CCR architectures at Ericsson

There exists other CCRs architectures at Ericsson, but these were not in the scope of the thesis to implement. Below will briefly analyse one such CCR architecture used in another ASIC project, and discuss why it was not analysed further in the thesis.

The analyses of the other CCR architecture revealed big differences between that CCR architecture and the CCR analysed in this thesis. That CCR IP is provided by a third party vendor. That IP could not be analysed, but Ericsson had im-

plemented a Always-On (AON) IP which is responsible for reset sequencing and driving the vendor CCR. This way of using the CCR was totally different from this thesis, but shows that CCR can be made configurable enough to make it a third party IP, with additional implementation needed on top of it.

The other CCR was developed independently from the ASIC projects analysed in this thesis. This makes the general architecture dissimilar, with differences like communication protocols used and the interaction with other IPs. The clock and reset specification probably also differ, making the CCR specification different.

Making a unified CCR between these projects would require extensive work. Firstly one would have to analyse the different clock and reset specifications and what needs to be provided by the configurable CCR. One way to unify the designs is to design a general CCR module and do a specific, project based module interacting with the general CCR. Similarities between these projects clocks and reset specification are probably PLL initialisation, reset request handling and clock observation. Differences would be the reset sequences and overall ASIC architecture.

Chapter 6

Conclusions and future work

This chapter presents conclusions about the thesis, which are our collected thoughts about the project, and recommendations for other designers tackling similar tasks. The future work section is a summarized bullet list over future work, with further explanations found in chapter 5.

6.1 Conclusions

In conclusion, designing heavily configurable code, one needs to chose between using code generation or, using SystemVerilog language features or similar languages. When using SystemVerilog, the readability of the code might be reduced. The code is however updated and written in one file with known syntax, which gives designers a familiar design flow. Code generation is a very powerful alternative, but has some drawbacks. It can come with higher development time then writing the software scripts. It is hard to mix code generation with normal SystemVerilog, and once code generation is chosen, all the designers need to adhere to it when modifying the design. After doing this thesis and exploring both code generation and SystemVerilog features, we would advice higher usage of code generation, when targeting tasks requiring heavy configurability. This is since it offers much higher possibilities of configurability, so the designer does not have to think first "can I do this configurability in SystemVerilog?". Code generation also generates more readable code, which is important when working in larger groups and when debugging.

Making an architecture configurable might lead to changes being needed. A good approach when designing something configurable is to have a parametrised general core that is applied in a specific way. This can ensure the the general core avoids re-design, thus becoming a reliable tried and true solution. The general core can then be driven in a specific way, by using a project specific module. This can reduce the time and complexity of design and verification. The difficulties lies in finding what should be part of the general core and what should not, which was seen in

the thesis when designing the reset request handler. Instead of the thesis approach of making the modules inside the CCR configurable, perhaps the approach is to make an overhaul and redesign of the CCR architecture, focusing on what should be divided into the general part and the specific one. However, this needs a lot of architectural knowledge of current constraints and future requirements, to truly make a good solution.

One conclusion that can be drawn from the clock divider example (see section 4.4 and 5.2.2) is that with more work, a configurable design can be more optimized than maybe first assumed. When making a design, a handwritten one will tend to be more optimized than one set by parameters. However, by writing code that can analyse chosen parameters and optimize based on that, the design can be more in line with a handwritten one. This approach is one that can be used for future designs, especially for blocks that are more widely used and has more of an impact on overall power, area, etc, of the chip.

From the clock observation block, the options for the clock mux could be used in the future if glitch free clock transitions are needed. In particular, based on all metrics that have been observed in this thesis (area, power, transition speed, critical path), the one-hot clock mux seems to be better for cases where more than 2 clock can be multiplexed between. One potential problem with this new approach is that it need more testing and potentially more time to develop if a cell is needed leading to higher costs.

Designing a wrapper module usually has low technical difficulty, but is challenging in understanding the flow of integration of the module. When choosing the ports of the wrapper it is hard to conclude "what is best", but the wrapper still has to be good enough to prefer using it over not using it. The automation flow presented is an attempt on making the wrapper easier to use. This workflow or a similar workflow for integration of third-party IPs through a wrapper, could have potential to save time in design, verification and integration, especially if it is broadly adopted across many teams.

The run-time FSM could potentially be used for state machines that are very important for the functionality of an ASIC (such as the CCR). For such cases, the increase in area and power might be an acceptable trade-off. The architecture explored in this thesis has many limitations when it comes to what state machines it can implement. Therefore, if this is interesting to explore further, then other architectures should be considered.

6.2 Future work

- Design a configurable reset sequencer.
- Investigate other run-time configurable FSM architectures, or explore ways to expand current architecture and compiler.
- Expand PLL wrapper automation flow to generate: all parameters for PLL

wrapper, connectivity assertions, register file spreadsheet and modification of verification framework.

- Create configurable UVM components and testcases.
- Create a scheme that automatically resolves CDC, by inserting synchronizers.
- Test physical properties of the new clock mux design.
- Create a configurable CCR top module.

References

[1] F. Rincon, F. Moya, J. Barba, and J. C. Lopez, *Model reuse through hardware design patterns*, 2007. arXiv: 0710.4755 [cs.SE]. [Online]. Available: https://arxiv.org/abs/0710.4755.

[2] K. Shu and E. Sánchez-Sinencio, *CMOS PLL Synthesizers: Analysis and Design* (The Springer International Series in Engineering and Computer Science), 1st ed. Springer New York, 2005, pp. XVI+216, Hardcover: 2004-11-12; eBook: 2006-01-20; Softcover: 2010-12-08, ISBN: 978-0-387-23668-1. DOI: 10.1007/b102174.

[3] P. Analog. "Understanding the basics of PLL frequency synthesis." Accessed: Jan. 26, 2026. [Online]. Available: https://www.edn.com/understanding-the-basics-of-pll-frequency-synthesis.

[4] AMD, *Ultrascale architecture clocking resources user guide (ug572)*, Accessed: Apr. 8, 2026, May 2025. [Online]. Available: https://docs.amd.com/r/en-US/ug572-ultrascale-clocking/PLL-Ports.

[5] Intel, *Altpll (phase-locked loop) ip core user guide*, 683732, Accessed: Apr. 8, 2026, Jun. 2017. [Online]. Available: https://docs.altera.com/r/docs/683732/17.0/altpll-phase-locked-loop-ip-core-user-guide/ports-and-parameters.

[6] M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*. Dordrecht: Springer, 2011.

[7] S. Zeidler, O. Schrape, A. Breitenreiter, and M. Krstić, "A glitch-free clock multiplexer for non-continuously running clocks," Accessed: Feb. 2, 2026, IHP – Leibniz Institut für Innovative Mikroelektronik, University of Potsdam. [Online]. Available: https://conferences.computer.org/dsd/pdfs/DSD2020-3gccs2DX4TPUFhzwStqjoz/953500a011/953500a011.pdf.

[8] M. U. Merino and D. Juneja, "Powering ics on and off," *Analog Dialogue*, vol. 49, no. 1, p. 11, 2015, Editor's Notes section, Analog Devices, Analog Dialogue Volume 49, Number 1. [Online]. Available: https://www.analog.com/media/en/analog-dialogue/volume-49/number-1/articles/powering-ics-on-and-off.pdf.

[9] Intel, *Arria 10 hard processor system technical reference manual*, 683711, Accessed: Feb. 2, 2026, Jan. 2026. [Online]. Available: https://docs.altera.com/r/docs/683711/25.3.1/arria-10-hard-processor-system-technical-reference-manual/arria-10-hard-processor-system-technical-reference-manual-revision-history.

[10] E. Billauer. "The logic for starting off and resetting an FPGA properly." Accessed: Jan. 27, 2026. [Online]. Available: https://www.01signal.com/verilog-design/reset/powerup-state-machine/.

[11] "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, pp. 1–1354, 2024. DOI: 10.1109/IEEESTD.2024.10458102.

[12] V. Dhare and D. Sonar, "Verilog code generation from script for generic digital designs," in *2025 5th Asian Conference on Innovation in Technology (ASIANCON)*, 2025, pp. 1–4. DOI: 10.1109/ASIANCON66527.2025.11280825.

[13] R. S. Ferreira, J. Nolte, F. Vargas, N. George, and M. Hübner, "Runtime hardware reconfiguration of functional units to support mixed-critical applications," in *2020 IEEE Latin-American Test Symposium (LATS)*, 2020, pp. 1–6. DOI: 10.1109/LATS49555.2020.9093692.

[14] R. Damaševičius, G. Majauskas, and V. Štuikys, "Application of design patterns for hardware design," in *Proceedings of the 40th Annual Design Automation Conference*, ser. DAC '03, Anaheim, CA, USA: Association for Computing Machinery, 2003, pp. 48–53, ISBN: 1581136889. DOI: 10.1145/775832.775847. [Online]. Available: https://doi.org/10.1145/775832.775847.

[15] S. Verma and A. S. Dabare. "Understanding clock domain crossing issues." Accessed: Mar. 13, 2026. [Online]. Available: https://www.eetimes.com/understanding-clock-domain-crossing-issues/.

[16] Intel, *Clocking and pll user guide agilex™ 5 fpgas and socs*, Accessed: Apr. 8, 2026, Mar. 2026. [Online]. Available: https://docs.altera.com/r/docs/813671/25.1.1/clocking-and-pll-user-guide-

agilextm-5-fpgas-and-socs/iopll-ip-parameters-advanced-parameters-tab.