Implementation of ADPLL Networks on FPGAs

Conor Dooley

This thesis is submitted to the School of Electrical and Electronic Engineering

in the College of Engineering and Architecture of University College Dublin

in partial fulfilment for the requirements for the degree of

Master of Engineering

Research Supervisors:	Dr Elena Blokhina & Brian Mulkeen
Head of School:	Prof Andrew Keane

April 2019

Contents

List of Figures

List of Tables

Acronyms

ADPLL All-Digital Phase Lock Loop

ASIC Application Specific Integrated Circuit

C2C Cycle-to-Cycle

CPU Central Processing Unit

DCO Digitally Controlled Oscillator

EDA Electronic Design Automation

FIR Finite Impulse Response

FPGA Field Programmable Gate Array

FSM Finite State Machine

GSLS Globally Synchronous Locally Synchronous

HDL Hardware Description Language

IC Integrated Circuit

IIR Infinite Impulse Response

k_i Integral Gain

k_p Proportional Gain

LF Loop Filter

MSB Most Significant Bit

NCO Numerically Controlled Oscillator

PD Phase Detector

PFD Phase Frequency Detector

PI Proportional Integral

PLL Phase Lock Loop

Pmod Peripheral Module interface

RO Ring Oscillator

RTL Register Transfer Level

SoC System-On-Chip

SR Set-Reset

TDC Time-to-Digital Converter

TDL Tapped Delay Line

TIE Time Interval Error

UCD University College Dublin

VCO Voltage Controlled Oscillator

XOR Exclusive Or

Acknowledgements

I would like to thank my supervisors Elena Blokhina and Brian Mulkeen for their support and guidance throughout the course of this project, without which I would have been spinning my wheels.

Thanks are also owed to Pierre Bisiaux for his assistance in identifying problems with my designs and the explanations thereof, and without whom the measurement process would have been an order of magnitude longer.

Abstract

Low power, high frequency clock distribution systems will be in ever increasing demand in the near future as the need for high performance digital circuitry grows. At these frequencies, however, the conventional clock distribution systems are unable provide a clock signal of adequate quality without compromising on either of these problems. Many devices have turned away from using Globally Synchronous clock distribution systems in favour of those that divide the area of the chip into Globally Asynchronous but Locally Synchronous areas. However, new technologies seek to enable the use of Globally Synchronous methods at high frequencies, one of which being the use of oscillators coupled in phase, each responsible for delivering the clock to a subregion of the chip. Each oscillator forms part of a Phase Lock Loop (PLL), and in order to enable the synchronisation of the network each PLL is linked those controlling the adjacent clock regions. For a digital system it is expedient to implement these PLLs digitally as an All-Digital Phase Lock Loop (ADPLL) as this provides a number of advantages. It has been shown in theory and experimentally that this method can produce the high quality clock desired with low power consumption.

As the production of test chips is expensive and time consuming through simulation and validation of a design is vital, traditionally carried out for mixed signal circuitry using complex behavioural and theoretical models. For an ADPLL Network the consistency of a signal both with respect to itself, and to the other clock signals on the chip is a key performance attribute and much of the variation is due to random processes that may be difficult to simulate effectively.

This thesis will demonstrate that the Field Programmable Gate Array (FPGA) can be used in order to simulate, model or validate ADPLL network architectures in a cost and time effective manner, as a complement to conventional methods. The key benefit is that many system dynamics that will be seen when a network is implemented on an Application Specific Integrated Circuit (ASIC), but be overlooked in software simulations can be examined in the hardware simulation that an FPGA provides.

This thesis implements networks using three designs of ADPLL, each using a different architecture, and highlights the use cases to which each is best suited. The performance of each design is then analysed and this compared to the suggested use cases. Additionally the impact of more minor architectural modifications is tested and documented.

Lay Abstract

With the increasing proliferation of “smart” devices the need for low power yet high speed devices has never been greater. Each smart device contains a processor to control the device, each containing complex circuitry, requiring extremely exact synchronisation. The task of this synchronisation falls to the “clock” signal which must occur at the same instant in all areas of the processor. Conventional solutions to this problem cannot satisfy both power and speed requirements simultaneously, so designers have proposed the ADPLL Network which approaches the problem from the other side. Rather than generate the clock once and send it around the chip, which consumes a large amount of power, an ADPLL network divides the chip into a grid and generates many signals that each serve a local area, only requiring non time critical control signals to be sent over large distances.

The production of chips to test designs is both expensive and time consuming so designers must ensure that mistakes have not been made, accomplished by simulating the behaviour of the designs using complex behavioural and theoretical models. This thesis will discuss the use of Field Programmable Gate Arrays (FPGA) as a hardware testing, simulation and validation platform, to be used prior to test chip production, that is both cost and time effective. The hardware nature of an FPGA enables the analysis of behaviours that would not be possible in software, without the cost and time penalties of a custom chip. This is made possible by the ability to reconfigure the chip at will, albeit with comparatively lesser capabilities.

This thesis implements networks using three designs of ADPLL, each using a different architecture, and highlights the use cases to which each is best suited. The performance of each design is then analysed and this compared to the suggested use cases. Additionally the impact of more minor architectural modifications is tested and documented.

Chapter 1

Introduction

This thesis will put forward the Field Programmable Gate Array (FPGA) as a tool in the design of All-Digital Phase Lock Loop (ADPLL) networks to bridge the gap between software simulations and implementation in custom silicon by providing a hardware-based simulation, modelling and validation platform. While an FPGA lacks the direct control over the schematic and layout that an Application Specific Integrated Circuit (ASIC) provides, the hardware nature of this platform enables the analysis of system dynamics that are not easily modelled in simulation and the testing. ADPLLs are an entirely digital implementation of a Phase Lock Loop (PLL), advantageous as this allows for easier integration into the modern high transistor count Integrated Circuit (ICs) for which ADPLL networks are a proposed clock distribution system.

The goal of this project is to design and implement an extensible platform that can be used by the research team in University College Dublin (UCD) going forward as they seek to understand the behaviour of ADPLL networks at a higher level and to serve as a hardware test-bed for proposed new architectures or system components. In order to accomplish this goal a number of potential ADPLL architectures will be investigated, implemented and tested to ensure they are function correctly. Individual ADPLLs, however, will not give sufficient insight into the behaviour of a network, so once the ADPLL designs have been established, each will be implemented as part of a network of increasing sizes. Each contrasting design will be analysed based on the results of measurements and tests, and these results will be used to corroborate claims made regarding which FPGA based ADPLL design or ADPLL network architecture is best suited for particular use cases.

The design of each block, or component part, used in the network will be discussed, starting with the reasons for their selection and an explanation of the design methodology, along with any major pitfalls encountered their creation. The impact on performance caused by modifying the design of these blocks will again be assessed on the basis of measurement results, before comparison is made to both theoretical expectations and their use case.

An FPGA based test platform is ideal for those who wish to examine system dynamics without

the time delays, financial burden or expertise required to develop a complete mixed-signal system on an ASIC but retain the ability to realistically simulate the behaviour of an ADPLL network. The result of this project is such a platform, designed to be extensible, with flexibility built into each component/module used.

Chapter 2

Background Review

2.1 Brief Overview

In a world where the demand for high performance hand-held computing devices continues to grow and the prevalence of "smart" devices is increasing, there is unprecedented demand for System-On-Chip (SoC) devices to control systems as varied as medical devices and entertainment systems. As these applications become more and more demanding, with ever increasing amounts of data to process and the expectation that today's devices will outperform those of yesterday, the problem of maintaining the steady gain in performance of SoCs remains at the fore.

The main drivers of performance in SoCs are the number of transistors on a chip, which is correlated with the number of calculations that can be carried out simultaneously, and the frequency at which the device operates, which determines the number of calculations performed per second. Moore's Law, based on the famous observation by Gordon Moore in 1965 [1], predicted a doubling in the transistor count of Integrated Circuit (ICs) per year for the forthcoming decade. This behaviour has carried on to this day as a result of the ever decreasing size of transistors, and has only begun to slow down in recent years. However, as the number of transistors on a chip has increased roughly following Moore's Law, the increase in clock frequency has not been able to follow a similar linear trajectory, having remained roughly equivalent for the last number of years [2], indeed the clock speed of the Intel Core family of Central Processing Unit (CPUs) has not changed since their introduction in 2009 [3].

This plateauing of clock frequency has been caused by high power consumption due to the demands placed by the global distribution of a high frequency clock, often the single biggest consumer of power on the chip [5]. With the growth of the Internet Of Things (IOT) market where low power devices are desirable, with many of the emerging uses of SoCs being portable and thus without a permanent power source, high power consumption goes directly against one of the key pillars of the technology. This forces many of these devices to use lower performance hardware in order to

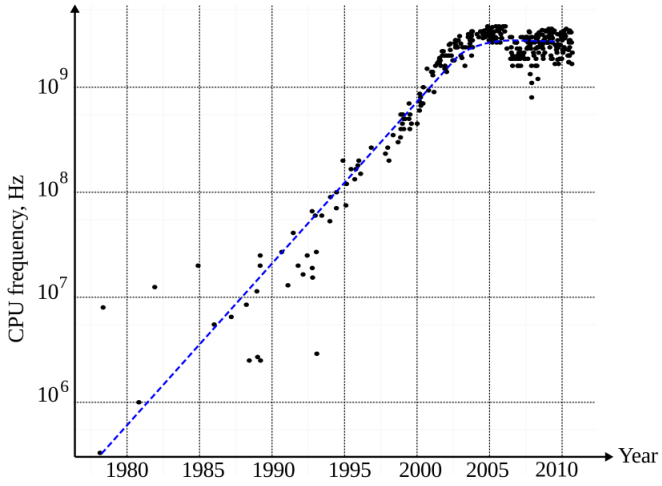


Figure 2.1: Frequency of the Intel microprocessors over past 30 years [4].

reduce the power consumption, and increase the battery life, of their devices.

In digital systems, two main approaches are used when designing the clocking system. In both cases, the chip is broken down into small areas in which all transistors are clocked synchronously, with the size constrained by the ability to deliver a quality clock signal to all transistors. The first of these methods is Globally Synchronous Locally Synchronous (GSLS), where the clock signals in each of these subregions of the chip are synchronised with one other. In practice, however, this is very difficult to achieve, as extremely high precision is required across the ever increasing number of transistors and the entire area of the chip, and doing so leads to high power consumption.

In contrast in a Globally Asynchronous Locally Synchronous (GALS) clock delivery system the "local" areas are not synchronised with other. This reduces the clocking system's complexity and thus the power consumption and chip area used, at the expense of communication speed between blocks. This disadvantage comes from the need to then somehow synchronise the messages being sent from one area to another to avoid the corruption of any messages. A GSLS, system, however has the advantages of deterministic behaviour and greater rates of communication between clocking areas and, as such, remains a desirable system design. A number of methods which deliver GSLS clocking exist at present such as clock trees as well as emerging technologies such as ADPLL networks.

2.2 The Impact of Clocking Errors

In Figure 2.2 the data path between two synchronously clocked registers is shown, with the circuit's function being carried out by the combinatorial network between the registers. Each register has a setup time, which represents the amount of time that the input value to a register must remain constant before the clock edge, and a hold time, the time for which the input must remain constant after a clock edge.

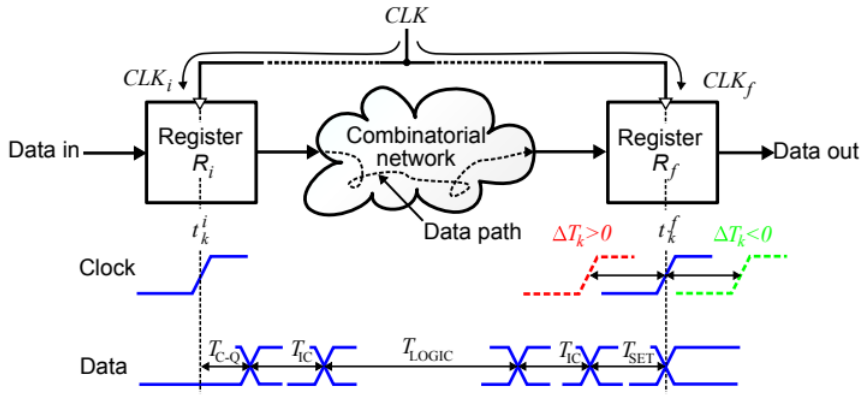


Figure 2.2: Data flow in a clocked system [4].

A lack of synchronisation between the clock edges will manifest itself as a time difference between the clocking events at both registers, $\Delta T = t_k^i - t_k^f$. ΔT is considered to be ergodic and can be described by an average deviation called skew and random process, normally modelled as a Gaussian random variable. If ΔT is negative this reduces the time available for the intervening combinatorial network thereby, having the same effect as a reduction in clocking frequency. Correspondingly a positive ΔT for depicted registers implies a negative ΔT for R_f and the subsequent register. The most common sources of clock error are caused by mismatches which usually stem from production, such as differences in the length of clocking paths, buffer delays or in the parameters of either active or passive components in the clock distribution network, which as the size of components on an IC reduces becomes more difficult to avoid. All sources of mismatch will manifest themselves in the clock distribution system as skew between transistors, while the noise in active components or the power supply system will appear as jitter in the clock signal.

2.3 Traditional Solutions

A number of traditional solutions exist which provide GSLS clocking systems, using a variety of techniques. The most simple of these implement clock distribution systems that are symmetrical in order to distribute a centrally generated clock signal to all areas of the chip at the same phase. These systems are named in accordance with their geometry, with the most common variants being branch, X or H trees.

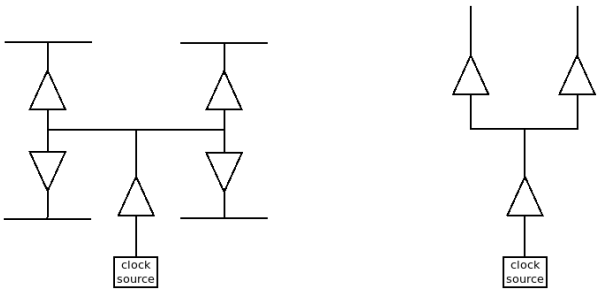


Figure 2.3: H and branch tree clock distribution systems.

While on the surface these appear simple, the task of obtaining an exact matching is, in practice, the limiting factor in this design. Even if the clock distribution system is geometrically symmetrical by design, production mismatches in either active or passive components will lead to a skew that varies from part to part. In order to minimise the impact of production tolerances, the dimensions of components in the distribution network can be increased, thus reducing the relative variation possible. However this has the impact of increasing the power consumption of the distribution network [5].

A mesh clock distribution network is an alternate design where the clock is delivered using a Cartesian grid of distribution lines. Compared to a tree type system, the variation in skew seen with a clock mesh is inversely proportional to the density of the grid while the sources of jitter remain identical. According to Abdelhadi *et al* (2010) clock meshes "*achieve low and deterministic skew, low skew variations, and low jitter*", all desirable characteristics for a clock distribution system. However they dissipate more power due to extra capacitive loading, attributable to vast number of lines required to form the grid. Similarly mesh distribution networks suffer from potential mismatch in production and alleviation through increasing of the dimensions of interconnects will, as with a tree type system, lead to higher power consumption [6]. Alternative designs replace the electrical lines used in the tree networks with waveguides for optical signals, with only the distribution

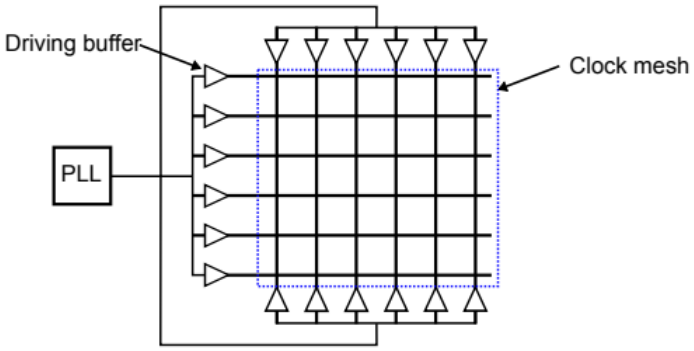


Figure 2.4: Mesh clock distribution system [4].

in the local area carried out using regular wires. This technique presents many advantages [7]: optical clock delivery is immune to the noise sources that affect electrical clock distribution systems, consume less power and do not suffer from the electrical losses present in a regular tree system.

2.4 Skew Compensation

In a tree type distribution system, skew is the main issue affecting clock accuracy and as such some effort has gone into addressing the problem. Skew due to the manufacturing process can be, at least, partly accounted for by means of active control through a skew compensator. This is a circuit, or controller, that compares the skew of each local clocking area on the chip and attempts to ensure in-phase clock delivery. Two main strategies exist to provide skew compensation, each named according to the location of the control mechanism. Designs featuring the controller located at the clock source, are known as “centralised” methods, and those with multiple controllers in the individual clocking areas known as “decentralised”. Regardless of the controller placement these techniques allow for the tuning of the propagation delay between the centralised clock source and the local clocking areas.

In a centralised skew compensation circuit, the skew across the chip is calculated by the central controller which then manipulates the distribution network in order to deliver a more in-phase clock around the chip. This calculation is done by measuring the round trip time from the clock source to both the root of local clock tree, and to the individual “leaves” of the tree. The controller then has a limited ability to tune the propagation path. The downsides of this technique are the resolution of both the measurement and compensation are poor, allowing for the correction of just skew and

not of any jitter that may be present in the system, and that the extra circuitry required for both the tunability of the forward path and the two extra return paths contribute to an increased footprint and power consumption.

As the name suggest a decentralised skew compensation technique delegates the responsibility of tuning the propagation path to the individual clock regions. This strategy has the advantage of not requiring the return paths present in a "centralised" design. Instead comparison is made betweeen the leaves of different clocking areas and on this basis the propagation delay is varied. For example, Yamashita *et al* (2005) designed a system in which each clocking area or "leaf node" contains a partial clock tree. Each of these "leaves" is able to compare its clock phase to the neighbouring node, and based on the result, tune an adjustable delay buffer [8]. While this method can compensate for process, voltage and temperature variation, it does not address the power consumption due to the delivery of a high frequency clock across the entire chip area nor does it have any impact on clock jitter.

2.5 Multi-oscillator Designs

The designs described previously, are all similar in that they have a single central oscillator that provides the clock for all areas of the chip, whereas the following methods attempt to synchronise multiple oscillators, each of which provides the clock for a single clocking area. The main advantages of a multi-oscillator design, are that as each clocking area has its clock created locally, there is no degradation in the quality of the signal as it is distributed around the chip and the number of potential noise sources is reduced. In order to obtain global synchronisation some method of comparison between local clocking areas is required, and how this is done depends on the architecture. Regardless of the comparison is made, it is carried out between neighbouring clocking areas and as such the feedback network need not have a large footprint or power overhead.

One such method is a network of oscillators as in Figure 2.5 which uses coupled Phase Lock Loop (PLL) to generate local clocks. Here the output of a leaf node is compared with an external reference and the operating frequency of each Voltage Controlled Oscillator (VCO) tuned based on the result. The advantage of this method is the simplicity of the feedback network, requiring just the divided clock output from a single leaf node. The VCOs then adjusted by the control voltage, v_c, which needs delivery to all areas of the chip. However, this is a regular signal and as such

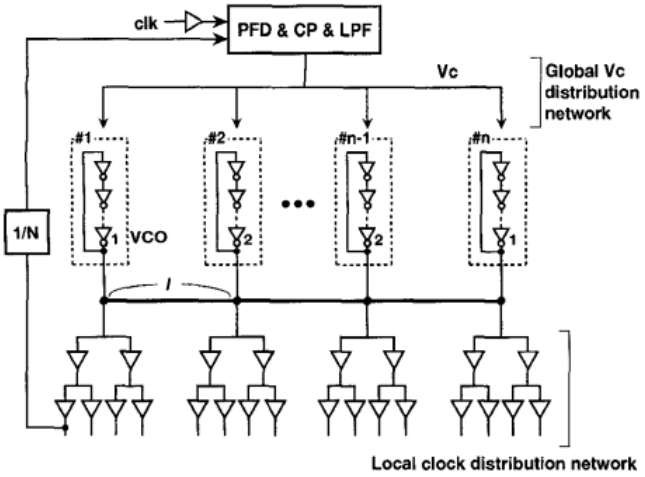


Figure 2.5: Coupled oscillator clock delivery circuit [9].

does not suffer from skew or jitter. This alleviates the need for a power hungry distribution circuit, while also being more noise-immune than the transmission of a high frequency clock. However this design still suffers from clock variation as all VCOs are fed the same control voltage, and thus the manufacturing tolerance issues present in conventional designs persists here also. This is acknowledged by the authors:

> *Unfortunately, as with the conventional ... method, distributing the VCOs over the entire chip causes the problem that jitter and skew are increased by variations in the fabrication process (static), temperature, and power supply (dynamic)* [9].

This type of multi-oscillator design is implemented by analogue circuits, and as a result not only are the clock signals, but also the control signals are liable to variation due to noise, fabrication mismatch and power supply dynamics.

Another potential multi-oscillator clock distribution system uses the phase relationship between the oscillators driving neighbouring clock areas in order to obtain synchronisation. Once again, this negates the requirement for a global distribution structure and the signals used for comparisons need only be sent between neighbouring clocking areas. As a PLL is being used it is again possible to perform the phase comparisons using a divided version of the generated clock. This in turn means the hardware transporting the divided clock signal to the phase comparator, has significantly lower requirements placed on it, thus lowering the power consumption due to electrical losses. Pratt and Nguyen initially proposed method of clock distribution in their 1995 paper entitled "*Distributed Synchronous Clocking*" in which they propose a Cartesian grid of clocking areas, each

with their own PLL, which has become known as a PLL Network [10]. In this design any given node is synchronised with its neighbours and one of the corner nodes is additionally synchronised with the reference. According to the authors this is "a simple, effective way to achieve low cost, high quality, low skew clock generation in a synchronous parallel processor". They did, however, note the presence of a phenomenon called "mode locking", which is setting of the network into a stable equilibrium where there are non-zero relative phases.

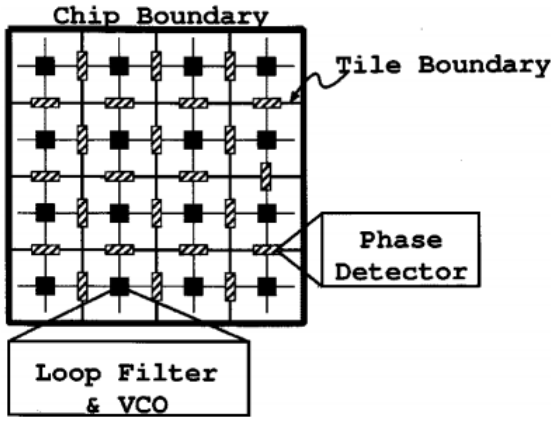


Figure 2.6: PLL network topology [11].

This architecture of clock distribution network was then implemented by Gutnik *et al* (2000) who fabricated a 4x4 array of oscillators, operating at a centre frequency of 1.2 MHz. The oscillator was implemented as a voltage controlled "nMOS-loaded differential ring oscillator", and in order to mode locking the phase detector was implemented as a highly non-linear circuit. The design was a success and the authors concluded:

> *Design and measurements on this chip confirm that generating and synchronizing multiple clocks on chip is feasible. Neither the power nor the area overhead of multiple PLLs is substantial compared to the cost of distributing the clock by conventional means* [11].

The remaining benefits of such a clock distribution system are: As the individual oscillators have their own control signal mismatch between different oscillators is not a factor as they will also receive different control signals. Secondly, and unlike the conventional methods, sources of jitter in the system such as power supply dynamics can be accounted for. Finally symmetry between the different oscillators is not required, once again attributable to the individual control signals in use.

2.6 ADPLL Networks

As a PLL network is an analog circuit, its integration in a modern IC is a barrier to usage, and as such it has not been used in any commercial designs [12]. An alternative design that is more suitable for current fabrication techniques eschews from using analogue components and instead implements the network of controlled oscillators using only digital circuitry, hence the name All-Digital PLL. A 4x4 All-Digital Phase Lock Loop (ADPLL) network was designed and prototyped in 65 nm CMOS by Zianbetov and Shan in order to test the suitability of the technique as a clock distributor [4, 13].

In this design the oscillators are once again laid out in a Cartesian grid, with each node coupled to their neighbours in phase. As this is now a digital system the coupling is carried out using digital phase comparators, which attempt to measure the phase difference between two oscillators. Figure 2.7 shows high level detail of the architecture of both the entire clocking system and that of an individual node in the design. The digital nature of this architecture brings with it a number of advantages over traditional analogue implementations, as it can benefit from advancements in digital circuit design suites, be reconfigurable and programmable and has a significantly greater immunity to perturbations inherent to its digital nature, as the exact voltage of signals is of no importance [4]. This last advantage is of particular use in a digital environment, as otherwise there is potential for clock degradation resulting from switching of transistors. The drawback of the switch to a digital architecture however is the presence of quantisation. Analogue designs both deliver continuous control signals to the oscillators and have a continuous phase detection capability, unlike a digital system where these actions are carried out with fixed resolution. Looking at the design of a given node it is notable that the function carried out by the Error Combiner is akin to a average, therefore as mentioned by Pratt and Nyugen, there is potential to a mode locked equilibrium in which the oscillators are not synchronised. In their paper, they presented a method where initial start-up was performed uni-directionally and, once all nodes are close to alignment, full connectivity could be restored, however this was not viable in an analogue system as reconfigurability was not an option [10]. Javidan *et al* (2011) found that this was in fact the case, and stated:

> [Mode-locking] is solved in a simple and original way, by a dynamic reconfiguration of the network interconnection topology at the starting stage.

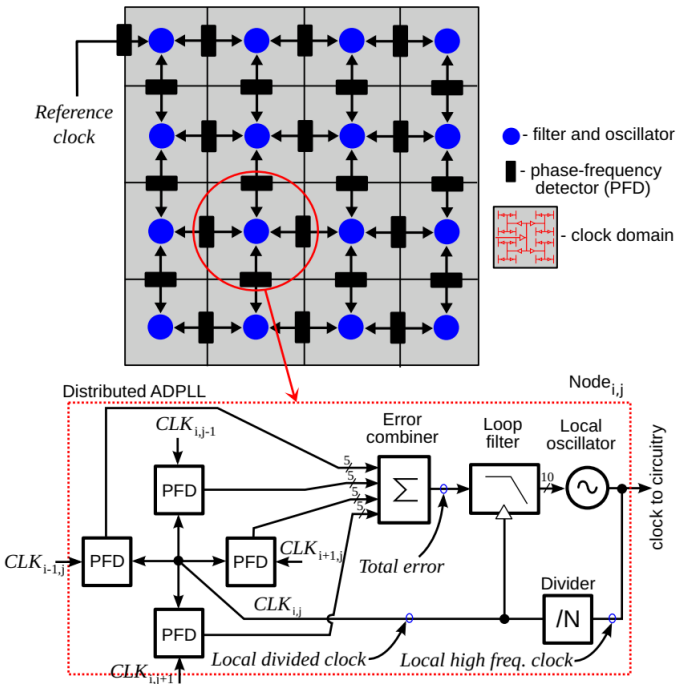


Figure 2.7: Architecture of the ADPLL network and of a single node [12].

In creating an entirely digital system, Zianbetov and Shan could easily reconfigure the network topology and by implement a uni-directional start-up avoid the problem of convergence into a mode locked state, without having to design a non-linear phase detector.

2.7 ADPLL Architecture

As indicated in Figure 2.8 the three main building blocks of a conventional PLL are the Phase Detector (PD), Loop Filter (LF) and VCO. In an ADPLL these blocks are then replaced by their digital counterparts, necessitating quantisation in order to remain physically realisable. The "All-Digital" moniker is a misnomer as the oscillator and Phase Detector are usually both implemented by mixed signal circuits.

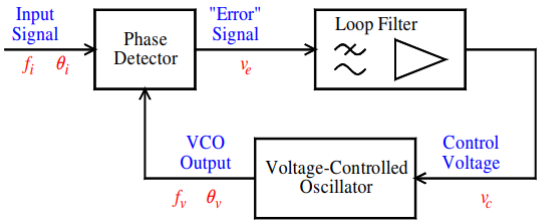


Figure 2.8: Block diagram of a Phase Lock Loop, *Wireless Systems Notes*, B. Mulkeen (2017).

2.7.1 Digitally Controlled Oscillator

In a digital system there are a very limited number of voltages representable, most commonly just two, so using a voltage to control the oscillator is not a viable strategy. Instead a fixed bit width signal is used to control the oscillator's period, selecting the number of inverters in a ring oscillator or the varactor configuration of a travelling wave oscillator [14]. The decisions made in the design of the Digitally Controlled Oscillator (DCO), or Numerically Controlled Oscillator (NCO), determine many of the other ADPLL parameters. While tuning range and centre frequency, as well as linearity, carry over from the analogue counterpart, a DCO also has a frequency step which in combination with the bit width of the control signal determines the range over which the oscillator can be tuned. Figure 2.9 illustrates a basic ring oscillator design. A ring oscillator is an inherently unstable circuit composed of an odd number of inverters connected in a circle, which allows a signal to propagate infinitely, with the signal at any point in the circuit appearing as a square wave. The half-period of this oscillator is the time taken for the signal to propagate once through the chain, n times the propagation delay through one inverter. The frequency of operation can then be set by modulating the length of the chain, in steps of two inverters to maintain an odd number, by means of `f_select`. The main impact of output frequency quantisation is that only frequencies which are integer multiples of the frequency step away from the centre frequency can be easily reproduced, with intermediate values only obtainable in a manner akin to Fractional-N synthesis with the control code toggling back and forth. This acts as a source of jitter in the system.

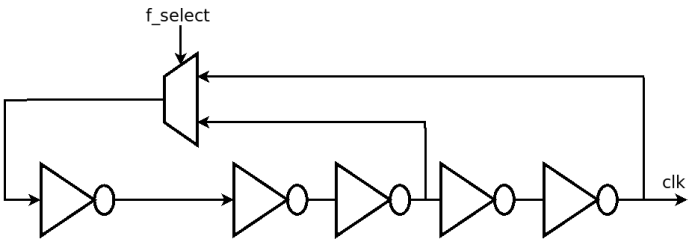


Figure 2.9: Basic ring/inverter chain oscillator.

It is also possible to implement an NCO by means of a counter, in a manner that will produce either linear period or linear frequency steps. Both methods use the most significant bit of the counter's value to form the output signal. Period linearity is achieved by varying the reload value of the counter after overflow depending on the control code, thereby changing the period by a

multiple of a fixed step. Alternatively frequency linearity can be achieved if the reload value is left constant, but instead the amount added to the counter every clock cycle is changed according to the control code, once again a fixed step size is used.

2.7.2 Digital Phase Detector

Once again quantisation impacts the Phase Detector, as rather than a continuous output the phase detector in an ADPLL has a finite number of output values, thus limiting the accuracy of the phase detector. A second form of quantisation is also present, as unlike an analog system, a digital phase detector does not provide continuous data in the time domain either, instead relying on sampling. At its most basic, a digital phase comparator may only output an indication of which signal is leading, a design known as a Bang-Bang Detector, which can be constructed using a single D Flip Flop with one the generated signal connected to the "D" input and the reference signal acting as the clock. As the output only has two levels the resultant word is only 1 bit wide and as such, limits the range over which the output frequency can be controlled. More complex designs such as that in Figure 2.10, implemented by Shan, build on this by measuring the time difference between edges of the signals using a Time-to-Digital Converter (TDC) in his case using a Tapped Delay Line (TDL) [13]. A TDL is constructed by a chain of elements of a fixed delay, and the signal to be timed is applied to this start of this chain. After the timing interval elapsed the values at each point in the chain are examined, and using temperature coding, these are converted to a digital signal, the width of which is the bit width of the PD's error signal. This mimics a time measurement and allows for the phase difference to be recorded in a non binary manner.

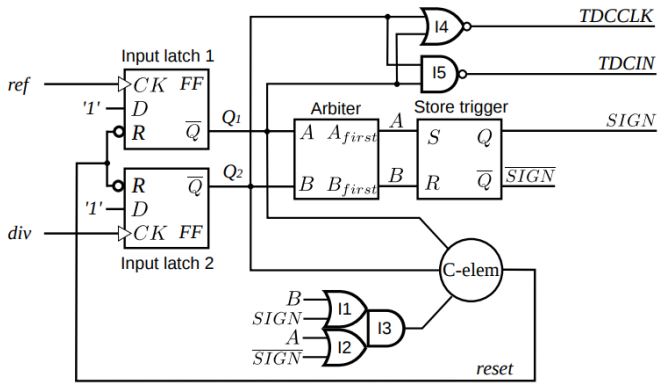


Figure 2.10: Bang-bang phase/frequency detector architecture [13].

2.7.3 Digital Loop Filter

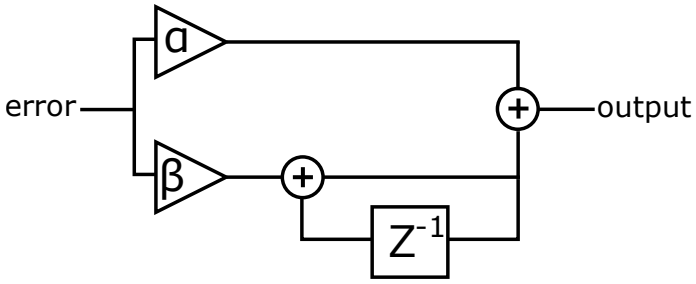


Figure 2.11: Basic PI controller architecture.

The Loop Filter in an ADPLL can be implemented as a PI controller, as only a low-pass filter is required, such as that in Figure 2.11. In the case of a node in an ADPLL network the input of this filter is a weighted of the phase difference relative to the neighbouring local clocking areas. In one example topology a digital system the proportional section can be implemented by a simple multiplier, whereas the integral path is constructed by adding the result of a multiplication by the proportional gain to an accumulator. This delayed summation can be easily implemented by an accumulator to which the current value of the multiplication is added each cycle, and as such the system has an infinite impulse response. The value of these gains determine the response and stability of the ADPLL network. The transfer function of such a controller is given by [13]:

$$H(z) = \alpha + \beta\frac{1}{1-z^{-1}} = \frac{(\alpha+\beta)-\alpha z^{-1}}{1-z^{-1}}$$

It has been found by Koskin *et al* (2018) that stable operation can be achieved when the integral gain, k_i, is less than half the proportional gain, k_p [15]. In the same study a range of values was found which would produce low jitter operation of the network. As these values are all less than one, the filter must implement fixed point arithmetic in an effort to maintain the simplicity of the clock distribution network, rather than incurring the penalty of floating point calculations.

2.7.4 Error Combiner

The ADPLLs used in a network need to combine the error signals from multiple neighbours to determine what the average difference from its neighbours is, and this necessitates the addition of the Error Combiner. In a digital system this can be implemented by a weighted average of the different error signals, with the weight being modifiable at run-time. This configurability is what

permits the system to implement uni-directional mode and also allows for the weighting applied to certain signals, such as the external reference, to be modified. The ease of implementation of a configurable error combiner is one of the main advantages of an ADPLL over an analogue system.

2.8 The Role of the FPGA

A Field Programmable Gate Array is a type of IC that is designed to be configured by a designer after the chip itself has been manufactured. An FPGA contains a large number of logic elements that can be connected together in order to perform complex logic, written using the same Hardware Description Languages (HDLs) used to design the digital blocks of Application Specific Integrated Circuits (ASICs). They may also implement started modules such as adders, multiplexers and Random Access Memory (RAM) as a fundamental element. More complex logic is often implemented using multiplexed lookup tables rather than true logic elements. High end devices such as the Xilinx Zynq Ultrascale even implement Multi-Processor SoCs. Compared to an ASIC the designer does not have direct control over the layout of the system but rather describes its behaviour, possibly down to the basic logic elements of inverters or other gates. Limited control is possible over the placement of the individual modules, but Electronic Design Automation (EDA) tools are responsible for the exact placement of elements. As a result, it is not possible to have precise control over the delays experienced as signals propagate through the design. To assist with the resolution of any issues EDAs provide tools to analyse timing behaviour.

Prototyping on an FPGA is a common verification stage for conventional ASIC designs as it allows for a hardware validation of any digital circuitry, and the detection of any potential flaws or errors made by the design before the expensive of an ASIC implementation. In their 2013 ADPLL network implementation Zianbetov and Shan used an FPGA in order to validate their programming interface, the design of the error processing block, ensure they had eliminated mode locking behaviour and to ensure phase synchronisation was possible [4, 13]. However they experienced two main limitations, they were not able to implement the mixed-signal Phase Frequency Detector (PFD) and DCO, and the maximum frequency of operation possible was orders of magnitude lower than the GHz range of their ASIC implementation. These issues were circumvented by implementing an alternative PFD and DCO designs which were driven by the clock distribution network provided by the FPGA and every clock frequency scaled by the same amount such that the results

of testing would remain indicative. One of the main advantages they saw was that the hardware description used for the digital blocks of their ASIC could be directly ported over to the FPGA.

A similar technique was used by Lata *et al* (2013) to implement an FPGA clocked oscillator with a 200 kHz centre frequency, however it is hard to understand exactly what either the goal was or the specifics of the DCO used [16].

It is however possible to implement limited mixed-signal circuits on an FPGA through the use of primitive logic elements, however, as control over the implementation is restricted to the module level it is not possible to mirror the implementation of a design intended for an ASIC. As these implementations are analogous to a mixed-signal circuit on an ASIC, the verification of theoretical behaviours, as done by Koskin *et all*, in order to test the findings of his PhD thesis [17], is made possible without the expense of ASIC fabrication. An FPGA based mixed-signal ADPLL network is also seeing use here in University College Dublin (UCD) as a initial prototyping platform for the validation of new modules for use in ASIC based ADPLL networks.

FPGAs have been used in other fields to simulate and experiment with new technologies, although not all of these attempt to implement the mixed-signal circuitry using primitive elements. Fernandez-Alvarez *et al* (2016) proposed a method for prototyping high-voltage system controllers suited to higher end FPGAs in which a co-processor on the FPGA simulates the mixed-signal circuitry while the digital section of the design is implemented on the FPGA itself [18]. While the interfacing between hardware and software remains a challenge they found:

> Obtained data are compared to the data obtained by means of using ... simulation. The proposed solution speeds up the evaluation in around one order of magnitude keeping the accuracy. The output signal differs in less than 0.6 mV (RMSD).

Mixed-signal circuits were, however, simulated in hardware by Óscar Lucía *et al* (2011) on an FPGA in order to overcome the excessive time penalty imposed by software based simulations that required the behaviour of both a digital and mixed-signal peripheral and that of a micro controller running code in C to be simulated side by side [19]. They concluded

> ... the proposed system provides a versatile and fast method to develop ad hoc control architectures, avoiding the need for time-consuming mixed-signal simulations and the risk of damaging the actual power converter implementation.

By carrying out simulations on an FPGA, Guanhua Wang *et al* (2013) achieved a 3000 times decrease in run-time when compared to an identical simulation in MATLAB used for the verification of a calibration algorithm for successive approximation analogue-to-digital converters [20]. Many other examples can be found of FPGAs used for the simulation of mixed-signal or RF circuits.

2.9 ADPLL Performance Characterisation

As already stated, goal of a clock distribution system is to synchronise clocking events in all areas of the chip. The effectiveness of this synchronisation is characterised by two main metrics, jitter and skew, which together describe the distribution of clocking events. Skew is the average time delay between a clocking event in one area of the chip and that of a reference event. It can be easily measured by computing the average value of this delay. Jitter has a number of definitions depending on how it is measured, but at its most basic it is the standard deviation of the time delays with respect to the reference clocking edge.

The simplest form of clock performance characterisation is on a Cycle-to-Cycle (C2C) basis, in which the reference is the signal under test itself. Here the standard deviation of the individual periods gives the jitter of the signal, while skew has no meaning with a self reference the mean period can be used to compute the centre frequency over the time interval. For a more informative measurement the Time Interval Error (TIE) can be computed, which takes into account another signal as the reference. TIE is calculated by comparing the delay between the signal in question and a reference that is treated as ideal. The mean value gives the relative skew between the two signals and once more the standard deviation of the measurements gives the jitter with respect to this reference.

Both above forms of measurement assume a large number of sequential measurements, however, there are other ways that jitter and skew can be calculated [21]. Phase jitter is an important characteristic for communications systems as it can be used to calculate phase noise, important to ensure spurious emissions are within regulations. Long term, or accumulated, jitter represents the cumulative effect of jitter on the signal over several cycles. Long term jitter in particular affects RADAR as it will manifest itself as a Doppler shift in the return signal. For a PLL network, with an appropriately designed filter, the long term jitter should be zero so long as the reference signal is stable.

Chapter 3

ADPLL Designs for FPGAs

3.1 Chapter Overview

The first step in creating a Field Programmable Gate Array (FPGA) based network of ADPLLs is the design of the ADPLL itself, which will be addressed in this chapter. The nature of an FPGA necessitates a number of compromises in the design of a given block which limits transferability to Application Specific Integrated Circuit (ASIC) designs. In this chapter the potential designs for each individual block, or module, investigated will be explained and the case for their selection in an FPGA based ADPLL made. A number of blocks have implement purely digital circuitry and as such can be transferred in their entirety from an FPGA and vice versa. However, those that will be used to emulate mixed-signal circuitry, such as the Digitally Controlled Oscillator (DCO) and Phase Frequency Detector (PFD), will be examined in greater detail. A Register Transfer Level (RTL) diagram of the ADPLL topology that will be assumed in the following sections is shown in Figure 3.1.

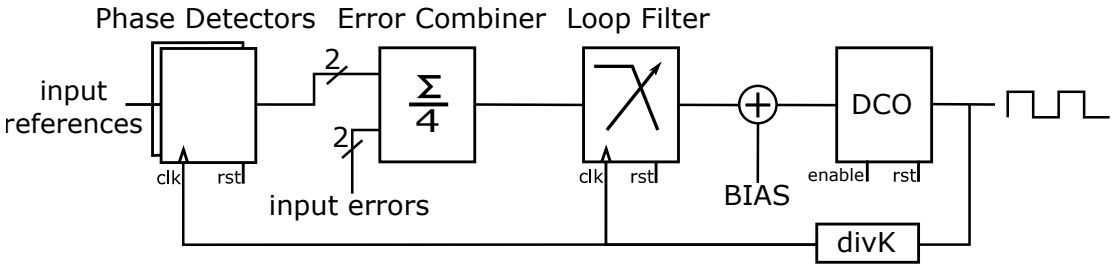


Figure 3.1: ADPLL designed for use in a Cartesian grid network.

Important to note is that despite a Cartesian grid leading to four neighbours, only two phase detectors are required per oscillator. Borrowing matrix indexing, and choosing the oscillator located at index i, j, $i, j \neq 1, N$ where N is the number of oscillators in each direction, comparison need only be done with the signals generated by oscillators located at $i-1, j$ and $i, j-1$. The comparison with oscillators $i + 1, j$ and $i, j + 1$ will be carried out in those ADPLLs, and the negation of the

errors measured used in the ADPLL located at i, j.

3.2 Digitally Controlled Oscillators

The choices made in the design of the DCO have the greatest impact on the effectiveness of the overall platform and which use cases the ADPLL containing it are suitable for, as the key performance benchmarks are all done using the waveform this block generates. This project will address three distinct designs of ADPLLs suitable for implementation on an FPGA, two derived from the clocks generated by the FPGAs own distribution network and one generated independently of this clock, using a chain of inverters. These are not the only ways in which an oscillator could be synthesised on an FPGA, however other designs were deemed to be unsuitable for extensible and portable implementations.

A prime example of this is the use of Xilinx proprietary `IODELAY` blocks to create an oscillator, as detailed in Xilinx Application Note XAPP872 [22]. The key idea here is that the bulk of the period is made up by the propagation time through one of the `IODELAY` blocks, which can be set at implementation time. This is combined with a section of an inverter chain, and a multiplexer used modify the length of this segment, the output of which is fed back into the `IODELAY` block. This method was discarded as the number of `IODELAY` blocks is very limited, so expanding to a larger network would be impossible, and they are all located around the edge of the chip, not suited to the construction of a Cartesian grid.

The main issue with the creation of DCOs on an FPGA is the inability to create mixed-signal circuits, such as those that would be intended for use on an ASIC. As such the FPGA based oscillator must emulate the behaviour of a mixed-signal circuit in some way.

3.2.1 FPGA Driven, Linear Period DCO

The first design of DCO to be examined is of the type used by Zianbetov in his ADPLL network test bed and relies on a counter driven by the clock manager on the FPGA [4]. At each event on the FPGA provided clock a counter is incremented, overflowing upon increment past the maximum possible value. The Most Significant Bit (MSB) of this counter forms the waveform generated by this oscillator, the period of which is controlled through an adjustable value that is loaded into the counter once overflow is reached, and forms the starting points for the counter. The period of

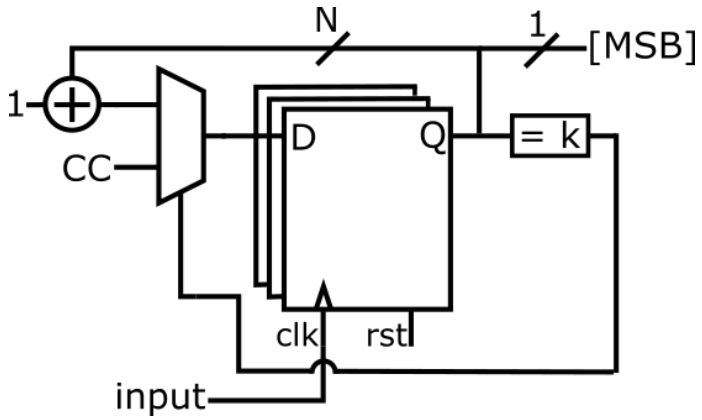


Figure 3.2: FPGA driven, linear period DCO RTL Diagram.

oscillation is given by:

$$T_{osc} = \left(2^{width} - \left(BIAS + CC\right)\right) \times T_{FPGA} \tag{3.1}$$

where T_{FPGA} is the clock period of the FPGA, CC is the control code input, $BIAS$ centres the oscillator in the middle of the tuning range in the event that the control code is zero and $width$ is the width of the counter used. As the only variable here is the control code, period step of this design is T_{FPGA}, thereby providing period linearity with respect to the control code. This is the key advantage of this design, as most ASIC implementations of a DCO are also linear in period. The other main reason to choose this design is that its FPGA clocked nature allows for exact control over the frequency of operation, and the number of tunable parameters make it possible to configure multiple ways to achieve the same frequencies of operation. Combined these attributes make it very easy to create an oscillator that emulates the behaviour of a design intended for an ASIC, however, at a greatly reduced frequency. This restriction on the frequency of operation arises out of the period step size, which in order to obtain a good resolution must be orders of magnitude smaller than the intended period to be generated. As the output waveform is taken from the counter's MSB, the reload value of the counter must never go beyond $2^{width-1}$, as otherwise the output waveform will become a constant 1. As the reload value varies the low time of the MSB, if the desired output waveform is a square wave this design will not be suitable.

In being FPGA clocked this design has pseudo-deterministic characteristics, with each period step being almost identical across oscillators and over the entire tuning range, unlike an ASIC where process variation will impact the layout of a high frequency oscillator. The only variation

in this design will come, ironically, from jitter or skew in the FPGA's clock distribution network, which as the frequencies will typically be in the low hundreds of MHz is very minor. In the case of the Xilinx XC7A100T-1CSG324C this is at most 100 picoseconds, or 0.05% of the period of an intended output clock at 5 MHz. To put this value into perspective, on this board the minimum value of T_{FPGA} that could be used to drive this oscillator is 3.87 nanoseconds, 1.935% of the period.

The resulting DCO is best suited to applications that do not seek to gain a better understanding of oscillator performance, but rather those focused on validating the entirely digital blocks in the system, the role in which Zianbetov and Shan used this type of oscillator [4, 13].

3.2.2 FPGA Driven, Linear Frequency DCO

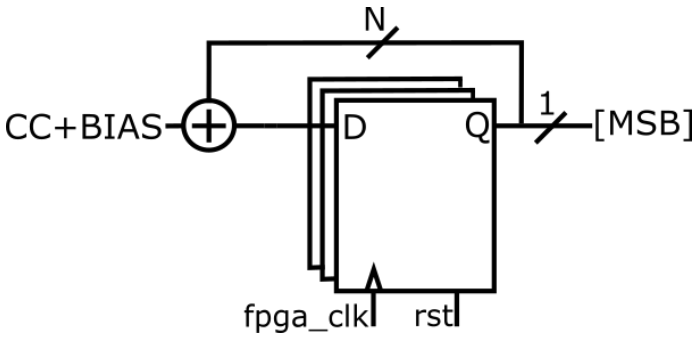


Figure 3.3: FPGA driven, linear frequency DCO RTL Diagram.

The second FPGA clocked oscillator is similar in most attributes to the above design but eschews period linearity for frequency linearity. Again the overflow property of a counter is used with the counter's MSB as output of the block, however, this time it forms a square wave. Rather than setting the reload value of the counter, instead the increment is adjusted depending on the control code, thus requiring $\frac{2^N}{BIAS+CC}$ increments to overflow. Accordingly the frequency of operation is set by:

$$f_{osc} = f_{FPGA} \times \frac{BIAS + CC}{2^{width}} \tag{3.2}$$

Here the control code CC and bias are added to the value stored in the accumulator at each event of the FPGA, clock until overflow is reached. This occurs at 2^{width} where, as before, $width$ is the bit width of the counter, thus valuing each control code increment at $\frac{f_{FPGA}}{2^{width}}$ Hz. As with the previous design, this oscillator is better suited to frequencies where the output of the DCO is orders of magnitude lower than the clock signal driving it, as this ensures that the incremental change due

to the control code remains a small fraction of the period.

This design is just as configurable as its linear-in-period counterpart, and well suited to the low frequency emulation of ASIC based oscillators that are themselves linear in frequency. In sharing the FPGA as a clock source again the pseudo-deterministic characteristics return, once more meaning this oscillator is better used for testing, simulating or verifying other blocks in the system.

3.2.3 Ring Oscillator

The best approximation of the non-idealities of an ASIC DCO implementation can be obtained using a Ring Oscillator (RO), also know as an inverter ring, as the DCO. In an ASIC an RO is constructed by connecting a number of NOT gates back in a chain, with the output of the last NOT gate fed input the input of the first. As NOT gates are not available on an FPGA, they are replaced by inverter primitives. The inverter ring is an inherently unstable circuit that leverages the propagation delay through an odd number of inverters to create a waveform that, when viewed at a fixed position, is a square wave.

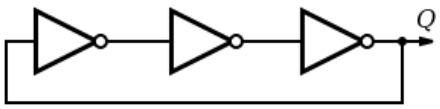


Figure 3.4: Basic ring oscillator RTL diagram [23].

The output value of the inverter at the end of the chain changes a non-zero amount of time after the input to the first element is set. It can be trivially shown that if the number of inverters is odd, this will be the inverse of the value initially applied to the first element. This sets in motion an oscillation, the period of which is governed by the number of inverters in the chain. The formula to calculate the oscillator's period is simple:

$$T_{osc} = 2 \times \tau_{inv} \times N \tag{3.3}$$

Here, τ_{inv} is the propagation time through one inverter, N is the number of inverters in the chain and the multiple of two arises from the necessity to invert twice to produce a square wave.

The main advantage of this type of oscillator stems from its similarity to that used on an ASIC which leads to many non-idealities carrying over from the truly mixed-signal implementation.

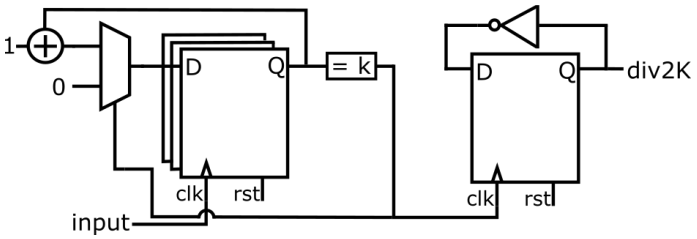
Rather than being just a tool to validate that other modules, the performance of this oscillator can be analysed more deeply as well as the performance of networks containing it. Koskin *et al* used such a design to test theories proposed regarding the synchronisation of oscillators during his PhD using this type of oscillator implemented on an FPGA [17, 24]. Also in this design's favour is the period step size, four times the inverter propagation delay, which was found experimentally to be than in the region of a nanosecond on Xilinx Artix-7 and Zynq FPGAs.

While it is advantageous for the purposes of analysis that this design is affected by variation in temperature, implementation and other non-idealities this brings with it challenges not present in a true mixed-signal design. In a design intended for an ASIC the design has near total control over the placement of circuit elements, however, the FPGA based counterpart control is available over just the area in which the oscillator may lie, and the Electronic Design Automation (EDA) is responsible for the exact placement. Unlike the ASIC implementation, in which the propagation delay through each element of the ring is well bounded, the lack of control over placement means that the propagation time through each inverter is not well bounded and may vary wildly. This makes it time consuming to ensure that the frequency tuning ranges of each oscillator match that desired, and use of this oscillator less straight-forward than that of the previous two designs. This is compounded by the simulation tools in EDA packages intended for FPGAs being unable to accurately simulate the time delays, thus negating any benefit of test benches for the alignment process.

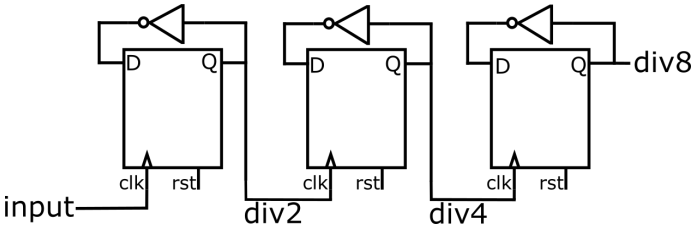
3.3 Frequency Divider

Many Phase Lock Loops (PLLs), be they digital or analogue, incorporate a frequency divider in the feedback path so that comparison can be made with a reference operating at a lower frequency. This is a feature that is advantageous for ADPLL networks, as comparison using this divided clock allows the synchronisation signal sent between modules to be of a frequency orders of magnitude lower than that of the clock distribution network and thus require circuitry that consumes significantly less energy. This division can be obtained trivially on an FPGA by a number of means, of which this thesis will mention just two. The first of these methods allows for integer division, by inverting a signal each time a counter reaches a certain value, k. When this condition is true the value stored in the counter is also reset to zero, thus implementing division of the generated clock

frequency by $2k$. By limiting to division by powers of two, the divider can be simplified with the



(a) Divider 1



(b) Divider 2

Figure 3.5: Frequency Divider RTL Diagrams.

removal of any comparison logic. One flip-flop and an inverter are required per power of two. As in Figure 3.5 (b), the inverted $\overline{Q}$ output of each flip-flop is inverted and connected to the D input, with the non-inverted Q of the previous flip-flop acting as the clock. The first flip-flop in the chain is clocked by the DCO output.

3.4 Phase Detector

Three potential Phase Detector designs will be addressed in this thesis, sign-only detection with a Bang-Bang Detector, FPGA clocked detection using a Finite State Machine (FSM) and an emulation of a Tapped Delay Line (TDL) using inverter primitives to create a delay, similarly to the RO seen previously. Both designs will have to have a representation for both situations where the generated signal leads and lags the reference signal, and for the magnitude of this difference, a problem most obviously solved by using a signed binary representation of the difference. Most easily implemented in hardware is two's complement, as operations can be carried out by the same hardware as unsigned arithmetic. The sign convention for phase error assigns the positive values to the situation where the generated signal lags the reference, and vice versa for the leading case. Apart from the Bang-Bang Detector, each design features two blocks, one to determine the sign of the error and a second which estimates the magnitude. These blocks could theoretically be mixed-and-matched, but as they operate on different paradigms no benefit would be obtained.

3.4.1 Bang-Bang Detector

A Bang-Bang Detector is a sign-only detector, thus outputting a phase error just a single bit wide. This can be accomplished using a single flip-flop, with the reference signal acting as the clock, in this case using the rising edge, a design verified by Predraig in his 2011 thesis [25]. The generated signal is applied to the D input, thus when a rising edge is seen on the reference the value of the generated signal is held as the Q output until the next rising edge of the reference. If the generated signal has gone high prior to the reference, and leading it, the value of Q will be 1, thereby matching the expected sign behaviour. Trivially it can be seen that the case where the generated signal lags the reference the Q will show 0.

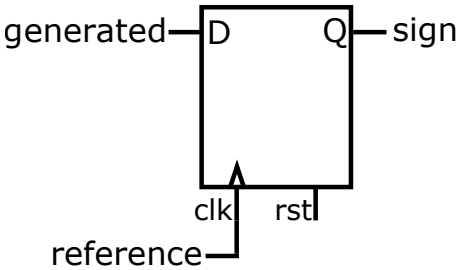


Figure 3.6: Bang-bang phase detector RTL diagram.

This design is simple but effective, and can even be used without a loop filter, albeit with a very limited oscillator tuning range. This is especially valuable an attribute at an early stage of platform development, as it may be used in order to test that other aspects of the system work as intended. The binary resolution of this detector, however, restricts its usefulness as the ADPLL cannot react more quickly to larger phase differences.

3.4.2 FPGA Clocked Phase Detector

The easier to implement of the two designs uses a combination of a state machine to determine which edge occurred first, and to control a two's complement up-down counter, which acts as the Time-to-Digital Converter (TDC) in this detector, and estimates the magnitude of the error.

3.4.2.1 State Machine Based Detection

While sign detection clocked by the FPGA can be performed in a variety of ways, a state machine is a straightforward way in which to accomplish this. A Finite State Machine consists of three main components: the states themselves, the next state logic and the output logic. In this case

the next state logic is the generated and reference signals, and the output logic sets the count instructions for the up-down counter as well as ensuring the phase error output is held constant between measurement periods. The FSM will have a state representing the case where either the reference or generated signals occurred first, in addition to a wait state used between the end of one measurement interval and the commencement of the next. Two types of FSM exist, Moore and Mealy Machines, the difference being that Moore Machines base their output only on the current state while Mealy Machines do so on both the current state and the inputs to the system. If a Moore Machines is used an extra state is required in order to hold the value of the phase error at the end of the measurement interval.

As this design is clocked by the FPGA, the phase error is quantised to the the FPGA clock periods, thus setting the resolution of the design. While, depending on the oscillator design, the generated clock may be synchronous to the FPGA, the reference signal arrives from off-chip and thus will not be synchronised to the measurement clock. In order to avoid any metastability events arising from an input signal changing within the set-up and hold times of the registers sampling them, synchroniser circuits are used to ensure the value sampled is either logic high or low at the instance of the measurement. As these synchronisers are the first FPGA clocked elements in the measurement chain, they also act as the system's quantisers. There is no degradation in detection resolution, as the signals would be quantised to the FPGA clock anyway.

There are three situations that may occur while the FSM is in the waiting state. Firstly the case where the reference signal sees a rising edge before the generated clock, and thus the generated clock is said to be lagging the reference. As previously mentioned the convention dictates that the phase difference have a positive sign, so the counter is incremented to count in the positive direction. Similarly if the generated signal is leading the reference the sign of the error is negative and thus the counter is instructed to count downwards. The third potential scenario occurs when both clock signals have a rising edge within the same period of the FPGA clock, which due to quantisation will be interpreted as both signals occurring at exactly the same time. In this case the counter receives no instruction and a zero value is stored until the next measurement interval. Once the FSM is in a measurement state, and the counter either incrementing or decrementing, a rising edge occurring on the opposite signal to that which initiated the measurement will end the interval.

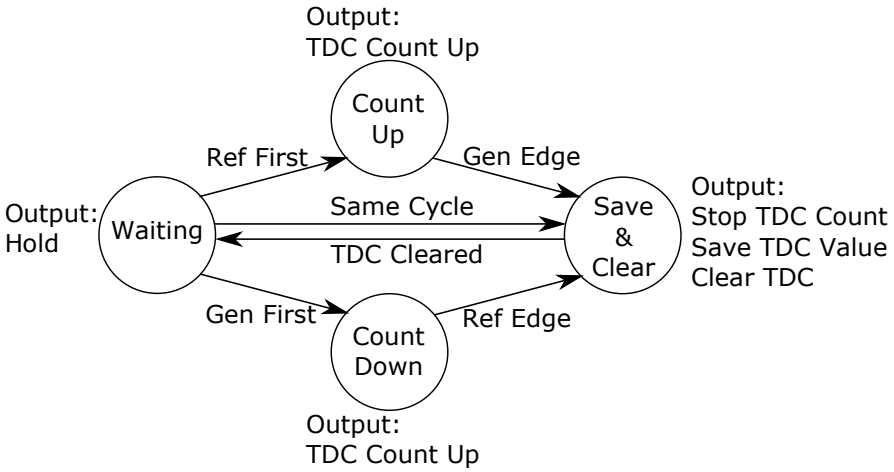


Figure 3.7: Example State Transition Diagram for a Moore Machine.

3.4.2.2 Up-Down Counter

The up-down counter in this design performs estimates the magnitude of the phase difference between the reference and generated signals. The state machine provides control instructions, indicating whether the count should be in the increasing or decreasing directions, and and holds the counter in reset once the measurement interval has been completed. The counter counts in two's complement, so counting in the downward direction can be carried out by adding on the representation of -1, the binary signal containing all 1s. The other advantage of counting in two's complement is that count value is already in the correct format for the follow block, the phase detector in a regular ADPLL or the error combiner in this instance, therefore the count value at the end of the measurement interval can be stored in a register, which maintains the output value of the phase detector between measurements. Two's complement is an advantageous way to represent signed numbers in hardware as the mathematical operations are identical to their unsigned counterparts, thus allowing use of any adder or carry hardware that may be present on the FPGA. An important attribute of this counter is the ability to freeze the count at the minimum and maximum count values in order to avoid a situation where the counter overflows, thus at high phase offsets potentially alternating between positive and negative differences which may restrict the locking range of the ADPLL. Disallowing overflow also aligns the behaviour of this phase detector with that of those implementing delay lines, allowing better interchangeability and comparison, which for a testing platform may be valuable.

This counter is best suited for use with the aforementioned FSM sign detection circuitry, and

would see no benefit from a combination with the un-clocked sign detection circuit that will be presented in the following section. This is the case as if quantisation to the FPGA clock had not already occurred in the sign detection circuit it would occur in this counter, as increments are only possible at FPGA clocking events, thereby enforcing quantisation to the FPGA clock. The coarseness of this measurement, restricted to time delay increments equal to the FPGA clock period means that it is an ideal detection method for the clocked oscillators, as the minimum phase difference that can be detected matches exactly with the minimum step by which the oscillator period can be tuned. This design can, however, be used with the Ring Oscillator, provided the Loop Filter gains are adjusted to maintain the oscillator to detector step relationship.

3.4.3 SigNum Detector

The SigNum detector is based on the methodology used in the Bang-Bang Phase Detector seen previously, but rather than just performing sign detection, the magnitude of the phase difference can also be measured. In the case of the FPGA clocked design the counter handled the representation of both sign and magnitude, but as the name SigNum (Sign Number) suggested both are handled separately in this design. The modified Bang-Bang detector establishes which signal occurred first and generates a control signal that is analogous to the time delay between a rising edge on each input signal. A TDL then uses this control signal to convert the phase difference between the signals to a digital signal. This magnitude is combined with the sign detected by the bang-Bang detector to form a two's complement error signal.

This method of phase detection is advantageous as similarly to the Ring Oscillator, the TDL can be implemented using inverters, and therefore is not synchronised to the FPGA clock and the propagation time through inverters sets the resolution, thus allowing a smaller step than that of the clocked designs. The disadvantages, however, are also consistent with the RO in that the propagation time will vary significantly depending on the EDA's placement of the circuit. This is not as severe a limitation for the TDL as the RO as while the tuning range is affected by this variation it cannot cause the centre point to vary, whereas the impact of placement on the RO caused both variation in range and centre frequency.

3.4.3.1 Sign Detection

The Bang-Bang detector structure changes significantly in order to provide the control signals required for the TDC. The TDC expects a long pulse of logic high for duration of the measurement interval, going low once the interval has ended. The original single flip-flop design maintains a value between rising edges of the reference clock and could not generate the required pulse. Instead two flip-flops are used, each clocked on an input signal with their D inputs connected to logic high. The Q outputs of these registers are connected to an XOR gate which generates a pulse as long as a rising edge has only been seen on one signal. Once a second rising edge occurs the XOR output will be set to 0, thus terminating the pulse. On completion of the measurement interval, the flip-flops must be reset in preparation for the next measurement cycle, and this can be accomplished by ANDing together the Q outputs of each signal, producing a `clear` signal that will go high when both flip-flops have a logic high output.

To determine the sign itself, which clock edge first had a rising edge must be recorded, and this can be done by setting a Set-Reset (SR) latch if the rising edge of the generated signal occurs first and resetting in the case of the reference clock having the first rising edge. Looking at the whole measurement cycle however, it is apparent that all 4 combinations possible of two inputs occur, putting the SR latch into which the output is undefined. Careful design of the system can avoid problems arising from this. Examining the truth table in Table 3.1, the combination to be avoided is the presence of a 1 on both outputs, precisely the situation seen at the end of the measurement interval. In order to avoid the undefined state occurring at the end of the measurement interval, both inputs and outputs to the SR latch can be inverted using a NOT gate. Table 3.1 also illustrates how the $\overline{Q}$ output now behaves as Q did previously, however the end of measurement state where both input registers have been triggered now causes the SR latch to enter the hold state where it maintains the previous values. The sign, $\overline{Q}$ of the SR, can then be loaded into a register using the `clear` signal generated previously, to maintain it until the next measurement completes.

This, however, ignores the potential for metastability which may arise if both signals have a rising edge within an extremely short period of time. While it may in other systems be feasible to dismiss this occurrence as a low probability event, the goal of a PLL is to reduce the time delay between rising edges to zero which increases the possibility of this occurring, which may lead to indeterminate behaviour in other parts of the feedback network. In order to avoid this in an

S	R	Q	$\overline{Q}$
0	0	Q_{n-1}	$\overline{Q}_{n-1}$
0	1	0	1
1	0	1	0
1	1	undefined	

Table 3.1: SR latch truth table

ASIC, an arbitration circuit is placed between the SR latch and the register storing the sign between measurements, which will default to one of the two signals in this case. This is however a mixed signal circuit, such as that proposed by Tierno *et al* (2008) and implemented by Zianbetov and Shan [4, 13, 26], which requires two P-type Metal Oxide Semiconductor (PMOS) and two N-type Metal Oxide Semiconductor (NMOS) transistors and cannot be implemented on an FPGA. As the output of the SR latch will be indeterminate prior to the detection of the first edge, the impact of both edge detection registers' output is that the SR latch attempts to hold this indeterminate value as an output. To avoid this unwanted behaviour the arbiter present on an ASIC is replaced by a second SR latch, connected as in Figure 3.8. This ensures the value presented to the register storing the sign is always determinate.

When	Gen	Ref	$\overline{Gen}/S^a$	$\overline{Ref}/R^a$	Q^a	S^b	R^b	$Sign/Q^b$
Between Measurements	0	0	1	1	undefined	01 or 10		0 or 1
Ref. First	0	1	1	0	1	0	1	0
Gen. First	1	0	0	1	0	1	0	1
Measurement Complete	1	1	0	0	Q^a_{n-1}	$\overline{Q}^a_{n-1}$	Q^a_{n-1}	$\overline{Q}^a_{n-1}$

Table 3.2: Sign detector truth table

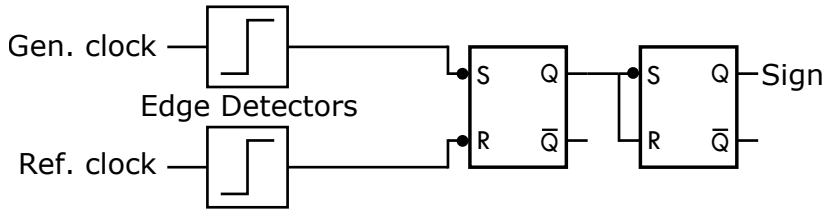


Figure 3.8: Example Sign Detector.

3.4.3.2 Tapped Delay Line

To complement the asynchronous sign detection circuit, an asynchronous TDC is also required, as otherwise there would be no benefit to using the above method over the more easily understood FSM based design as quantisation to the clock period would happen in the TDC rather than the sign detection circuitry. The standard method for implementing a TDC on an ASIC is through a tapped delay line, where a signal is propagated through a chain of fixed length delays, and at the end of the measurement interval it is recorded whether or not the signal has propagated to each point in the chain. This architecture is illustrated in Figure 3.9. As with the mixed signal circuitry previously addressed, a TDL cannot be exactly replicated on an FPGA, but through the use of inverters it is possible to replace the unrealisable well bounded delays. As with the RO, the delay though the inverters is not nearly as well bounded, and significant variation between delays, in both individual TDLs and within the same TDL, due to the implementation occurs This variation notwithstanding, the inverter pairs are a suitable replacement, and this method of time-to-digital conversion provides both resolution advantages and the presence of implementation based variation over a clocked design. These prove particularly beneficial when attempting to verify theoretical results.

Similarly to the sign at the output of the detection circuit, flip-flops store the value of each tap between phase error measurements. Prior to storage, the tap values must be converted to two's complement. As the signal propagates through the taps, magnitude of the phase error is determined by how many taps the signal has passed through. Conceptually, this thermometer coding is first converted to an unsigned integer before conversion to a signed quantity based on the sign of the error.

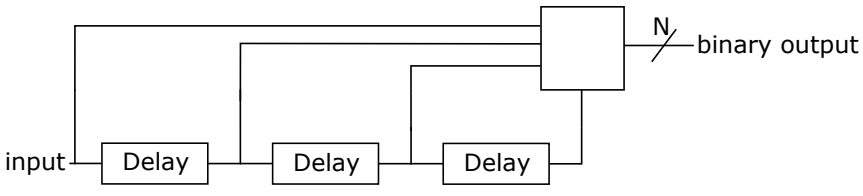


Figure 3.9: Example Tapped Delay Line Structure.

3.5 Error Combiner

In a regular ADPLL an error combining circuit is not required as there is never more than one reference signal, and as such the input of this block would be equivalent to the output. However in an ADPLL network multiple reference signals are required, one for each neighbour, thus requiring some method to combine. Simply averaging these signals was not found to be viable by Pratt and Nyugen in their original paper for analogue systems, however, the work of Javidan *et al*, showed that in a digital system this method will not result in mode locking, provided start-up is performed uni-directionally [27].

Accordingly, any intended error combiner design implementing an average should also be reconfigurable, so that it satisfies the reconfigurability constraint. Given this is a digital system the most straightforward way to implement this is as a weighted sum, followed by a division. The final ADPLL network will be laid out in a Cartesian grid so a maximum of four values will possibly be combined, with only the non-corner edge nodes having a number of neighbours that is not a power of two. The simplest, and most processing time efficient, method of calculating the weighted sum restricts the weights to powers of two, as this permits their implementation as a left bit shift which is trivially achieved in hardware by a part select. Similarly this reasoning can be expanded to the division, where the complicated division by three for a non-corner edge node is instead replaced by division by four. This means the division is implemented identically by all error combiners in the network, and implemented trivially by a right shift of two bits.

3.6 Loop Filter

The Loop Filter in an ADPLL can be implemented as a Proportional Integral (PI) controller, a circuit that can be implemented in a wide variety of ways, with the exact implementation having little impact on behaviour. The first choice to be made is whether the designer wishes to implement the controller as a Finite Impulse Response (FIR) or Infinite Impulse Response (IIR) filter. Both methods will produce a controller that fundamentally behaves the same way, but will have a starkly different implementation. In both cases the filter can be implemented using either fractional or integer representation of the data, depending on the wishes of the designer. Carrying out computations using fractional values may be conceptually easier to understand as the magnitude

of individual paths or taps will be independent of another, whereas using integer representations allows for a simpler implementation. If fractional representation is used, it will be done with fixed-point arithmetic as the time complexity of floating point calculations is too great and the potential benefits are inconsequential. The Loop Filter output may be applied directly to the oscillator, or an intermediate calculation done to set a bias point.

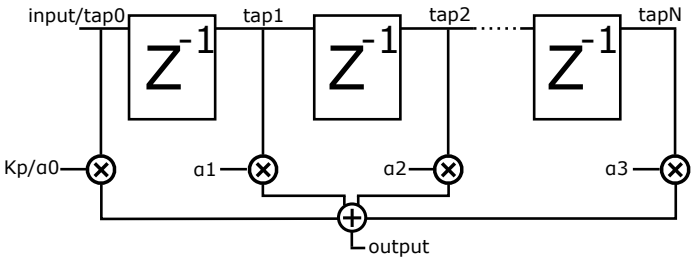
3.6.1 FIR Loop Filter

The Finite Impulse Response implementation of the Loop Filter will take the form presented in Figure 3.10 (a), and consists series of time delays, or "taps", the output of which is multiplied by a coefficient and the result of each multiplier added together to form the filter output. Various tools, Matlab for example, can be used to compute the coefficients and number of taps required to obtain a specific filter response. Using an FIR filter is advantageous as it is always stable due to the lack of a feedback path, however a large number of taps are required to obtain a frequency response akin to that of the IIR implementation, thus requiring an equal number of multipliers and $taps - 1$ adders. Compared to an IIR, which requires just a two adders, two multipliers and a flip-flop this has a much greater hardware footprint. It is possible to condense the FIR multiplication and addition to a single multiplier and adder, depicted in 3.10 (b), however this will take as many clock cycles to complete as there are taps. This, however, is not viable in this system as the filter must produce its output in a single clock cycle. In the case of any coefficient symmetry, addition can be performed prior to multiplication, thus reducing the number of multipliers for each identical coefficient.

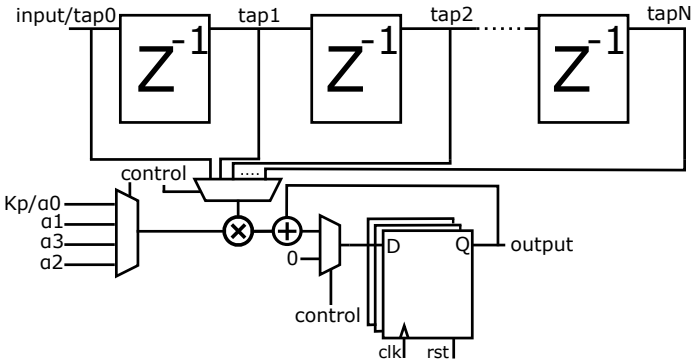
The filter can be implemented using fixed-point arithmetic, in which the designer must ensure alignment of the radix in additions and correct radix position in multiplication results. Consequently the fractional width of the each multiplication need not be the same, most commonly used when the proportional gain, the output of the 0th tap multiplication, is significantly larger than that of the other taps. Integer arithmetic instead assumes all calculations are carried out using integers, and output of the addition stage is divided by a large number to form the module output. The integer approach simplifies the task of the designer and reduces the likelihood of incorrect calculations due to misalignment.

3.6.2 IIR Loop Filter

A PI controller can be easily implemented as an Infinite Impulse Response filter in hardware,



(a) Typical FIR Filter



(b) Single Multiplier FIR Filter

Figure 3.10: FIR PI loop filter RTL diagrams.

as in Figure 3.11, requiring significantly fewer circuit elements than an FIR based design as the long series of taps and multipliers is not required, as instead integration is performed by a single multiplier and an accumulator. As before the frequency response of the controller can be calculated using the following equation, where α is the proportional and β the integral gain:

$$H(z) = \alpha + \beta\frac{1}{1 - z^{-1}} = \frac{(\alpha + \beta) - \alpha z^{-1}}{1 - z^{-1}}$$

As the scaling factor used in this multiplier is the Integral Gain (k_i) itself, reconfiguration of the integral path of the Loop Filter can be carried out solely by varying this gain, as opposed to the FIR solution where the value of each tap must be recalculated to change the response of the system. The combination of reconfigurability and simpler hardware make the IIR the more desirable LF design.

Again both integer and fixed-point methods of LF design are possible, with the integer method requiring logic combining the two paths to preserve the relationship between the proportional and integral gains, which from the work of Koskin *et al* is known to be several binary orders of magnitude [15]. Failure to preserve a relationship of this order may lead to the ADPLL becoming unstable, as in the upper-left region of Figure 3.12. In the fixed-point system the relationship

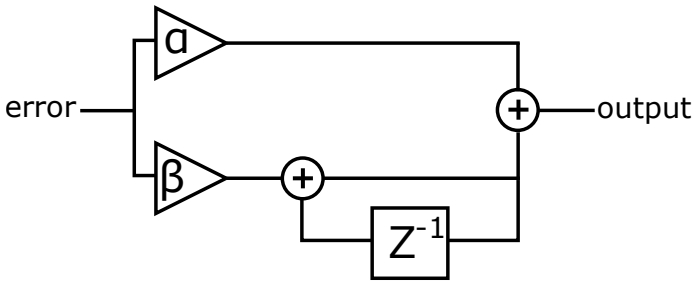


Figure 3.11: Basic IIR PI loop filter.

between Proportional Gain (k_p) and k_i is preserved during the combination of proportional and integral paths as both values have their radix point aligned before addition is carried out.

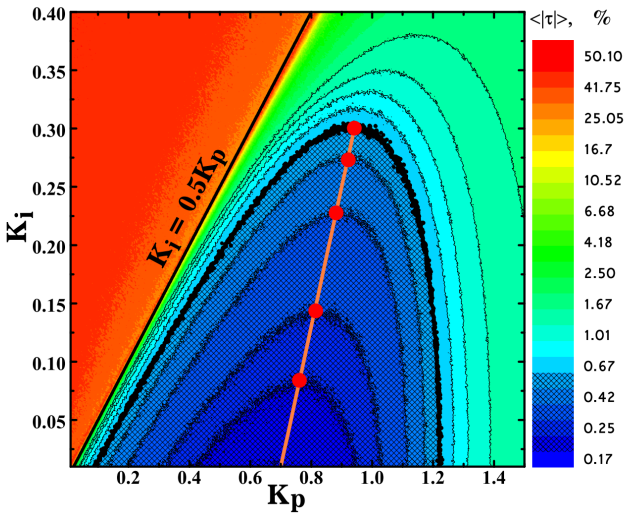


Figure 3.12: Stability of Loop Filter gains [15].

Chapter 4

Network Implementation

4.1 Chapter Overview

This chapter will cover the network implemented in the course of the project, beginning with the individual All-Digital Phase Lock Loop (ADPLL)'s composition. A number of attributes for each design will be used to analyse and compare their performance. The impact of some minor architectural changes will also be covered, before the implementation of test network will be described. This chapter will also highlight some pitfalls encountered in the process. The Hardware Description Language (HDL) Verilog was selected for use in this project due to my prior exposure to that language. 5 MHz was chosen as the target centre frequency for these ADPLLs, during initial testing it was discovered that the waveform output via the Digilent Nexys4 development board's General Purpose Input/Output (GPIO) ports began to degrade around the 10 MHz mark, preventing accurate analysis.

4.2 ADPLL Architectures

Three different architectures of ADPLL were implemented in this project for the purposes of testing, and re-use in the future within the research team in University College Dublin. Between each design a number of blocks remain constant, as only a single instance of that block was implemented, as variation its design would have negligible impact on the final system. These blocks were the error combiner and frequency divider. The loop filter was implemented both using fractional and integer arithmetic, but as the exact same calculation was performed in both cases there was no impact on the output of the module.

The three architectures chosen for implementation represent a progression from entirely a Field Programmable Gate Array (FPGA) clock driven system to an ADPLL with all components asynchronous with respect to that clock. Therefore the first ADPLL design features a clocked oscillator and phase detector, specifically the linear frequency design mentioned in Chapter 3 and the clocked

phase detector using a state machine and a bi-directional counter. Skipping over Design 2, the final design used was asynchronous in totality, featuring the Ring Oscillator and SigNum/Tapped Delay Line (TDL) phase detector. Design 2 sat between these two, using the RO as its oscillator, but retained the clocked phase detector seen in Design 1.

The extensibility of the end product of this project was to the forefront during the design process. Whilst, apart from frequency, there were no guidelines as to what the system should resemble and a number of attributes could be chosen arbitrarily, a future user of this platform could have entirely different requirements or specifications. As such each block was implemented in such a way that changing the number of bits used for a certain signal, the target frequency or swapping between different blocks could be done with the sole requirement of changing Verilog `localparam`[1]s in the module [2] in which change was made, which would then propagate to any sub-modules affected. Starting the design process with this in mind was a major benefit at later stages when frequent modifications were being made to test their impact, or when changing between ADPLL designs.

4.2.1 Generic Components

As mentioned in the chapter overview, a number of components were used in all ADPLLs implemented, as changes in their design would not impact the overall system.

4.2.1.1 Error Combiner

The error combiner in this design had a number of requirements in order to be suitable for use in both a single ADPLL as well as a network operating in both uni- and bi-directional modes, alongside modifications which would change the widths of the error signals requiring averaging. The module was implemented on the basis that the Loop Filter (LF) input width would be identical to the phase detector output width.

The error combiner was designed to take in up to four error signals, all of an identical width which defaults to eight, and four weights to apply in the summation process. A weighted summation is performed by multiplying each error signal by the weight and adding the results together. Regardless of the number of input error signals, a division by four is then performed in order to compute the average error, done by shifting the value right by two. In a HDL, unlike C pro-

[1] A `localparam` is a constant that cannot be modified in a module instance statement [28]

[2] Module is the Verilog term for a number of logic elements grouped to provide a certain functionality

gramming or similar, a shift requires no computation and is instead implemented by performing a part-selection. As all error signals are two's complemented signed integers, the error combination process is also carried out using signed values. The code implementing the error combiner can be found in ERRORCOMBINER.V, attached in Appendix A Listing A.3.

4.2.1.2 Frequency Divider

The frequency divider is implemented using "Divider 2" from Chapter 3, as only divisions ratios of 2, 4 and 8 were selected for use in the testing process. For ease of testing, the divider had outputs representing each of these division ratios simultaneously, which could be multiplexed between at runtime if so desired. A fourth output passed the input through unmodified, allowing for the divider to be removed without regeneration of the FPGA configuration.

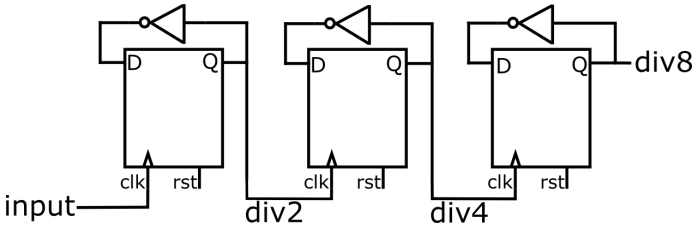


Figure 4.1: Frequency divider RTL diagram.

4.2.1.3 Loop Filter

Two Loop Filters were implemented, both as Infinite Impulse Response (IIR) filters. Finite Impulse Response (FIR) was dismissed due to the extra hardware required to implement the calculations within a single clock cycle, as the LF is clocked using the oscillator output, and the greater difficulty of gain adjustment compared to an IIR. Clocking on using the generated signal ensures that the discrete time integration is only carried out once per phase comparison. A pitfall that may be encountered implementing an ADPLL featuring a divider is not clocking the module on the divided clock, which will result in the integration being carried out at the oscillator output frequency.

Each of integer and fixed-point arithmetic were used to implement a filter respectively, however both designs are interchangeable as they perform the integration identically, given the same proportional and integral gains. All testing was carried out using integer arithmetic, as in Figure 4.2, however the interchangeability will be confirmed later in this chapter, in Section 5.4. Additionally the LF supports variation of both k_i and k_p at runtime, however, if a static value is desired

the runtime variation may be disabled using a `parameter`[3].

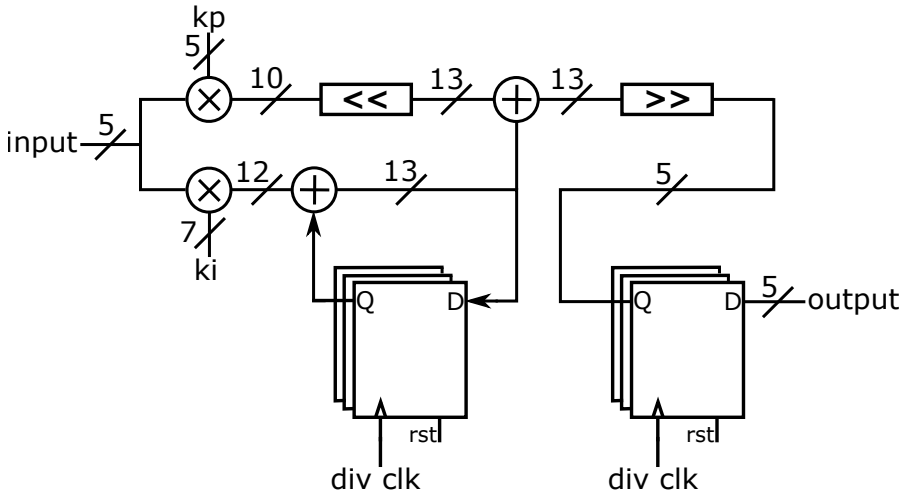


Figure 4.2: Loop Filter implemented using integer arithmetic RTL diagram (signal widths using default values).

Figure 4.2 contains an RTL diagram describing the implementation. Important to note is the shift applied to the result of the multiplication by k_p which, as the integration is performed with a larger width to avoid accumulator overflow, preserves the intended relationship between proportional and integral gains. The value of k_p and k_i can be computed using the following formulae:

$$\text{Proportional gain} = k_p \times 2^{-path\_width\_change\_p} \tag{4.1}$$

$$path\_width\_change\_p = lsh\_width - rsh\_width \tag{4.2}$$

$$lsh\_width = ki\_width - kp\_width \tag{4.3}$$

$$rsh\_width = error\_width + kp\_width + lsh\_width - control\_code\_width \tag{4.4}$$

$$\text{Integral gain} = k_i \times 2^{-path\_width\_change\_i} \tag{4.5}$$

$$path\_width\_change\_i = -rsh\_width \tag{4.6}$$

In the example system shown in Figure 4.2 the proportional gain can be computed as $k_p \times 2^{3-8} = \frac{k_p}{32}$ and the integral gain is $k_i \times 2^{-8} = \frac{k_i}{128}$. The Verilog implementation of the LF using integer arithmetic can be found in LOOPFILTER.V, attached in Appendix A Listing A.4.

[3] Parameters are constant values that may be changed at compile time, or in the module instance statement [29]

4.2.2 ADPLL Design 1

ADPLL Design 1 is comprised of the aforementioned generic components and FPGA clocked implementations of the Phase Frequency Detector (PFD) and Digitally Controlled Oscillator (DCO). This was the first solution addressed in the course of this project as it is the simplest to both implement and test, as all aspects of the ADPLL are driven by the FPGA clock. Compared to later designs utilising the RO, Vivado simulations could be carried out, with accurate timing behaviour, depicting the locking process. The DCO was chosen to be linear in frequency in order to obtain a square wave output as it was not yet known whether the varying pulse width of a period linear design would cause a problem in modules which, at that stage in the project, had not yet been designed. Indeed, a later discarded version of the FPGA clocked PFD relied on equal low and high times. The PFD itself was implemented using a Moore type Finite State Machine (FSM) driving a up-down counter to convert the time between rising edges to a digital signal.

4.2.2.1 Digitally Controlled Oscillator

As the FPGA clocked, linear frequency oscillator is based on the use of counters, the first stage in its design was the decisions as to their required widths. The equation given previously is used here, with the control code set to zero in order to obtain the centre frequency, and 5 MHz as the target DCO frequency:

$$f_{osc} = f_{FPGA} \times \frac{BIAS + CC}{2^{width}} \tag{4.7}$$

$$5 \times 10^6 = f_{FPGA} \times \frac{BIAS}{2^{width}} \tag{4.8}$$

This still leaves three parameters undecided: The FPGA clock, the bit width of the counter and the bias point. As the FPGA clock determines the frequency step of the design, a test counter was implemented initially to determine the maximum frequency at which the timing violations would not occur, at this was determined to be 275 MHz. The equation can then be updated, giving the relationship between bias point and counter width:

$$5 \times 10^6 = f_{FPGA} \times \frac{BIAS}{2^{width}} \tag{4.9}$$

$$\frac{5}{275} = \frac{BIAS}{2^{width}} \tag{4.10}$$

The maximum value of the counter must also be large enough such that with a bias point that allows for a reasonable range of frequency steps to be added or removed. This oscillator was originally designed to allow for 64 control codes either side of the bias, which called for a bias point of 65 or greater.

$$\frac{5}{275} = \frac{65}{2^{width}} \tag{4.11}$$

$$2^{width} = \frac{65 \cdot 275}{5} = 3575 \tag{4.12}$$

$$width = \lceil \log_2 3575 \rceil = 12 \tag{4.13}$$

From the above equation, the smallest counter that can satisfy this constraint is 12 bits wide, however when testing was carried out using a 275 MHz clock timing violations were discovered by Vivado, resulting FPGA clock speed reduction to 258 MHz in order to resolve these violations. The correct bias point could then be calculated as 79:

$$\frac{5}{258} = \frac{BIAS}{2^{12}} \tag{4.14}$$

$$BIAS = \left\lfloor \frac{5 \cdot 2^{12}}{258} \right\rfloor = \left\lfloor \frac{20480}{258} \right\rceil = 79 \tag{4.15}$$

The frequency step can now be computed as all parameters have been chosen:

$$f_{step} = \frac{f_{FPGA}}{2^{12}} = 62.988 \text{ kHz} \tag{4.16}$$

At the target frequency of 5 MHz this corresponds to a change in period of:

$$f_{step0} = 79 \times f_{step} = 79 \times 62.988 \text{ kHz} = 4.976 \text{ MHz} \tag{4.17}$$

$$f_{step1} = 80 \times f_{step} = 80 \times 62.988 \text{ kHz} = 5.039 \text{ MHz} \tag{4.18}$$

$$T_{step} = \frac{1}{f_{step0}} - \frac{1}{f_{step1}} = \frac{1}{4.976 \times 10^6} - \frac{1}{5.039 \times 10^6} \tag{4.19}$$

$$T_{step} = 2.5 \text{ ns} \tag{4.20}$$

From the frequency step, control code range and bias the frequency range of this oscillator can be found:

$$f_{osc} = (BIAS + CC) \times f_{step} = (79 + CC) \times 62.988 \text{ kHz} \tag{4.21}$$

$$f_{min} = (79 + CC_{min}) \times 62.988 \text{ kHz} = (79 - 63) \times 62.988 \text{ kHz} = 1.008 \text{ MHz} \tag{4.22}$$

$$f_{max} = (79 + CC_{max}) \times 62.988 \text{ kHz} = (79 + 63) \times 62.988 \text{ kHz} = 8.944 \text{ MHz} \tag{4.23}$$

Figure 4.3 contains an RTL diagram of this oscillator's implementation. As phase error is a two's complement signed value, its addition to the bias must be carried out using signed arithmetic. Provided the bias is greater than the minimum possible phase error, as has been ensured in this case, the result of this addition is positive signed integer. Being a positive quantity, additional logic performing a conversion from a signed to an unsigned representation is required.

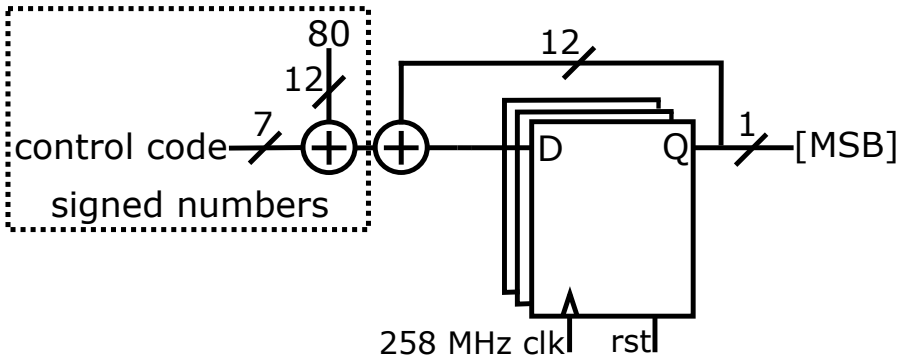


Figure 4.3: DCO RTL diagram as implemented.

4.2.2.2 Phase Detector

As previously mentioned the PFD is also implemented using the FPGA clocked approach, with sign detection carried out using a Moore Machine. The state transition diagram given as the example in Chapter 3 is that used to design this sign detector. The hardware description of this block is found in PHASEDETECTOR.V, attached in Appendix A Listing A.1.

The synchroniser circuits are implemented by a pair of "D" flip-flips connected in series, clocked on the FPGA clock. This synchroniser works by assuming that metastability will only persist for the duration of one FPGA clock cycle as depicted in Figure 4.5. In the case where only one clock signal may experience metastability, two synchronisers are required in order to maintain

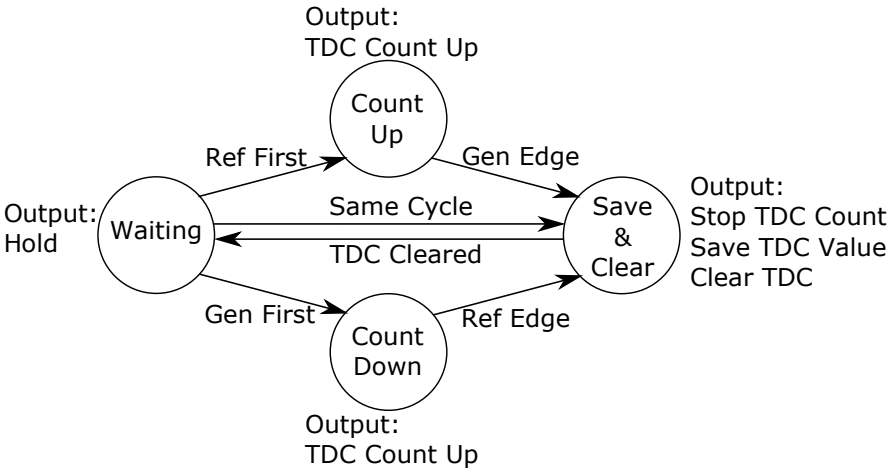


Figure 4.4: Example State Transition Diagram for a Moore Machine.

symmetry in the phase comparison.

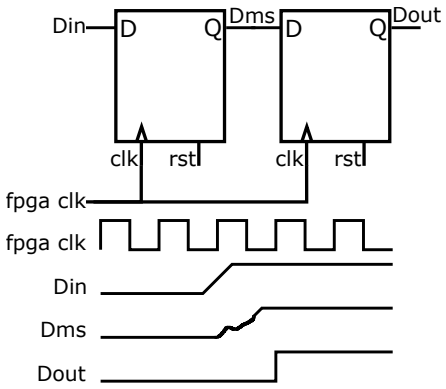


Figure 4.5: Double "D" flip-flip synchroniser circuit RTL diagram.

This FSM then controls an Up-Down counter of the same width as the control code, counting in two's complement. As the control code has already been defined as a 7 bit wide two's complement integer, the phase detector's output can theoretically lie in the range $[-64, 63] \cap \mathbb{Z}$.

As this detector samples each signal at the FPGA clock frequency, the phase detector resolution is equal to the FPGA clock period at:

$$t_{res} = T_{FPGA} = \frac{1}{258 \times 10^6} = 3.875 \text{ ns} \tag{4.24}$$

an angular resolution of:

$$res = \frac{t_{res}}{T_{osc}} \cdot 360° = \frac{3.875 \text{ ns}}{200 \text{ ns}} \cdot 360° = 6.975° \tag{4.25}$$

The counter is implemented using saturation arithmetic to prevent overflow at high phase differences which could potentially harm the ability of the ADPLL to obtain a lock. In the case of this Phase Detector (PD) the phase error at which overflow may occur can be computed using the bit width of the counter and temporal resolution. $error_{max} = 63 \cdot 3.875\ \mathrm{ns} = 244.125\ \mathrm{ns}$. As this is greater than the period it cannot occur for purely a phase difference, but may occur in the presence of either a frequency divider in the feedback path, thus increasing the period of the signal used for comparison, or of a significant frequency difference between the signals. When implementing the clamping logic, the decision was taken to make the detection range symmetrical, thus the output range was reduced by one increment to $[-63, 63] \cap \mathbb{Z}$.

Figure 4.6 depicts the implementation of this counter. Apart from the aforementioned capping of the measurement range implemented using a pair of comparators and a multiplexer, important to note is the bit width of the increment. Despite overflow being disallowed when the number is interpreted as a signed number, the addition of a 7 bit wide -1 to the accumulator value will cause overflow in the adder. This, however, is perfectly valid as an implementation of subtraction in two's complement. On the right side of the diagram the output register can be seen. Clocked using the FSMs measurement interval complete signal, this register preserves the phase error until the termination of the following interval.

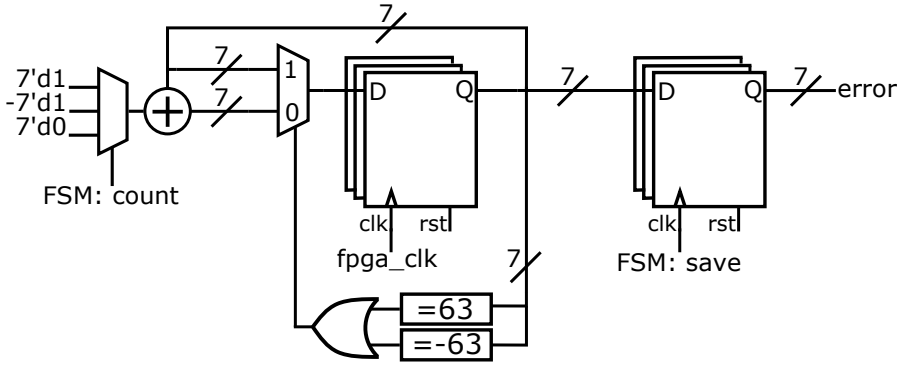


Figure 4.6: RTL diagram of up-down counter.

A major pitfall was encountered in this design of phase detector, which lead in some conditions to a form of mode-locking, in which each oscillator would lock $180°$ phase shifted from the oscillators used as a reference. This was later discovered to be as a result of the measurement interval termination conditions in the FSM, which in addition to those described in Figure 4.4, would terminate the interval if a falling edge was observed on the signal that originally triggered

the measurement. Figure 4.7 will be used to describe the exact circumstances of the error.

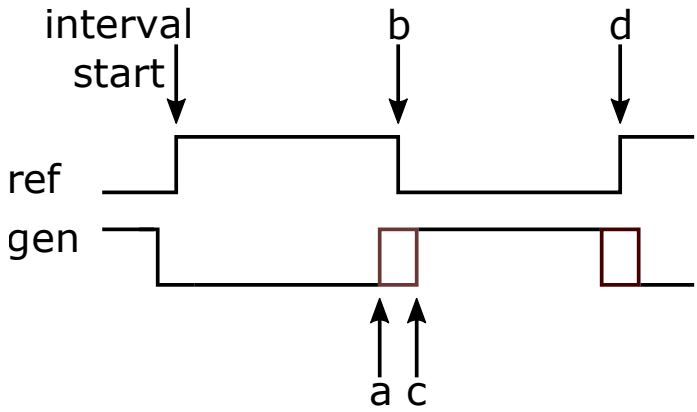


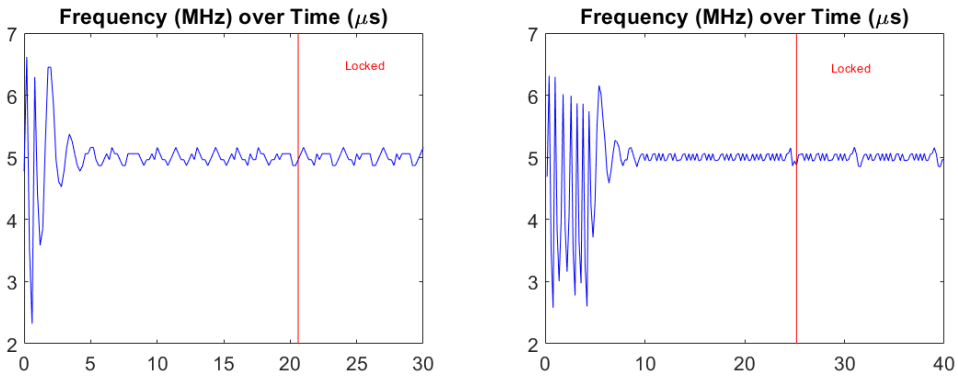
Figure 4.7: Circumstances of mode locking behaviour.

The situation would occur when the measurement interval began with a large phase difference between the generated and reference signals and a simultaneous difference in frequency. Without a frequency difference the measurement beginning at “interval start” would terminate at location “a”, measuring a large lag, with the following measurement interval starting at location “d”. However in the presence of a frequency differential, the rising edge on the generated signal may occur only at location “c”, and with the falling edge termination condition causing the interval to terminate before this edge was detected, at point “b”. This lead to a slightly lower magnitude phase lag detected, however this is a minor flaw, and not the cause of the mode locking. As the rising edge at point “c” occurs after the interval has terminated, it is treated as a fresh measurement interval, terminating at “d”. As in this interval, the generated signal has seen the first rising edge, a large lead will be recorded. As each correction is made with a delay of one cycle, in order to avoid timing violations in the feedback path, there is potential for oscillation between lead and lag detection to occur depending on the initial conditions. This low probability behaviour was observed by chance, due to coding error, while using a frequency divider in the feedback path as this ensured the lead and lag measurements were of identical magnitude.

4.2.2.3 Simulations

As this was the first ADPLL design implemented, various Vivado test benches were used to verify the behaviour of each module. Simulations with this design were made more efficient due to deterministic timing in all modules, and absence of any inverter based components which required the use of post-implementation simulations in order to avoid zero delay oscillations. Deterministic

timing means ADPLL 1 offers an additional, and significant, advantage over other designs as it can be entirely simulated in Vivado with accurate timings. This ability was vital initially, as it revealed the presence of edge situations that had not been accounted for, and through exporting the waveforms to a log file and analysis performed using Matlab comparison made to the measured system behaviour.



(a) Vivado simulation.

(b) Hardware measurement.

Figure 4.8: ADPLL 1 Vivado locking simulations.

In Figure 4.8 (a), the startup behaviour of the system can be observed. This simulation was performed with a reference at exactly 5 MHz and an initial deviation of 252 kHz, and the plot shows the inverse of the DCO period against the time since the oscillator was enabled. After some initial oscillation the period settles around that of the reference. Once locked, cycle-to-cycle jitter was calculated to be 3.491 ns using the standard deviation of the periods, with a worst case cycle-to-cycle change of 9.299 ns. Lock was visually determined to have occurred after 20.91 µs, after which the oscillator frequency began to change in a repetitive pattern. later the same test carried in hardware with identical initial conditions, the results of which are shown in Figure 4.8 (b). This plot is derived from data captured using an Agilent MSO7054A, using a setup that will be further explained in Section 4.3. Despite lacking a non ideal reference, less violent initial oscillation, and a lesser until lock could be visually determined, the simulation provided a good insight into the system's behaviour once implemented on an FPGA.

4.2.2.4 ADPLL 1 Design Summary

The listed components were brought together in NETWORKADPLL.V, attached in Appendix A Listing A.8. Important to note is that each ADPLL only contains two Phase Detectors despite dur-

ing network operation the need for phase comparisons to be made with up to four neighbours. The two "missing" PDs are in fact synthesised as part of the other ADPLLs, with which the comparison would be made. This allows for a 50% reduction in the number of PDs required, while also ensuring identical error is received by both ADPLLs. Each ADPLL performs comparison with the DCO "above" and "left" of them in the Cartesian grid, and a port in the module outputs the results for use by the other ADPLLs involved. Table 4.1 contains a brief overview of the bit widths, tuning ranges and other configuration information for this ADPLL.

DCO Tuning Range

-	Minimum	Step	Maximum
Frequency	1.008 MHz	62.988 kHz	8.944 MHz
Period	992.0 ns	2.5 ns (at 5 MHz)	111.8 ns

Configuration as Implemented

DCO Counter Width	12 bits	DCO Control Width (max)	12 bits
DCO Bias Point	79	Runtime Gains	Enabled
Phase Detector Error Width	7	Error Sum Weight Width	4
k_p width	5	k_i	8
k_p	$2^{-5} \times [0, 15] \cap \mathbb{Z}$	k_i	$2^{-8} \times [0, 15] \cap \mathbb{Z}$
Detection Resolution	3.875 ns	Detection Phase Resolution	6.975° (at 5 MHz)

Table 4.1: ADPLL Design 1 Summary.

4.2.3 ADPLL Design 2

The second ADPLL design implemented during the course of this project was somewhat of a stepping stone between fully synchronous and asynchronous to the FPGA clock. The clocked phase detector is reused from above, albeit with an altered error width, while the DCO is replaced by a Ring Oscillator. While the phase detector was known to work due to testing in ADPLL 1, the DCO could not be simulated using accurate timings so all testing had to be done in hardware. This presented a challenge, as depending on the implementation, the characteristics of the DCO could change, most importantly the centre frequency. The module implementing this ADPLL can be found in NETWORKRINGADPLL.V, attached in Appendix A Listing A.7.

4.2.3.1 Digitally Controlled Oscillator

ADPLL 2 exchanges the FPGA clocked oscillator for RO, with the aim of obtaining performance more akin to that of an Application Specific Integrated Circuit (ASIC) based mixed-signal implementation, due to the variation in period steps both between oscillators and within the same oscillator due to the oscillator layout. Also advantageous is the improvement in period resolution obtained using this design, when compared to that of the FPGA clocked design. Taking the formula given in Chapter 3 and the inverter propagation time used by Vivado in post synthesis/implementation simulations of 315 ps, the period step can be computed:

$$t_{step} = \text{inverters per step} \times \text{inversions per period} \times \text{propagation delay} \tag{4.26}$$

$$t_{step} = \text{inverters per step} \times \text{inversions per period} \times \tau_{inv} \tag{4.27}$$

$$t_{step} = 2 \times 2 \times 315 \text{ ps} = 1.26 \text{ ns} \tag{4.28}$$

Using the propagation delay, τ_{inv}, the number of inverters required to produce the signal of period 200 ns can easily be computed:

$$\text{num\_inverters} = \left\lfloor T_{osc\ @\ 5\ \text{MHz}} \times \frac{1}{2 \times \tau_{inv}} \right\rceil = \left\lfloor \frac{200 \text{ ns}}{0.63 \text{ ns}} \right\rceil = 317 \tag{4.29}$$

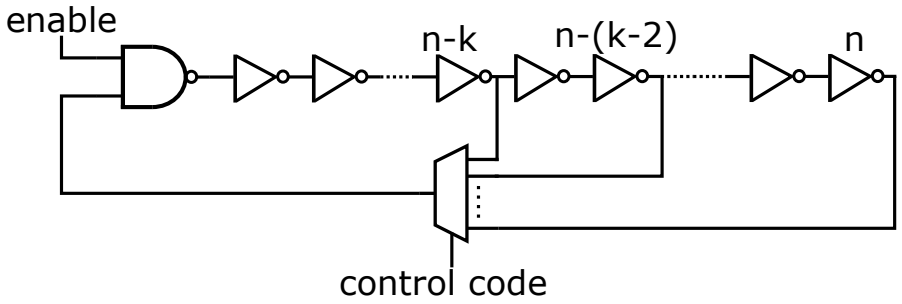


Figure 4.9: Ring Oscillator RTL diagram.

Figure 4.9 contains an RTL diagram of the RO as implemented. The inverter chain is specified with a maximum length of n and a maximum number of removable inverters of k. In order to preserve the unstable circuit, by maintaining an odd number of inverters, only multiples of two inverters may be removed at a time. The already introduced DCO and PFD designs all followed the same convention regarding phase error and the corresponding impact on the control code: In

the PFD, if the generated signal was lagging the reference, meaning it should "go faster" in order to "catch up", the error given a positive sign. Similarly in the FPGA clocked DCO, a positive change in the magnitude of the control code resulted in a higher frequency. In the interests of consistency and extensibility the same behaviour was implemented for this DCO also, with the number of inverters in the chain given as:

$$\text{num inverters} = n - 2 \times CC,\ CC < \frac{k}{2} \tag{4.30}$$

Where CC is the unsigned integer control code input. k is determined by the bit width of the control code which, as in the case of the clocked DCO, is also an unsigned number. An inverter chain will oscillate freely and cannot be disabled. To this end the first inverter in the chain was replaced by a NAND gate to grant control over its operation.

The bias point functions differently in this oscillator, as unlike the FPGA clocked design it is not required to set the centre of the tuning range, as in the RO this lies at $n - \frac{k}{2}$. Instead the bias point servers as a conversion between the signed LF output, which is in two's complement form, and the unsigned requirement of the control code. This is achieved by adding on the midpoint of the unsigned range and storing the result in an unsigned value. Using the case where the width of the control code is three: The range of the signed three bit integer is from -4 (100) to 3 (011) which when directly converted to unsigned values become 4 and 3, thus destroying the relationship between lead and lag. Worse still the signed value of -1 (111) becomes 7. Adding on the mid range value solves this problem, with a signed value of -4 now mapping to 0 and 3 to 7. Comparing this behaviour to response of the RO to the control code, it can be trivially seen that the minimum possible period will still correspond to the largest control code prior to conversion and vice versa for the maximum period.

A control code width of 5 bits was chosen, as with two inverters removed per control code, the tunable range would consist of 20% of the total inverter count. To achieve this range with a centre of 317 inverters, the maximum number of inverters in the chain was required to be $317{+}2{\times}2^{5-1} = 349$,

dropped to a minimum at 285. The corresponding minimum and maximum frequencies then are:

$$f_{osc} = \frac{1}{T_{osc}} = \frac{1}{(n - 2CC) \times 2 \times \tau_{inv}} = \frac{1}{(n - 2CC) \times 2 \times 315 \text{ ps}} \tag{4.31}$$

$$f_{min} = \frac{1}{T_{max}} = \frac{1}{349 \times 2 \times 315 \text{ ps}} = 4.548 \text{ MHz} \tag{4.32}$$

$$f_{max} = \frac{1}{T_{min}} = \frac{1}{285 \times 2 \times 315 \text{ ps}} = 5.569 \text{ MHz} \tag{4.33}$$

The first pitfall with this type of DCO comes in the HDL stage, with the Electronic Design Automation (EDA) likely to optimise away what it sees as a long chain of inverters carrying out no function. This can be avoided by adding an `attribute` as a prefix to the instantiation of a logic element or net that should not be "optimised" away. It is important to choose the correct directive, otherwise the compiler may not behave as expected. In the case of Vivado, it is important not to confuse the `KEEP` or `KEEP_HIERARCHY` commands with that of `DONT_TOUCH`, with former commands only ensuring that the logic elements will be maintained in synthesis, but not in any subsequent stages [30].

As mentioned in previous chapters, the absence of direct control over layout can lead to significant variation of the propagation time though inverters. This may occur in three ways: Firstly, the layout of individual ROs may be significantly different, thus resulting in poor overlap of tuning regions. Anecdotally, while implementing a 3x3 network, 7 of the 9 oscillators had centre frequencies within a 100 kHz span but two lay more than 500 kHz away in opposite directions which prevented the network from locking. Secondly, within an RO the propagation delay between each inverter may vary, which results in a variable frequency step, possibly changing the locking range. These two effects represent an extreme version of the variation due to process or manufacturing seen on ASICs. The final variation is possibly the most frustrating, and occurs when between implementations the EDA, Vivado in the case of this project, changes the layout of the RO. This may occur as a result of a direct change, or as a result of seemingly innocuous changes to unrelated modules. While the final problem can be avoided, once the performance of the RO is satisfactory, by locking down module, the remaining two problems can only be mitigated somewhat.

This is achieved by assigning specific areas of the chip in which that module must lie, although these must be of a size approximately 40% larger than the minimum space required in order to avoid having the router create a complex, delay intensive layout to fit the RO. To achieve greater

consistency, the implementation directive can be modified such that the router will attempt to use the minimum area that does not require complex routing. `congestion_spreadlogic_low` was used for this purpose in this project, although other options may obtain similar results. The other options available in Vivado can be viewed in the *Vivado Design Suite User Guide - Implementation* [31].

With the hardware description of the RO completed, testing of an instance of the RO with a constant control code of zero was attempted, and it was discovered that the estimation of the centre frequency based on the propagation delay was off by 100s of kHz. The addition of 100 inverters was required to restore a centre of 5 MHz. After the RO was integrated into ADPLL 2 the centre frequency was discovered to have changed once more. The minor variation caused by the now variable control code had changed the average propagation delay through the inverters once more, now requiring 373 inverters for 5 MHz.

When, later in the project, additional ROs were implemented to form networks, 373 inverters proved a suitable starting point which consistently delivered DCOs with a centre frequency within 200 kHz of 5 MHz. For the purposes of describing a typical tuning range, the average propagation delay of an RO consisting of 373 inverters with a centre frequency of 5.06 MHz will be used. To avoid the aforementioned issues with constant control codes, the fixed code was achieved by implementing an ADPLL and setting the Proportional Integral (PI) filter gains to zero at runtime before performing a reset.

Using this updated maximum number of inverters the frequency range can be recalculated. Firstly, the new minimum and maximum numbers of inverters are 309 and 373 respectively, with a mid point at 341. The inverter delay and period step are then recomputed based on the centre frequency of 5 MHz as:

$$t_{step} = \frac{T_{osc}}{0.5 \times (n - \frac{k}{2})} = \frac{200\text{ns}}{0.5 \times 341} = 1.176 \text{ ns} \tag{4.34}$$

$$\tau_{inv} = \frac{t_{step}}{2 \times \text{inverters per step}} = \frac{t_{step}}{2 \times 2} = \frac{1.176 \text{ ns}}{4} = 293 \text{ ps} \tag{4.35}$$

The frequency range is then:

$$f_{osc} = \frac{1}{T_{osc}} = \frac{1}{(n - 2CC) \times 2 \times \tau_{inv}} \tag{4.36}$$

$$f_{min} = \frac{1}{T_{max}} = \frac{1}{(373 - 0) \times 2 \times 293 \text{ ps}} = 4.571 \text{ MHz} \tag{4.37}$$

$$f_{max} = \frac{1}{T_{min}} = \frac{1}{(373 - 2 \times 32) \times 2 \times 293 \text{ ps}} = 5.518 \text{ MHz} \tag{4.38}$$

The module implementing the RO can be found in RINGOSC.V, attached in Appendix A Listing A.5. In the module instance statement it can be seen that a second generated output is also present. This second output is taken from a point in the fixed part of the chain, a significant number of inverters prior to the minimum possible due to control code adjustment. This signal is at an identical frequency to that of the generated signal, however will not be immediately affected by a change in the control code, which could in some edge cases change the polarity of the signal. This signal is used to drive the LF in order to avoid an extraneous addition to the accumulator in the integral path. This early signal must be taken from an inverter at an index significantly lower than $n - k$ to prevent this addition causing the problem it has been implemented to avoid.

4.2.3.2 Phase Detector

The PFD used in this oscillator is, apart from the bit width of the counter, identical to that used in the previous design. The reduction in counter width will have no impact of performance of the ADPLL once a lock has been acquired, however, the capture speed will be reduced for large phase differences due to the counter reaching saturation point significantly sooner. 5 bits corresponds to a counter range of $[-15, 15] \cap \mathbb{Z}$, which will saturate at a time delay of $15 \times 3.875\text{ns} = 58.125$ ns, a phase lead or lag of $\times 360° = 104.4°$ at 5 MHz. This is an adequate range, as saturation will occur well outside of the normal operating range. This reduction is most noticeable when a divider is active in the feedback path, however during locked operation a phase shift of 58.125 ns will not occur, so this impact is only felt during acquisition.

4.2.3.3 ADPLL 2 Design Summary

As previously mentioned, ADPLL 2 represents a midway point between FPGA clocked designs and those which are asynchronous. As the source of the asynchronicity is the inverter chain DCO,

a more realistic jitter would be expected from this design, with there being a significant distribution of period steps. Among other aspects, this will have a noticeable impact in steady state conditions, where the degree of fractional synthesis required will vary from oscillator to oscillator depending on the closest distance from the reference period to a period obtainable by the individual oscillator. This is a source of jitter that was not present in the clocked DCO used in ADPLL 1, as all period steps were equal. Once again the DCO tuning ranges and configuration details for future measurements are listed below, in Table 4.2.

DCO Tuning Range

-	Minimum	Step	Maximum
Frequency	4.571 MHz	29.180 kHz (at 5 MHz)	5.518 MHz
Period	218.2 ns	1.176 ns	181.1 ns

Configuration as Implemented

RO num. of inverters (max)	373	RO Control Width	5 bits
RO Bias Point	16	Runtime Gains	Enabled
Phase Detector Error Width	5	Error Sum Weight Width	4
k_p width	4	k_i	6
k_p	$2^{-4} \times [0, 15] \cap \mathbb{Z}$	k_i	$2^{-6} \times [0, 15] \cap \mathbb{Z}$
Detection Resolution	3.875 ns	Detection Phase Resolution	6.975° (at 5 MHz)

Table 4.2: ADPLL Design 2 Summary.

4.2.4 ADPLL Design 3

ADPLL Design 3 has no dependence on the clock provided by the FPGA's clock management utility, as both the RO and the Time-to-Digital Converter (TDC) are designed using inverters, while a Bang-Bang detector replaces the FPGA clock reliant FSM of the two previous designs. This introduces a second source of implementation based variation between ADPLLs, however, this has significantly lesser impact on the frequency range of the oscillator and no impact on the centre frequencies. The lack of an FPGA clocked element results means this design will have the closest behaviour to that of an ASIC based ADPLL implementation, although as the level of variation due to implementation is much larger on an FPGA, the impact of non-idealities will be exaggerated. In order to allow quick exchange of PDs or other modules, the module in which this ADPLL is implemented is still retains the FPGA clock as an input. In fact, this module shares its implementation with that of ADPLL 2, as while the PD designs are drastically different, the interface is identical.

The module implementing this ADPLL can be found in NETWORKRINGADPLL.V, attached in Appendix A Listing A.7.

4.2.4.1 Digitally Controlled Oscillator

The DCO used in ADPLL 3 is identical in form to that used in ADPLL 2, so the tuning range, period step and all effects of implementation variations carry over to this ADPLL. The decision to leave the RO unmodified is an easy one, as the intended centre frequency is the same and changing the control code width would remove the ability to fix the cells of each RO. Fixing the cells is beneficial as it will maintain the oscillator layout between implementation "runs", and thus allow for a better comparison between designs 1 & 2.

4.2.4.2 Phase Detector

Instead of an FPGA clocked phase detector, ADPLL 3 uses the modified Bang-Bang detector and TDL approach to phase comparison seen in Section 3.4.3. Sign detection is implemented exactly as described in that section, with the double Set-Reset (SR) latch approach used to determine the sign of the phase error. A flip-flop is inserted after the second SR latch, clocked on the signal `done`, which holds the output value and ensures that the sign changes at the same time as the magnitude. The previous discussion of this phase detector proposed no implementation of the edge detectors, which are easily implemented using flip-flops clocked on the signal of interest, with a constant 1 on the D input. In this form the edge detector is single use only, as subsequent edges will not cause any change in the flip-flop output. This is important as it ensures the measurement interval will only terminate when a rising edge is seen on the signal that did not start the measurement interval, thus avoiding the mode locking seen in the original implementation of the FPGA clocked PD. Once the measurement interval has completed, the `done` signal is used to reset the edge detectors asynchronously. It is possible that an edge may occur within the time taken to reset the counters, and thus the detector incorrectly gauge the start of a measurement interval, however this can only occur in exceptional circumstances in the acquisition period. Therefore it is possible to ignore this scenario, as the only impact will be an increased time taken to acquire a lock.

A two signal interface controls the TDC, a `count` that enables the TDC and a `done` signal that signals the termination of the interval. A single logic gate is required to generate each signal, as ORing the two edge detector outputs will produce logic 1 when the interval has begun while

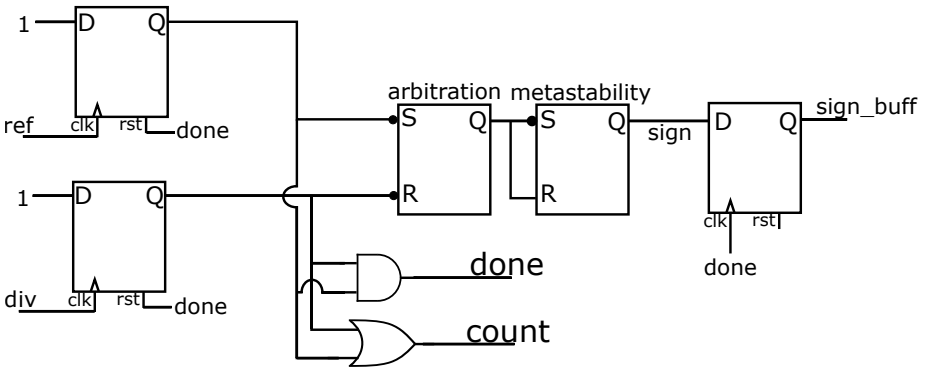


Figure 4.10: Modified Bang-Bang sign detector RTL diagrams.

ANDing these signals will indicate the end of the interval when both edges have occurred. The `done` signal is also used to clock the flip-flop acting as a buffer for the sign. The exact implementation of the sign detection circuit can be seen in Figure 4.10.

As previously mentioned, the time-to-digital conversion is carried out by a tapped delay line. As with the RO, inverters will be used to provide the delay between each tap, and thus the resolution of the TDL and the length of each individual delay is based on the layout chosen by the router in Vivado. Due to this, the exact resolution of the phase detector is unknown, although the delay through inverters used in Vivado simulations and the average delay computed based on RO centre frequencies can be used to estimate that of the phase detectors. Based these figures for the the propagation delay due to an inverter, τ_{inv} can be estimated to be in the region of 300 ps. As each tap consists of an inverter pair, the delay through the tap, $\tau_{tap} \approx 600$ ps. Characterisation of the TDL, and thus measurement of the delays in each TDL, is technically possible, however, this is an time consuming process as many TDLs would need to be characterised in order to accurately compute the average delay.

The delay line itself is shown in Figure 4.11, with each tap implemented by a pair of flip-flops and three inverters. For an N bit width phase error detection, $2^{N-1} - 1$ taps are required. In the case of this ADPLL, a 5 bit two's complement error signal is required, thus the range of the signal is $[-16, 15] \cap \mathbb{Z}$. If the range is made symmetrical, 15 taps are required to fill a signal of this size. From the diagram it can be seen that unless edges are detected on each signal at exactly the same instant, it is almost impossible to measure a zero phase difference between reference and generated signals. This is an intentional decision, [**?**]. The operation of the TDL is as follows: A delayed version of the `count` signal acts as the clock for flip-flops forming each tap, which are rising edge

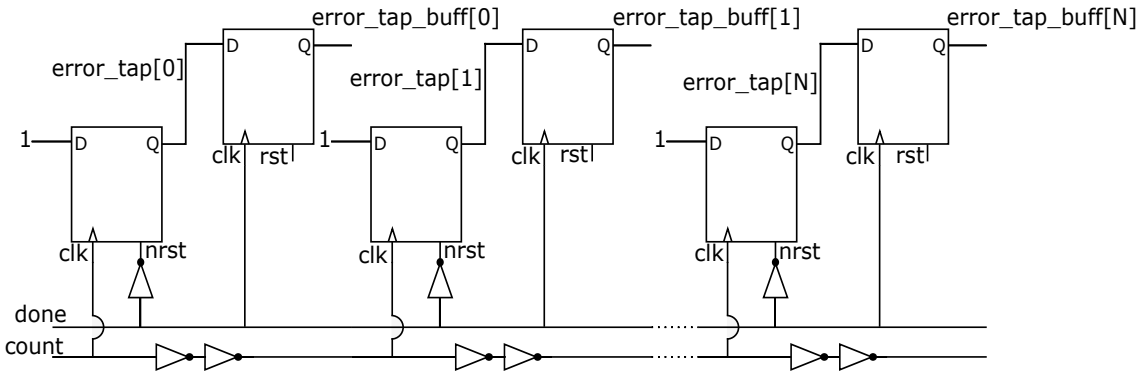


Figure 4.11: Inverter based TDL RTL diagrams.

activated. These flip-flops have a constant "1" at their D inputs, so as the signal propagates through the delay line, each flip-flop with propagate a logic "1" to their corresponding `error_tap[i]` signal. When the measurement interval has completed, the `count` signal clocks the 15 bit wide `error_tap` signal into `error_tap_buff` which act as the output buffers of the PD, holding the phase error constant between measurement intervals. As with the flip-flops performing edge detection, the flip-flops comprising the taps can only be used once before requiring a reset, which is again carried out using the `done` signal. As flip-flops have a hold time, the duration after the clock edge in which the signal applied at the D input must not change to ensure valid data, an inverter is used to delay the reset to avoid this issue.

The output of the TDL itself is not a binary signal, but rather the thermometer coded representation of the number of taps activated in the measurement interval. In order to convert to binary, the easiest solution is a switch statement, with an assignment performed based on the temperature code. However all ADPLL blocks have been designed to be extensible, requiring the lookup to be generated at compile time. This is somewhat difficult to implement, and a simpler solution exists for this problem. In a `generate` statement, a variable size loop is used to count the number of nonzero bits as it would be done in C. At compile time this loop is converted into lookup tables, similarly to the switch statement.

The now unsigned binary representation of the error magnitude can be combined then with the sign bit to form a two's complement error signal. The file PHASEDETECTORDL.V in Appendix A Listing A.2 contains the Verilog implementation of this module. `DONT_TOUCH` compiler directives were used here in a number of places to ensure that the inverters and other elements that are key for timing are not removed in the synthesis process, as similarly to RO, Vivado sees these

as unneeded elements serving no purpose.

4.2.4.3 ADPLL 3 Design Summary

Compared to the oscillators suggested earlier, ADPLL 3 achieves a significantly finer detection resolution, at the expense of significant variation in the detection step. This can be viewed as advantageous however, as the ADPLL exhibits an exaggerated version of the implementation and manufacturing defect based variability present in an ASIC. It is this type of ADPLL that both the UCD and Sorbonne research teams are using to examine potential designs and test theoretical results. As before, Table 4.3 presents the important numeric characteristics of this ADPLL.

DCO Tuning Range

-	Minimum	Step	Maximum
Frequency	4.571 MHz	29.180 kHz (at 5 MHz)	5.518 MHz
Period	218.2 ns	1.176 ns	181.1 ns

Configuration as Implemented

RO num. of inverters (max)	373	RO Control Width	5 bits
RO Bias Point	16	Runtime Gains	Enabled
Phase Detector Error Width	5	Error Sum Weight Width	4
k_p width	4	k_i	6
k_p	$2^{-4} \times [0, 15] \cap \mathbb{Z}$	k_i	$2^{-6} \times [0, 15] \cap \mathbb{Z}$
Detection Resolution	≈ 0.63 ns	Detection Phase Resolution	$1.134°$ (at 5 MHz)

Table 4.3: ADPLL Design 3 Summary.

4.3 Measurement Setup

For the following sections in which the performance of either individual ADPLLs or ADPLL networks the same measurement setup was used. The FPGA used was a Xilinx Artix-7 XC7A100T-1CSG324C on a Digilent Nexys4 board. The XC7A100T-1CSG324C features 15850 logic slices, each with 4 lookup tables and 8 flip-flops. There are six clock management tiles, 240 dedicated digital signal processing slices and an analog-to-digital converter. The maximum operating frequency is 464 MHz and 210 input/output ports are available for use. Further information can be found in the datasheet [32]. The Nexys4 is an evaluation/development board for the aforementioned FPGA, however directed at the education sector. It contains useful peripherals such as 16

switches, 8 7-segment displays and 4 user addressable Peripheral Module interface (Pmod) headers and can be used with the free of charge "Webpack" version of Vivado [33].

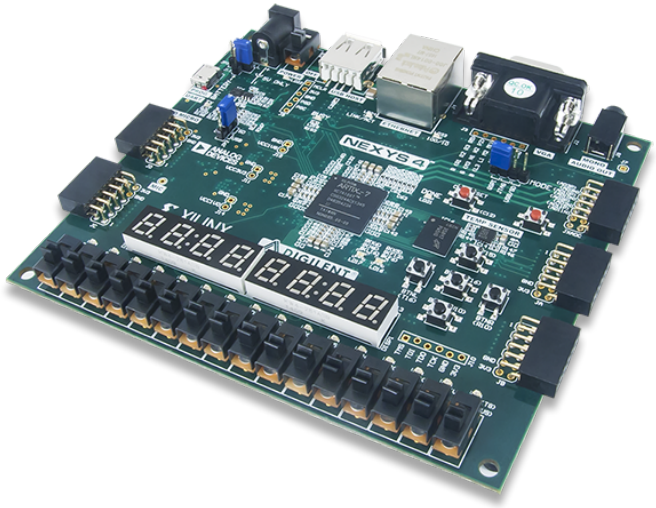


Figure 4.12: Nexys4 development board.

Measurements were performed using an Agilent MSO7054A, a four channel, $4\ \mathrm{GSa/s}$ oscilloscope. Signals were extracted over the regular Pmod headers as the board does not provide outputs better suited to the measurement of signals that are available to the user, such as SMA or BNC connectors, and the Pmod header for the analog-to-digital converter, which has lower impedance traces, is not available for the designer to re-configure. Data was acquired with the oscilloscope in quad channel mode at a sampling rate of $2\mathrm{GSa/s}$. One of the four channels was consumed by the external reference, leaving three free for signal measurements, in most cases the generated output of three ADPLLs were connected. $200\mu\mathrm{s}$ long captures were saved to an Agilent specified binary format which could be opened in Matlab for analysis.

4.4 ADPLL Characterisation

Before combining the ADPLLs in a network it was important to analyse the designs locked to an ideal reference, in order to establish a baseline performance. These tests were carried out with the ADPLLs locked to an external reference at $5\ \mathrm{MHz}$. As ADPLL 2 & 3 have significant variance in the step sizes, for these oscillators range and step based measurements were obtained by measuring a number of oscillators, and typical values will be shown. Jitter was calculated based on the performance on nine individual ADPLLs all locked simultaneously to the $5\ \mathrm{MHz}$ external reference.

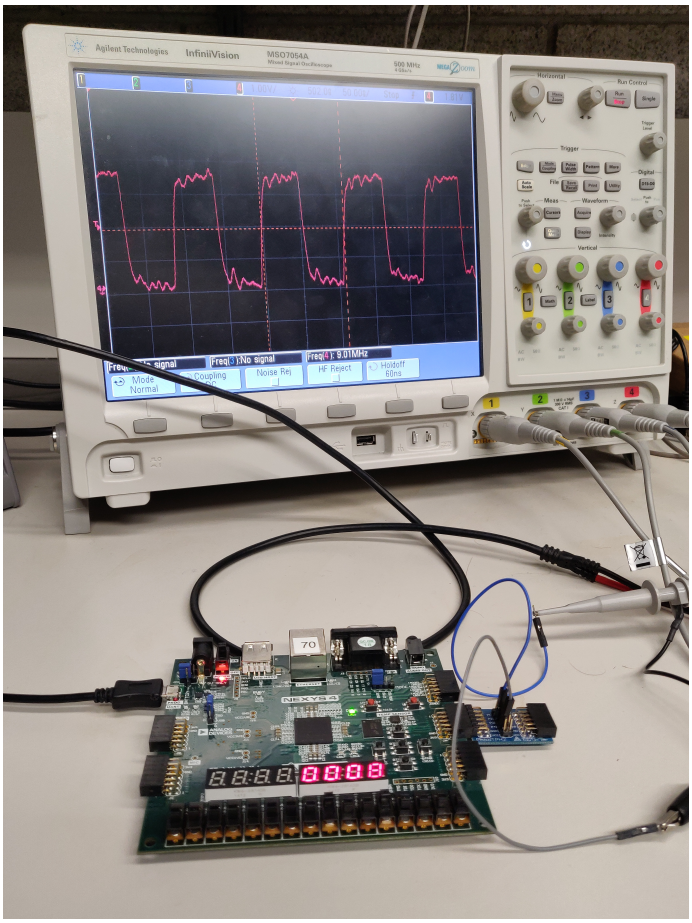


Figure 4.13: Measurement setup.

While jitter and skew are the key indicators of ADPLL network performance, when it comes to the individual ADPLLs themselves it is important to check lock and capture ranges also, to check that the implemented designs match with their intended values. The lock and capture ranges are given based on typical values, whereas jitter and skew are the median values from the ADPLLs tested.

The 4.4 contains the results of this analysis. The first thing to note is that to the kHz the lock and capture ranges of the ADPLLs implementing ROs are identical. As the lock range would be expected to be slightly wider than the capture range in a design limited by the loop filter, it can be deduced that the limiting factor in the tuning range of the ADPLL is the DCO frequency range. This is confirmed by a cursory comparison against the values in Table 4.2, where the RO range is given as $4.571 - 5.518$ MHz. This range of 0.947 MHz is almost identical to those of the ADPLL 1 & 3 which confirms the RO is the limiting factor. In ADPLL 1 however, the lock and capture ranges show some difference, with the lock range being slightly wider. The lower end of both ranges is identical, as rather than the instant loss of lock seen at all other thresholds, the low end of

	ADPLL Design 1	ADPLL Design 2	ADPLL Design 3
Capture Range	$1.08 - 8.61$ MHz	$4.869 - 5.877$ MHz	$4.377 - 5.363$ MHz
Lock Range	$1.08 - 8.635$ MHz	As above	As above
Jitter	1.8385 ns	0.5623 ns	0.7410 ns
Skew (Mea TIE to Ref.)	1.7439 ns	0.3741 ns	0.6381 ns
Maximum TIE	7.2840 ns	6.4770 ns	6.4578 ns
k_p	$\frac{1}{32}$	$\frac{1}{16}$	$\frac{1}{16}$
k_i	$\frac{1}{256}$	$\frac{1}{64}$	$\frac{1}{64}$

Table 4.4: ADPLL Design Comparison.

both lock and capture ranges saw a more gradual degradation, and 1.08 MHz is the point that the lock was observed to be lost.

Looking then at the jitter and skew data, unsurprisingly the entirely FPGA clocked ADPLL with its larger period step is has the worst jitter, by a significant amount. The maximum TIE is also correspondingly worse, which unless significant skew was present in the other ADPLL designs is also to be expected. ADPLL 2 & 3 experience low jitter, at less than 1% of the period. Somewhat surprisingly there is no apparent benefit in terms of jitter reduction due to the better delay resolution of the inverter based TDL. This may be down to either the increase in jitter from a second source of variation, or a less favourable fractional-N synthesis due to the distribution of periods.

Despite Vivado being forced to use the same layout for both ADPLL 2 & 3 through the use of fixed cells, the minimum and maximum frequencies are still significantly different, although the size of the range is almost identical. This highlights the difficulty of ensuring identical conditions between measurements which is the most significant downside of inverter based modules.

4.5 ADPLL Network Implementation

As the ADPLLs were designed with network usage in mind, the HDL aspect of their integration into a network is straightforward. For ADPLL 1, the oscillators need only be wired together, but for Design 2 & 3 which are reliant on inverters require some care to ensure their lock ranges overlap sufficiently. This is a time consuming task, as a number of iterations may be required, even when locking down the cells of lined up ROs, as the nets forming the connections may change. It is recommended only to carry out alignment when all other aspects of the design are unlikely to change and to save the bitstream generated. The projected time requirement in aligning ROs meant

that the creation of a 4x4 network was abandoned. FPGA clocked ADPLLs are more suited to larger networks as their centre point can be determined with ease.

The network operates using an external reference which is input over a Pmod header that is shared with a number of output signals, however it could be configured to use an internal reference generated by either the FPGA clock management utility or a DCO of the designer's choice.

In order to increase the flexibility of the network, the switches on the Nexys4 are used to change both the proportional and integral gains and the configuration of the network at runtime. The reconfigurability aspect of the error combiner allows for the network to operate in either uni- or bi-directional mode, or as a number of individual Phase Lock Loops (PLLs) all connected to the external reference. This is particularly advantageous for the inverter based designs, as multiple combinations can be measured without reimplementing, and thus changing the layout of either oscillator or Phase Detector.

Two different network sizes were implemented, a 2x2 and a 3x3 grid respectively. The 2x2 network was implemented first, and then this design extended to a 3x3 once measurements had been carried out on and any errors made had been corrected. In later sections matrix indexing will be used to refer to each oscillator, with $\text{ADPLL}_{1,1}$ closest to the reference, and higher numbers in either index getting progressively further from the reference, as in Figure 4.14. A 3x3 network was the minimum goal, with the aforementioned but eventually not implemented 4x4 a stretch-goal if time permitted. A 3x3 network set as the primary goal, as it is the minimum size network in which there exists an ADPLL with four neighbours. As will be explained in Chapter 5, ADPLL Design 1 was only implemented in a 2x2 configuration while both other ADPLLs were tested in a 3x3 configuration.

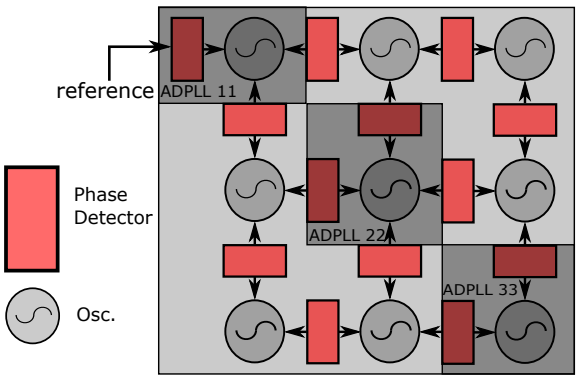


Figure 4.14: ADPLL indexing explanation.

Chapter 5

Testing and Analysis

5.1 Chapter Overview

This chapter contains details of the measurements carried out on networks of various sizes and investigation into the impact of a number of decisions made in the design process. Explanations will be given or analysis made of the results, which will be given in the form of of either plots or tables, depending on which allows for more concise presentation of the data. Both 2x2 and 3x3 network performance analysis was performed with no input delay to the Loop Filter (LF), which Section 5.4.4 will show to have a significant performance impact. All tests were carried out, unless otherwise stated, at a centre frequency of 5 MHz.

5.2 2x2 Network Performance Comparison

In order to investigate the impact of the ADPLL network on performance, and conversely the impact of each individual ADPLL design on the network, switches were used to alter the configuration of the network without requiring re-implementation and a number of captures made at each setting. The three configurations were: each ADPLL individually locked to the reference, the network in uni-directional mode and bi-directional mode. The centre frequency or gains were not changed between measurements in order to obtain a good comparison. For ADPLL 1 the gains were: $k_p = \frac{1}{32}$ & $k_i = \frac{1}{256}$. For ADPLL 2 and 3 they were: $k_p = \frac{1}{16}$ & $k_i = \frac{1}{64}$. The

There are two main trends worth highlighting here, with the first of these being as the feedback network becomes more complicated a large degree of skew is introduced, with the magnitude becoming larger in the ADPLLs further from the reference. This would appear to be attributable to a combination of the time taken for a change in the reference to propagate to the ADPLL node in question, and the fineness of period and detection resolutions. This second observation follows from the significant reduction in the skew growth in Designs 2 & 3 which have finer resolutions.

The other trend, is the variation in Cycle-to-Cycle (C2C) jitter across the ADPLL designs, with

coarseness of the Field Programmable Gate Array (FPGA) clocked Digitally Controlled Oscillator (DCO) becoming somewhat obvious. As free Phase Lock Loops (PLLs), the clocked DCO has almost identical C2C jitter and maximum Time Interval Error (TIE), implying that only two period steps are used the majority of the time. As these steps are somewhat further apart than that of the Ring Oscillator (RO), at 3.875 ns apart as opposed to an implementation dependant value in the region of 1.1765 ns, a much greater jitter is observed. This would confirm the belief that the FPGA clocked designs are better suited to operational frequencies where the generated clock period to FPGA clock period ratio is significantly lower than the $\frac{3.875 \text{ ns}}{200 \text{ ns}} = \frac{1}{51.6}$ in this design. Comparing the C2C jitter of ADPLL 1 & 2, where the main difference is the choice of Phase Frequency Detector (PFD), the approximate $\frac{1.6 \text{ ns}}{0.50 \text{ ns}} \approx 3.2$ ratio of observed jitters is almost identical to the $\frac{3.875 \text{ ns}}{1.1765 \text{ ns}} \approx 3.29$ ratio of the period steps at 5 MHz, implying the PFD is at fault for this difference.

The other notable difference in C2C jitter, is that the use of an inverter based Time-to-Digital Converter (TDC), and its accompanying variation due to implementation differences, has the expected result of additional C2C jitter, however, this impact is signfiicantly less than the that of changes in the RO period step, and the inverter based TDC in combination with the FPGA clocked DCO would still suffer the same problems as it does in ADPLL Design 1 with the clocked PFD.

	Jitter Standard Deviation (ns)			Max. Time Interval Error (ns)			Skew (ns)		
	PLL 11	PLL 12	PLL 22	PLL 11	PLL 12	PLL 22	PLL 11	PLL 12	PLL 22
ADPLL Design 1	-	-	-	-	-	-	-	-	-
Free PLLs	1.6051	1.6146	1.6251	1.6985	1.5833	1.2712	0.2700	0.5817	0.2491
Uni-dir.	1.7693	1.8463	1.8862	7.3060	7.2565	10.151	2.2791	1.2822	1.6299
Bi-dir.	1.9964	1.9146	1.9572	15.089	18.283	19.673	9.2449	11.612	12.052
ADPLL Design 2	-	-	-	-	-	-	-	-	-
Free PLLs	0.56222	0.49224	0.64124	5.3039	7.8626	5.4634			
Uni-dir.	0.55040	0.51284	0.62796	5.5312	7.5359	7.9697			
Bi-dir.	0.57056	0.50403	0.61644	11.515	15.943	17.428			
ADPLL Design 3	-	-	-	-	-	-	-	-	-
Free PLLs	0.66757	0.84477	0.71308	5.9142	6.1353	5.3939	-0.36997	-0.20981	-1.3344
Uni-dir.	0.66934	0.83894	0.71271	5.8779	7.1746	6.8914	-0.36645	-0.05647	-0.36178
Bi-dir.	0.68372	0.83193	0.68570	7.5705	9.6292	9.7715	0.68901	1.58720	1.5553

Table 5.1: 2x2 Network performance Comparison.

5.3 3x3 Network Performance Comparison

Subsequent to tests carried out using a 2x2 network, the corresponding tests were carried out using a 3x3 network in order to analyse the impact of a larger network. For these tests the gains were left unchanged but, as the 2x2 and 3x3 networks have different floorplans, the ROs have changed somewhat. This has especially manifested itself in the exceptional individual PLL performance of Design 3.

From analysis of the 2x2 network, it was evident that the best performing design was going to be ADPLL 3, as both ADPLL 2 & 3 contain the clocked PFD which appears to cause significant skew as the ADPLL node indices increase and move further from the external reference. The increased network size appears to have made the lag between reference edge and the rising edges of the individual generated signals worse, which would be expected as the number of clock cycles required to respond to a change in the phase relationship between reference and the furthest node has increased. ADPLL 3 suffers least from the greater complication of the network, likely as a result of its period and detector resolution being the finest of the three designs considered.

As ADPLL Design 1 suffered significantly more with skew as the node indices increased, so it is no surprise that in the larger network the problem has been amplified, meaning this design is not well suited for use in larger networks at this frequency due to poor resolution. This does not, however, impact the main role of this design method, which is the emulation of designs intended for use in Application Specific Integrated Circuits (ASICs) as the frequency of operation will be much lower than the $5\ \mathrm{MHz}$ used in these tests, which will bring with it a finer resolution.

5.4 Minor Variations

A number of minor design decisions will be tested in this section against their alternatives, as will the impact of varying the LF gains.

5.4.1 Impact of Gain Variation

The gains chosen for use in the Loop Filter have significant impact on the performance of the system, as the LF governs the response of the ADPLL to a given phase error. To large a proportional gain will result in overly aggressive overcompensation, thus negatively affecting the jitter. Even-

	Jitter Standard Deviation (ns)			Max. Time Interval Error (ns)			Skew (ns)		
	PLL 11	PLL 12	PLL 22	PLL 11	PLL 12	PLL 22	PLL 11	PLL 12	PLL 22
ADPLL Design 1	-	-	-	-	-	-	-	-	-
Free PLLs									
Uni-dir.									
Bi-dir.									
ADPLL Design 2	-	-	-	-	-	-	-	-	-
Free PLLs	0.47167	0.55012	0.36251	5.9649	6.6965	5.5010	3.4045	3.2369	2.3108
Uni-dir.	0.49292	0.55586	0.39054	5.958	8.9250	11.622	3.5052	6.9860	8.874
Bi-dir.	0.43263	0.48053	0.33778	18.867	35.973	38.773	15.193	28.965	30.841
ADPLL Design 3	-	-	-	-	-	-	-	-	-
Free PLLs	0.89042	0.64889	0.57405	4.0634	3.2467	2.3141	1.8373	1.1006	0.39334
Uni-dir.	0.93580	0.93353	0.71361	4.2675	5.8193	6.4139	1.8605	2.3172	3.1463
Bi-dir.	0.97717	0.85131	0.63863	5.6439	8.4856	9.4467	2.7129	4.1074	5.1717

Table 5.2: 3x3 Network performance Comparison.

tually the violence of the jitter will result in a loss of lock. However, if the gain is too small, the frequency or period adjustment will not be sufficiently large to cause the ADPLL to lock. Similarly, appropriate values of k_i are required, and the relationship between proportional and integral gains must satisfy the conditions set out by Koskin [15].

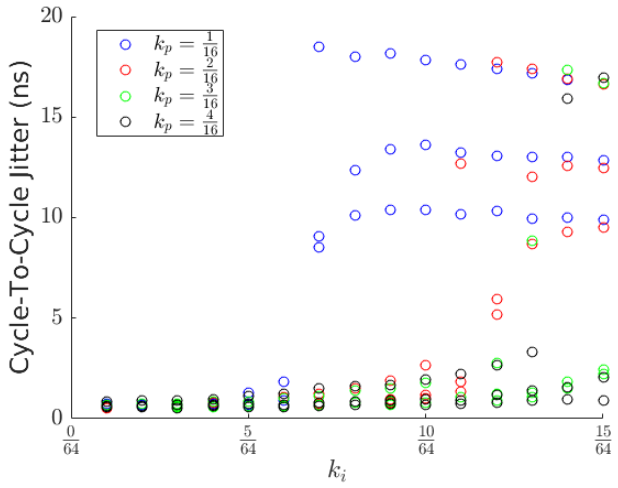


Figure 5.1: Fixed k_p integral gain sweeps.

Figure 5.1 contains a sweep across the possible range of integral gains for four different proportional gains. This test was carried out with a 3x3 network of ADPLL Design 3, each locked to the external reference at 5 MHz, and the generated waveform from the ADPLLs at locations "11", "22" and "33" were captured. It can clearly be observed that there are two distinct regions in the graph, with jitter values of less than 1 ns contrasting sharply with the points with jitter in the region

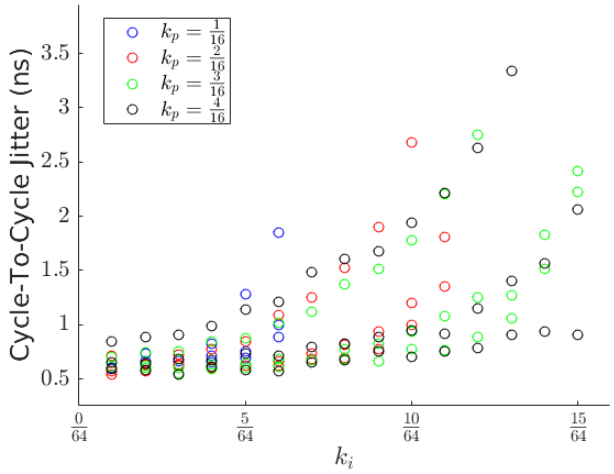


Figure 5.2: Fixed k_p integral gain sweeps.

of 10 ns. These distinct sections of the plot represent locked and unlocked operation respectively. Zooming in on the low jitter region in Figure 5.2, the loss of lock thresholds can be compared with the theoretical relationship of $k_i = 0.5 \times k_p$.

k_p	k_i	Ratio
$\frac{1}{16}$	$\frac{5}{64}$	0.8
$\frac{2}{16}$	$\frac{10}{64}$	0.8
$\frac{3}{16}$	$\frac{12}{64}$	1
$\frac{4}{16}$	$\frac{14}{64}$	1.1428

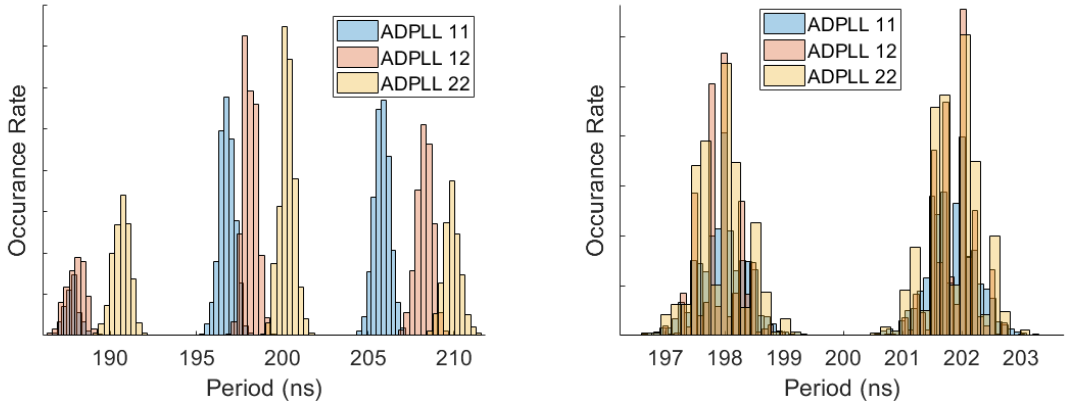
Table 5.3: Loss of Lock Gains.

Despite the relatively small number of proportional gains, it is clear that the relationship set out by Koskin does not hold in this system, but rather a scaled version of it. This difference is attributable to a difference in LF structure. In his simulations the LF design contains a register forming an input buffer, absent in the LF used for these tests. The presence of this register, clocked on the generated signal, causes the ADPLL to react more slowly to changes in the phase difference, thus increasing the jitter. The removal of this register causes a change in the relationship between proportional and integral gains, a change that will be investigated further in a later section.

5.4.2 Distribution of Period Steps

To add context to the jitter numbers produced for each oscillator previously, the variation of period steps in the RO, and delays in the Tapped Delay Line (TDL) due to implementation should be

examined. The best way to visualise this distribution is through a histogram, placing periods of in a certain range into the same bin. This can be visualised effectively using Matlab's `histplot`. In order to obtain data, three ADPLLs implementing both the FPGA clocked and inverter based DCOs were locked to an external reference signal at 5 MHz, and the period of each clock cycle measured. Figure 5.3 contains the output of `histfit` for each of the three oscillators superimposed



(a) Inverter propagation delay based RO.

(b) FPGA clocked DCO.

Figure 5.3: Example distribution of periods.

in one plot for each DCO type. As the RO has significantly finer period resolution, the period steps would be overlaid on each-other thereby making the plot difficult to read. In order to avoid this behaviour, the RO's mapping of control code to inverters to remove was altered from twice the control code to 16 times the control code. This results in a more pronounced separation of period steps while being indicative of the variation due to implementation. The plots paint a a picture of the unrealistic nature of the FPGA clocked design in comparison to its inverter based counterpart, as each oscillator will have identical intrinsic period steps thus eliminating any jitter that maybe be seen in the system due to misalignment. This effects the suitability of the FPGA clocked DCO for the improvement of simulation models, or the examination of how a novel block may impact the behaviour of the network. If this analysis was carried out for the period steps, similar behaviour would be observed.

5.4.3 FPGA Clocked DCO Width Variation

When introducing the design of the DCO for ADPLL 1, the minimum possible accumulator width was chosen. From the equation used to calculate the frequency step size, it may have seemed that

by doubling the bias point and increasing the width of the counter to 13 bits a finer resolution could be achieved.

$$f_{step} = \frac{f_{FPGA}}{2^{12}} = 62.988 \text{ kHz} \tag{5.1}$$

$$f_{step} = \frac{f_{FPGA}}{2^{13}} = 31.494 \text{ kHz} \tag{5.2}$$

As Table 5.4 shows, this is not in fact the case. Both measurements were carried out using identical loop filter gains, network configurations and reference signal. While in uni-directional mode some difference is observed in the maximum TIE, but in all other aspects no statistically relevant changes can be seen.

	Jitter Standard Deviation (ns)			Max. Time Interval Error (ns)			Skew (ns)		
	PLL 11	PLL 12	PLL 22	PLL 11	PLL 12	PLL 22	PLL 11	PLL 12	PLL 22
Accum. Width 12	-	-	-	-	-	-	-	-	-
Uni-dir.	1.7693	1.8463	1.8862	7.3060	7.2565	10.151	2.2791	1.2822	1.6299
Bi-dir.	1.9964	1.9146	1.9572	15.0890	18.2830	19.673	9.2449	11.6120	12.0520
Accum. Width 13	-	-	-	-	-	-	-	-	-
Uni-dir.	1.7946	1.8161	1.9048	6.8614	7.1495	9.371	2.2257	1.6328	1.3778
Bi-dir.	2.0016	1.9217	1.9605	14.9580	18.938	19.510	9.1678	12.0000	11.7690

Table 5.4: DCO Accumulator Width Comparison.

5.4.4 LF Input Delay Register

Previously, when comparing the loss of lock gain relationship to that proposed by Koskin *et al*, it was mentioned that the LF used in this project does not have an input delay register. The lack of agreement between the theory and experimental results was attributed to the input delay register, or lack thereof. In order to provide a basis for this claim, the performance of a network in 2x2 configuration, was analysed and the results of this analysis are presented in Table 5.5. While the C2C jitter is statistically similar, there is significant different to be observed in the TIE, where an approximately 50% increase can be observed, measurement for measurement. As the maximum TIE is significantly impacted by the presence of any skew, as skew is the average value of this difference, it is interesting to note that the relationship is in fact the opposite. It is in precisely this situation that C2C jitter as a measurement is non-ideal. C2C jitter does not differentiate between the situations in which while the period variation might be the same for multiple clock cycles in

the same or opposite directions. As the extra cycle of delay due to the input register increases the reaction time of the ADPLL to any changes, the delay between clock edges increases. TIE is, however, affected by this difference which can be clearly seen in the table.

	Jitter Standard Deviation (ns)			Max. Time Interval Error (ns)			Skew (ns)		
	PLL 11	PLL 12	PLL 22	PLL 11	PLL 12	PLL 22	PLL 11	PLL 12	PLL 22
W/ LF input reg	-	-	-	-	-	-	-	-	-
Free PLLs	0.66757	0.84477	0.71308	5.9142	6.1353	5.3939	-0.36997	-0.20981	-1.3344
Uni-dir.	0.66934	0.83894	0.71271	5.8779	7.1746	6.8914	-0.36645	-0.05647	-0.36178
Bi-dir.	0.68372	0.83193	0.68570	7.5705	9.6292	9.7715	0.68901	1.58720	1.5553
W/o LF input reg	-	-	-	-	-	-	-	-	-
Free PLLs	0.67899	0.65618	0.61415	4.2527	3.6478	3.8483	-0.31418	0.11897	-0.57619
Uni-dir.	0.68534	0.69405	0.63666	4.3257	4.2901	4.6185	-0.38849	0.23363	0.21156
Bi-dir.	0.68248	0.70175	0.61356	5.3497	6.2640	6.8079	0.52553	1.7025	1.9499

Table 5.5: Impact of LF Input Delay Register.

5.4.5 Impact of Frequency Divider

The major benefit of a PLL network over other forms of coupled oscillator based clock distribution systems, is that the coupling can be done using a signal significantly reduced in frequency, thereby reducing the power consumption of the related hardware. It is therefore important to ensure that the insertion of a feedback divider does not introduce any further jitter to the system. Table 5.6 presents the results of measurements at a number of divisions levels. As re-implemenation was performed between each measurement, there is some variation present. However, it is notable that for each ADPLL that saw a C2C jitter increase, there was another that performed better than at a lower level of division. Accordingly, it can be claimed that the insertion of a feedback divider does not impact the C2C jitter present. Intuitively this makes sense as the control code will change more violently, but proportionally less frequently.

It, however, does have a significant impact on the maximum TIE, and notably this occurs without any meaningful change in skew. This confirms an increase in jitter due to the greater division levels as the source of the performance degradation. By examining how the dividers behave, this can be explained. When a divider is inserted into the network, the phase error is computed at a greatly reduced rate. This means the rate at which the LF is updated also drops, and therefore the control code is changes n times less frequently, where n is the division level. The dividers reduce the speed at which the individual ADPLLs can react to a phase difference, while

	Jitter Standard Deviation (ns)			Max. Time Interval Error (ns)			Skew (ns)		
	PLL 11	PLL 12	PLL 22	PLL 11	PLL 12	PLL 22	PLL 11	PLL 12	PLL 22
No Divider	-	-	-	-	-	-	-	-	-
Free PLLs	0.66757	0.84477	0.71308	5.9142	6.1353	5.3939	-0.36997	-0.20981	-1.3344
Uni-dir.	0.66934	0.83894	0.71271	5.8779	7.1746	6.8914	-0.36645	-0.05647	-0.36178
Bi-dir.	0.68372	0.83193	0.68570	7.5705	9.6292	9.7715	0.68901	1.58720	1.5553
Divide by 2	-	-	-	-	-	-	-	-	-
Free PLLs	0.68252	0.51750	1.1074	4.9464	4.0728	6.7426	-0.91985	-1.7024	-1.84279
Uni-dir.	0.68441	0.53203	1.1081	4.8707	4.9684	8.9973	-0.99034	-1.0240	-0.63989
Bi-dir.	0.69524	0.54953	1.1033	6.8180	7.8760	12.288	0.00337	0.69148	1.3085
Divide by 4	-	-	-	-	-	-	-	-	-
Free PLLs	0.77564	0.55934	0.65770	10.576	6.7330	12.647	-1.1507	-1.8670	-2.3381
Uni-dir.	0.77443	0.56544	0.66019	10.210	7.4612	15.166	-1.2283	-1.2691	-1.0948
Bi-dir.	0.77657	0.57043	0.66809	11.408	10.859	17.242	-0.28439	0.40697	0.8351

Table 5.6: Network Performance at Different Divider Levels.

locking the resulting control code in place for a greater number of clock cycles as the division level increases. This greater duration may cause a control code to be applied for an excessively long number of cycles, resulting in a drift of the rising edge of the divided/generated signal away from the reference, which will manifest as TIE. These tests were carried out with identical gains for the proportional and integral paths in order to demonstrate the effect, however, an appropriate reduction in these gains can correct this problem by reducing the magnitude of the control code alteration. Figure 5.4 can confirm this behaviour. In Plot (a), as the division level increases, the number of cycles spent at a $200\ \mathrm{ns}$ period decreases, which would line up with the overcorrection, thus resulting in the increased TIE seen in Plot (b).

ADPLL Use Cases

As each design was introduced, the most appropriate use cases for that particular ADPLL were suggested. ADPLL networks can be used in a number of roles, from modelling system dynamics or the refinement of behavioural simulation models to the validation, verification and testing of potential ASIC based networks (Zianbetov & Shan) to the design of other hardware for the purposes of network performance improvement or control, however, some architectures of ADPLL are more suited to particular roles than others.

As mentioned when FPGA clocked designs were introduced, their configurability makes them particularly suited to the validation and verification of designs destined for use in ASICs, as was

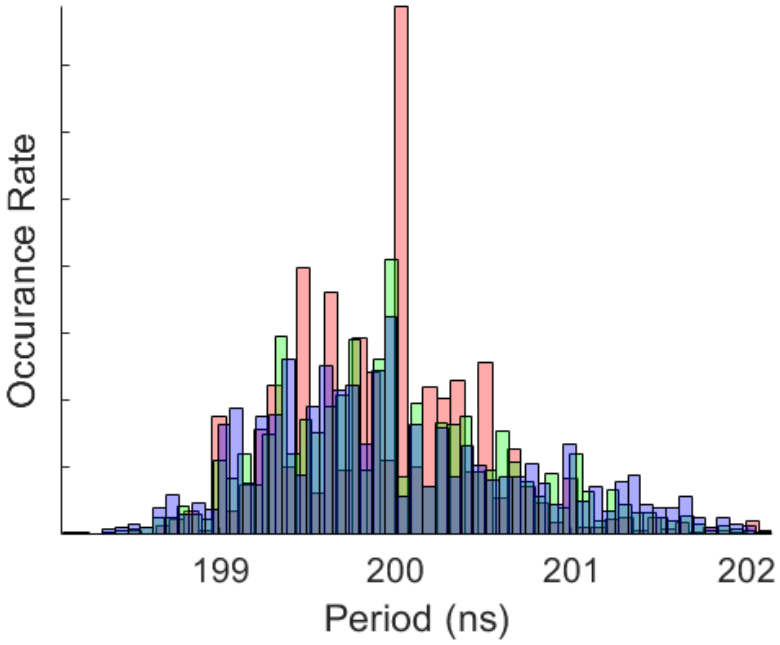


Figure 5.4: Period distribution at differen divider levels.

the case with Zianbetov and Shan [4, 13]. In contrast to the RO based designs, in which configurability is more more restricted, it is possible to replicate the resolution, centre frequencies and other ADPLL characteristics of the design requiring verification at a scaled down frequency. This can be carried out at large division ratios, where the FPGA clock period is not a restricting factor, such as the 50 kiloHz centre frequency used by Zianbetov and Shan. In order to reduce the frequency to this extent with inverters over 30 thousand would be required, introducing the potential for even greater variation in centre frequencies due to implementation. The other issue arising out of this number of inverters is the floorspace of the FPGA that would be consumed implementing just one RO, in the case of the Xilinx XC7A100T-1CSG324C and extrapolating based on the resource usage of the ROs implemented, a single oscillator would consume half the look-up tables available, eliminating the possibility of a network.

While the FPGA based designs feature characteristics making them particularly suitable for use as a validation tool, their performance in a network reduces their suitability for the refinement of software based simulations, or in the analysis of new control or performance enhancement tools. Referring back to Table 5.1, C2C jitter is significantly greater than the inverter based designs, and a large degree of skew is present making this design non-ideal for these purposes. In particular, as seen with the increase to a 3x3 network, this skew becomes an issue that limits the network size.

Figure 5.3, which demonstrated the impact of period distribution, highlights a second major unrealistic behaviour present in FPGA clocked ADPLL designs, as an ASIC destined design will not exhibit this identical linear period distribution. While ADPLL 2 will feature a large degree of variation due to implementation differences, these differences are only present in the RO not the PFD. As testing Designs 2 & 3 has shown, in particular the results displayed in Table 5.1, apart from a larger skew, the jitter performance is quite similar, meaning that the extra complexity of inverter based PFD is not required in order to achieve as accurate behaviour as can be achieved in this environment. The implementation based variation of the detector step manifests itself as an increase in the jitter across all network modes, as would be expected, compared to the FPGA based design.

Accordingly, if the aim of this platform is the enhancement of a model, the validation of theoretical results, or analysing the performance of new control or performance enhancement hardware the RO based designs are idea for this purpose. Two members of the research team here in UCD, Eugene Koskin and Pierre Bisiaux, have been using an ADPLL with the same architecture as Design 3 for these purposes.

Chapter 6

Conclusion

6.1 Conclusions

6.2 Suggested Future Work

6.2.1 Tapped Delay Line Characterisation

As previously mentioned, characterisation of the Tapped Delay Line (TDL) was not carried out due to the time it would have taken to perform the required measurements and tests. This characteristion can be accomplished through the use of an off-chip tunable delay. Off-chip is unfortunately a requirement, as the only delays available of the order required to test the step sizes are the variable delay of the inverters themselves. A resistor & capacitor delay circuit can be used to perform these test, and the output of the Phase Frequency Detector (PFD) analysed on a mixed-signal oscilloscope alongside the delayed and reference waveforms. The use of an off-chip delay element necessitates careful construction/cailbration of the measurement environment to eliminate the delay stemming from the signal path through input/output buffers as it leaves the chip.

6.2.2 Increased Network Size

A 4x4 network was a stretch-goal of this project, depending on the time remaining after once the minumim goal of a 3x3 network had been reached. Unfortuantely due to problems with the clocked phase detector which resulted in mode-locking, incorrect Ring Oscillator (RO) inverter selection logic and the projected time requirement to ensure alignment of each RO this was not achieved. A 4x4 network would better explain some dynamics, such as the skew growth as the All-Digital Phase Lock Loop (ADPLL) indices increase. However. progressing beyond a 3x3 network means that less than 20% of ADPLL output signals can be examined at any given point in time. As such, it may be useful to develop a more advanced measurement set-up that can either capture more signals simultaneously, or implement a measurement system that makes more efficient use of the available

probes.

6.2.3 FPGA Clocked, Linear Period DCO

While this design of Digitally Controlled Oscillator (DCO) was proposed earlier in this thesis, it was not implemented in any ADPLLs, as it was only realised that the output waveform did not matter, just the location of the rising edges was not made until later in the project, once the pitfall encountered in the clocked PFD design was resolved. Until this point the non-square wave output had eliminated the design from consideration. However, as in an Application Specific Integrated Circuit (ASIC) linear period designs are easier to implement, it would be ideal to have the benefit of frequency scaled hardware simulations with a DCO that more closely relates the the eventual system.

6.2.4 Procedural Network Instantiation

This suggestion is directed at making the user experience simpler. As implemented, networks based on ADPLLs containing ROs are implemented in separate files to their Field Programmable Gate Array (FPGA) clocked DCO based counterparts as are the networks of difference sizes. At a network size of 2x2 this was not a problem, but at the scale of 3x3 network the modules became somewhat cumbersome to work with, with a large number of interconnects. For 3x3 and larger networks it would be expedient to create the network with a `generate` statement, as this would reduce the number of lines of code and the amount of named interconnects required. Instead the user could set a number of configuration parameters and the network would be created accordingly. This could even allow for the use of different ADPLL types in the same network with ease. Unfortunately the tasks of floorplanning and oscillator alignment would still be left to the user to perform.

Bibliography

[1] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," 1965. 3

[2] P. E. Ross, "Why cpu frequency stalled," *IEEE Spectrum*, vol. 45, no. 4, 2008. 3

[3] "Intel ark," https://ark.intel.com/content/www/us/en/ark.html, accessed 2019-04-06. 3

[4] E. Zianbetov, "Distributed clocking for synchronous soc," Ph.D. dissertation, Doctoral School of Informatics, Telecommunications and Electronics, UPMC, 4 Place Jussieu, 75005 Paris, France, 3 2013. 4, 5, 7, 11, 16, 20, 22, 31, 72

[5] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing power in high-performance microprocessors," in *Proceedings of the 35th annual Design Automation Conference*. ACM, 1998, pp. 732–737. 3, 6

[6] A. Abdelhadi, R. Ginosar, A. Kolodny, and E. G. Friedman, "Timing-driven variation-aware nonuniform clock mesh synthesis," in *Proceedings of the 20th symposium on Great lakes symposium on VLSI*. ACM, 2010, pp. 15–20. 6

[7] G. Chen, H. Chen, M. Haurylau, N. A. Nelson, D. H. Albonesi, P. M. Fauchet, and E. G. Friedman, "On-chip copper-based vs. optical interconnects: Delay uncertainty, latency, power, and bandwidth density comparative predictions," in *2006 International Interconnect Technology Conference*. IEEE, 2006, pp. 39–41. 7

[8] T. Yamashita, T. Fujimoto, and K. Ishibashi, "A dynamic clock skew compensation circuit technique for low power clock distribution," in *Integrated Circuit Design and Technology, 2005. ICICDT 2005. 2005 International Conference on*. IEEE, 2005, pp. 7–10. 8

[9] H. Mizuno and K. Ishibashi, "A noise-immune ghz-clock distribution scheme using synchronous distributed oscillators," in *Solid-State Circuits Conference, 1998. Digest of Technical Papers. 1998 IEEE International*. IEEE, 1998, pp. 404–405. 9

[10] G. A. Pratt and J. Nguyen, "Distributed synchronous clocking," *IEEE transactions on parallel and distributed systems*, vol. 6, no. 3, pp. 314–328, 1995. 10, 11

[11] V. Gutnik and A. Chandrakasan, "Active ghz clock network using distributed plls," in *Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International*. IEEE, 2000, pp. 174–175. 10

[12] E. Zianbetov, D. Galayko, F. Anceau, M. Javidan, C. Shan, O. Billoint, A. Korniienko, E. Colinet, G. Scorletti, J. Akrea *et al.*, "Distributed clock generator for synchronous soc using adpll network," in *Custom Integrated Circuits Conference (CICC), 2013 IEEE*. IEEE, 2013, pp. 1–4. 11, 12

[13] C. Shan, "Distributed clocking for large synchronous soc," Ph.D. dissertation, Doctoral School of Informatics, Telecommunications and Electronics, UPMC, 4 Place Jussieu, 75005 Paris, France, 10 2014. 11, 14, 15, 16, 22, 31, 72

[14] Y. Chen and K. D. Pedrotti, "Rotary traveling-wave oscillators, analysis and simulation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 1, pp. 77–87, 2011. 13

[15] E. Koskin, D. Galayko, O. Feely, and E. Blokhina, "Generation of a clocking signal in synchronized all-digital pll networks," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 6, pp. 809–813, 2018. 15, 35, 36, 66

[16] K. Lata and M. Kumar, "Adpll design and implementation on fpga," in *2013 International Conference on Intelligent Systems and Signal Processing (ISSP)*, March 2013, pp. 272–277. 17

[17] E. Koskin, P. Bisiaux, D. Galayko, and E. Blokhina, "All-digital phase-locked loop arrays: Investigation of synchronisation and jitter performance through fpga prototyping," submitted. 17, 24

[18] A. Fernández-Álvarez, M. Portela-García, M. García-Valderas, J. López, and M. Sanz, "Hw/sw co-simulation system for enhancing hardware-in-the-loop of power converter digital controllers," *IEEE Journal of Emerging and Selected Topics in Power Electronics*, vol. 5, no. 4, pp. 1779–1786, 2017. 17

[19] O. Lucia, I. Urriza, L. A. Barragan, D. Navarro, O. Jimenez, and J. M. Burdio, "Real-time fpga-based hardware-in-the-loop simulation test bench applied to multiple-output power converters," *IEEE Transactions on Industry Applications*, vol. 47, no. 2, pp. 853–860, 2011. 17

[20] G. Wang and Y. Chiu, "Fast fpga emulation of background-calibrated sar adc with internal redundancy dithering," in *Proceedings of the IEEE 2013 Custom Integrated Circuits Conference*. IEEE, 2013, pp. 1–4. 18

[21] "Clock jitter definitions and measurement methods, *SiT-AN10007 Rev 1.2*," https://www.sitime.com/api/gated/AN10007-Jitter-and-measurement.pdf, accessed 2019-04-10. 18

[22] M. Kellermann, *Creating a Controllable Oscillator Using the Virtex-5 FPGA IODELAY Primitive*, 1st ed., Xilinx, Internet, XAPP872, 4 2009. 20

[23] "File:ring oscillator (3-stage).svg," https://en.wikipedia.org/wiki/File:Ring_oscillator_(3-stage).svg, accessed 2019-04-10. 23

[24] E. Koskin, "Mathematical modelling and system-level design of all-digital phase-locked loop networks," Ph.D. dissertation, School of Electrical and Electronic Engineering, University College Dublin, Belfield, Dublin 4, Ireland, 2019. 24

[25] P. Tasic, "Hardware-based investigation of nonlinear dynamics of bang-bang phase-locked loop," Ph.D. dissertation, University College Dublin, Belfield, Dublin 4, Ireland, 2 2011. 26

[26] J. A. Tierno, A. V. Rylyakov, and D. J. Friedman, "A wide power supply range, wide tuning range, all static cmos all digital pll in 65 nm soi," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 42–51, 2008. 31

[27] M. Javidan, E. Zianbetov, F. Anceau, D. Galayko, A. Korniienko, E. Colinet, G. Scorletti, J.-M. Akre, and J. Juillard, "All-digital pll array provides reliable distributed clock for socs," in *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*. IEEE, 2011, pp. 2589–2592. 33

[28] "Localparam, verilog reference guide," https://www.hdlworks.com/hdl_corner/verilog_ref/items/LocalParam.htm, accessed 2019-04-13. 38

[29] "Parameter, verilog reference guide," https://www.hdlworks.com/hdl_corner/verilog_ref/items/Parameter.htm, accessed 2019-04-13. 40

[30] "Vivado design suite user guide - synthesis," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug901-vivado-synthesis.pdf, accessed 2019-04-15. 51

[31] "Vivado design suite user guide - implementation," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug904-vivado-implementation.pdf, accessed 2019-04-15. 52

[32] "Artix-7 fpgas data sheet," http://www.farnell.com/datasheets/2301213.pdf, accessed 2019-04-17. 58

[33] "Nexys 4^{tm} fpga board reference manual," https://reference.digilentinc.com/_media/nexys:nexys4:nexys4_rm.pdf, accessed 2019-04-17. 59

Appendices

Appendix A

Key Verilog Modules

A selection of files containing the core Verilog modules are included in this appendix. All files used can be located on github at **https://github.com/ConchuOD/fpga_adpll**.

A.1 Phase Detector

```
`timescale 1ns / 1ps
/*****************************************************************************/
/* Author    : Conor Dooley                                                  */
/* Date      : ??-November-2019                                              */
/* Function : FPGA clocked phase detector with a resolution of 1/fpga_clock, */
/*            made up of a up/down counter & a state machine for sign        */
/*            detection                                                      */
/*****************************************************************************/
module PhaseDetector #(parameter WIDTH = 8) (
        input  wire                     reset_i,            //reset high
        input  wire                     fpga_clk_i,
        input  wire                     reference_i,
        input  wire                     generated_i,
        output wire signed [WIDTH-1:0] pd_clock_cycles_o
    );

    /*****************************************************************************/
    /* Define nets                                                               */
    /*****************************************************************************/

    wire signed [WIDTH-1:0] counter_val_x;
    wire        [1:0]       count_instr_x;
    wire                    save_and_clear_x;
    wire                    counter_cleared_x;

    wire ref_edge_x;
    wire gen_edge_x;

    wire generated_synced_i;
    wire reference_synced_i;

    /*****************************************************************************/
    /* Phase detector sub-modules                                                */
    /*****************************************************************************/

    //synchronisers to avoid metastability & perform measurement quantisation
    Synchroniser genSync (
        .clk_i(fpga_clk_i),
        .async_i(generated_i),
        .sync_o(generated_synced_i)
    );

    Synchroniser refSync (
        .clk_i(fpga_clk_i),
        .async_i(reference_i),
        .sync_o(reference_synced_i)
    );
```

```
    //two directional counter with variable width 2s complement answer
    UpDownCounter #(.WIDTH(WIDTH)) upDownCounter (
        .reset_i(reset_i),
        .clear_i(save_and_clear_x),
        .fpga_clk_i(fpga_clk_i),
        .count_instr_i(count_instr_x),
        .counter_val_o(counter_val_x)
    );

    //module holds the result constant between measureme
    SaveCounter #(.WIDTH(WIDTH)) saveCounter (
        .fpga_clk_i(fpga_clk_i),
        .reset_i(reset_i),
        .trigger_i(save_and_clear_x),
        .counter_val_i(counter_val_x),
        .counter_val_saved_o(pd_clock_cycles_o),
        .counter_cleared_o(counter_cleared_x)
    );

    //finite state machine governs phase detector behaviour
    StateMachine stateMachine(
        .reset_i(reset_i),
        .fpga_clk_i(fpga_clk_i),
        .reference_synced_i(reference_synced_i),
        .generated_synced_i(generated_synced_i),
        .counter_cleared_i(counter_cleared_x),
        .save_and_clear_o(save_and_clear_x),
        .count_instr_o(count_instr_x)
    );

endmodule // PhaseDetector

`timescale 1ns / 1ps
/*************************************************************************/
/* Author    : Conor Dooley                                              */
/* Date      : ??-November-2019                                          */
/* Function : Bidirectional counter of a width set at compile time. Output   */
/*              is a 2s complement signed integer                        */
/*************************************************************************/
module UpDownCounter #(parameter WIDTH = 20)(
        input wire reset_i, //reset high
        input wire clear_i,
        input wire fpga_clk_i,
        input wire [1:0] count_instr_i,
        output wire signed [WIDTH-1:0] counter_val_o
    );

    /*************************************************************************/
    /* Define constants and nets                                             */
    /*************************************************************************/

    //count instructions from state machine
    localparam [1:0] DISABLE = 2'b00, COUNT_UP = 2'b01, COUNT_DOWN = 2'b10;

    //constants for overflow prevention in accumulator
    localparam [WIDTH-1:0] MAXVAL = {1'b0, {(WIDTH-1){1'b1}}};
    localparam [WIDTH-1:0] MINVAL = {1'b1, {(WIDTH-2){1'b0}}, 1'b1};

    reg [WIDTH-1:0] count_r, next_count_r, count_inc_r;

    /*************************************************************************/
    /* Logic                                                                 */
    /*************************************************************************/

    //register to form counter
    always @ (posedge fpga_clk_i)
    begin
        if (reset_i == 1'b1)      count_r <= {(WIDTH){1'b0}};
        else if(clear_i == 1'b1) count_r <= {(WIDTH){1'b0}};
        else count_r <=           next_count_r;
    end
```

```
    //switch statement decides counter increment based on direction command
        from FSM
    always @ (count_instr_i)
    begin
        case (count_instr_i)
            DISABLE:
                count_inc_r = {(WIDTH){1'b0}};
            COUNT_UP:
                count_inc_r = {{(WIDTH-1){1'b0}}, 1'b1};
            COUNT_DOWN:
                count_inc_r = {(WIDTH){1'b1}};
             default : count_inc_r = {(WIDTH){1'b0}};
        endcase
    end

    //determine next count value based on direction and possibilty of overflow
    always @(count_inc_r, count_r)
    begin
        if (count_r == MAXVAL)
             next_count_r = count_r; //at maxval
        else if (count_r == MINVAL)
             next_count_r = count_r; //at symmmetrical minval
        else
            next_count_r = count_r + count_inc_r; //otherwise count
    end

    //assign count value as output
    assign counter_val_o = count_r;

endmodule // UpDownCounter
```

A.2 Phase Detector with TDL

```
`timescale 1ns / 1ps
/*****************************************************************************/
/* Author    : Conor Dooley                                                  */
/* Date      : ??-March-2019                                                 */
/* Function : Inverter based tapped delay line sig-num phase detector.       */
/*****************************************************************************/
module PhaseDetectorDL #(parameter WIDTH = 5) (
        input  wire                     reset_i,
        input  wire                     fpga_clk_i, //unused but preserved for
            interface compatability
        input  wire                     reference_i,
        input  wire                     generated_i,
        output wire signed [WIDTH-1:0] pd_clock_cycles_o //used & named as
            above
    );

    /*************************************************************************/
    /* Define nets and constants                                             */
    /*************************************************************************/

    //number of bits left after sign bit
    localparam MAG_WIDTH = WIDTH-1;

    //number of taps along the delay line
    localparam NUM_TAPS = (2 ** MAG_WIDTH) - 1;

    //liberal use of DONT_TOUCH required to maintain timing behaviour
    (* DONT_TOUCH = "TRUE" *) reg ref_q_r, gen_q_r;
    (* DONT_TOUCH = "TRUE" *) reg sign_delay_r;
    (* DONT_TOUCH = "TRUE" *) reg clear_tdl_c;

    (* DONT_TOUCH = "TRUE" *) wire done_c, count_c, clear_c;
    (* DONT_TOUCH = "TRUE" *) wire which_first_c;
    (* DONT_TOUCH = "TRUE" *) wire sign_c;

    (* DONT_TOUCH = "TRUE" *) wire [1:NUM_TAPS]    count_input_c;
```

```
(* DONT_TOUCH = "TRUE" *) reg  [1:NUM_TAPS]   error_taps_r;
(* DONT_TOUCH = "TRUE" *) reg  [1:NUM_TAPS]   error_taps_buff_r;
(* DONT_TOUCH = "TRUE" *) wire [0:2*NUM_TAPS] count_delayed_c;

reg [MAG_WIDTH-1:0] error_bin_r;
reg [WIDTH-1:0] error_2s_comp_c;

//double named net for clarity
assign clear_c = done_c;

/*****************************************************************************/
/* Sign detection                                                            */
/*****************************************************************************/

//reference edge detector
always @ (posedge reference_i or posedge reset_i or posedge clear_c)
begin
    if (reset_i || clear_c) ref_q_r <= 1'b0;

    else if (reference_i) ref_q_r <= 1'b1;

    else ref_q_r <= 1'b0;
end

//generated signal edge detector
always @ (posedge generated_i or posedge reset_i or posedge clear_c)
begin
    if (reset_i || clear_c) gen_q_r <= 1'b0;

    else if (generated_i) gen_q_r <= 1'b1;

    else gen_q_r <= 1'b0; //should be unreachable
end

//tapped delay line control signals
(* DONT_TOUCH = "TRUE" *) and done_and (done_c, ref_q_r, gen_q_r);
(* DONT_TOUCH = "TRUE" *) or  count_or (count_c, ref_q_r, gen_q_r);

//Arbitrator imitation circuit
(* DONT_TOUCH = "TRUE" *) SRLatchGate arbitration
(
    .R(~ref_q_r),
    .S(~gen_q_r),
    .Q(which_first_c)
);

//Sign detection circuit
(* DONT_TOUCH = "TRUE" *) SRLatchGate sign
(
    .R(which_first_c),
    .S(~which_first_c),
    .Q(sign_c)
);

//output buffer
always @ (posedge done_c or posedge reset_i)
begin
    if (reset_i)     sign_delay_r <= 0'b0;

    else if (done_c) sign_delay_r <= sign_c;

    else             sign_delay_r <= 0'b0; //should be unreachable
end

/*****************************************************************************/
/* Tapped delay line                                                         */
/*****************************************************************************/

//connect delay line to input signal
assign count_delayed_c[0] = count_c;

//tdl clear signal logic
```

```
always @ (count_c or done_c)
begin
    if (done_c) clear_tdl_c = 1'b1;

    else        clear_tdl_c = 1'b0;
end

//tapped delay line
genvar i;
generate
    for (i = 1;i <= NUM_TAPS;i = i+1)
    begin: TAPPED_DELAY_LINE
        //double inverter delay between taps
        (* DONT_TOUCH = "TRUE" *) not tap_delay_1(count_delayed_c[2*i-1],
            count_delayed_c[2*(i-1)]);
        (* DONT_TOUCH = "TRUE" *) not tap_delay_2(count_delayed_c[2*i],
            count_delayed_c[2*i-1]);

        assign count_input_c[i] = count_delayed_c[2*(i-1)];

        //tap registers
        always @ (posedge count_input_c[i] or posedge reset_i or posedge
            clear_tdl_c)
        begin
            //neg of clear_tdl_c
            if (reset_i == 1'b1 || clear_tdl_c == 1'b1) error_taps_r[i] <=
                1'b0;

            else if (count_c == 1'b1)                   error_taps_r[i] <=
                1'b1;
            // in case edge arrives after counting period has ended
            else                                        error_taps_r[i] <=
                1'b0;

        //output buffer registers
        always @ (posedge done_c or posedge reset_i)
        begin
            if (reset_i)      error_taps_buff_r[i] <= 1'b0;

            else if (done_c) error_taps_buff_r[i] <= error_taps_r[i];

            else              error_taps_buff_r[i] <= 1'b0;
        end

    end
endgenerate

/*****************************************************************/
/* Taps to output                                                */
/*****************************************************************/

//temperature code to binary
integer j;
always @ (*)
begin
    error_bin_r = {(MAG_WIDTH){1'b0}};
    for (j=1;j<=NUM_TAPS;j=j+1)
    begin
        error_bin_r = error_bin_r + error_taps_buff_r[j];
    end
end

//concatanation of sign and magnutide
always @ (error_bin_r or sign_delay_r)
begin
    if(sign_delay_r)    error_2s_comp_c = $signed({sign_delay_r , ~(
        error_bin_r[MAG_WIDTH-1:0])}-1'b1);
    else                error_2s_comp_c = $signed({sign_delay_r ,
        error_bin_r[MAG_WIDTH-1:0]});
end

//assign output
```

```
    assign pd_clock_cycles_o = error_2s_comp_c;

endmodule // PhaseDetectorDL
```

A.3 Error Combiner

```
`timescale 1ns / 1ps
/*************************************************************************/
/* Author    : Conor Dooley                                              */
/* Date      : ??-Feburary-2019                                          */
/* Function : Performs a weighted average of up to 4 error signals, with */
/*            weights set dynamically                                    */
/*************************************************************************/
module ErrorCombiner #(
        parameter ERROR_WIDTH  = 8,
        parameter WEIGHT_WIDTH = 4
    )
    (
        input  wire reset_i,
        input  wire signed [WEIGHT_WIDTH-1:0] weight_0_i,
        input  wire signed [WEIGHT_WIDTH-1:0] weight_1_i,
        input  wire signed [WEIGHT_WIDTH-1:0] weight_2_i,
        input  wire signed [WEIGHT_WIDTH-1:0] weight_3_i,
        input  wire signed [ERROR_WIDTH-1:0]  error_0_i,
        input  wire signed [ERROR_WIDTH-1:0]  error_1_i,
        input  wire signed [ERROR_WIDTH-1:0]  error_2_i,
        input  wire signed [ERROR_WIDTH-1:0]  error_3_i,
        output wire signed [ERROR_WIDTH-1:0]  error_comb_o
    );

    /*************************************************************************/
    /* Define constants and nets                                             */
    /*************************************************************************/

    localparam WEIGHTED_WIDTH = ERROR_WIDTH+WEIGHT_WIDTH;
    localparam SUM_WIDTH      = WEIGHTED_WIDTH+2; //increased by two to fit x4
        multiplication

    wire signed [WEIGHTED_WIDTH-1:0] weighted_0_c;
    wire signed [WEIGHTED_WIDTH-1:0] weighted_1_c;
    wire signed [WEIGHTED_WIDTH-1:0] weighted_2_c;
    wire signed [WEIGHTED_WIDTH-1:0] weighted_3_c;

    wire signed [SUM_WIDTH-1:0]   weighted_sum_c;
    wire signed [ERROR_WIDTH-1:0] result_div4_c;

    /*************************************************************************/
    /* Combinational logic                                                   */
    /*************************************************************************/

    //perform multiplication by weights
    assign weighted_0_c = weight_0_i*error_0_i;
    assign weighted_1_c = weight_1_i*error_1_i;
    assign weighted_2_c = weight_2_i*error_2_i;
    assign weighted_3_c = weight_3_i*error_3_i;

    //sum the results
    assign weighted_sum_c = weighted_0_c+weighted_1_c+weighted_2_c+weighted_3_c
        ;

    //divide by four to perform average
    assign result_div4_c = $signed({weighted_sum_c[WEIGHTED_WIDTH-1] ,
        weighted_sum_c[((ERROR_WIDTH-1)-1)+2:0+2]});

    //assign result to output
    assign error_comb_o = result_div4_c;

endmodule // ErrorCombiner
```

A.4 Loop Filter

```
`timescale 1ns / 1ps
/*************************************************************************/
/* Author    : Conor Dooley                                              */
/* Date      : 29-March-2019                                             */
/* Function : Variable gain PI controller for use in ADPLLs              */
/*************************************************************************/
module LoopFilter #(
        parameter DYNAMIC_VAL = 0,  //whether or not to set gains at runtime
        parameter ERROR_WIDTH = 5,  //width of the error signal
        parameter DCO_CC_WIDTH = 5, //width of the filter output
        parameter KP_WIDTH = 5,
        parameter KP = 5'd1,        //compile time default kp
        parameter KI_WIDTH = 7,
        parameter KI = 7'd1         //compile time default ki
    )
    (
        input  wire gen_clk_i,
        input  wire reset_i, //reset high
        input  wire [KP_WIDTH-1:0] kp_i,
        input  wire [KI_WIDTH-1:0] ki_i,
        input  wire signed [ERROR_WIDTH-1:0] error_i,
        output wire signed [DCO_CC_WIDTH-1:0] dco_cc_o
    );

    /*************************************************************************/
    /* Define constants and nets                                             */
    /*************************************************************************/

    localparam KP_MULT_RES_WIDTH = ERROR_WIDTH+KP_WIDTH;
    localparam KI_MULT_RES_WIDTH = ERROR_WIDTH+KI_WIDTH;
    localparam KI_ACCUM_OVERHEAD = 0; //integration accumulator extra width
    localparam SUM_WIDTH         = KI_MULT_RES_WIDTH;

    //gains
    reg  signed [KP_WIDTH-1:0] kp_x;
    reg  signed [KI_WIDTH-1:0] ki_x;

    //input delay register
    reg  signed [ERROR_WIDTH-1:0] error_delay_r;

    //kp path nets
    wire signed [KP_MULT_RES_WIDTH-1:0] kp_error_c;
    wire signed [KI_MULT_RES_WIDTH-1:0] kp_error_padded_c;

    //ki path nets
    wire signed [KI_ACCUM_OVERHEAD+KI_MULT_RES_WIDTH-1:0] ki_error_c;
    wire signed [KI_ACCUM_OVERHEAD+KI_MULT_RES_WIDTH-1:0] ki_error_inte_c;
    reg  signed [KI_ACCUM_OVERHEAD+KI_MULT_RES_WIDTH-1:0] ki_error_inte_delay_r
        ;
    wire signed [KI_MULT_RES_WIDTH-1:0] ki_error_resize_c;

    //path combining nets
    wire signed [SUM_WIDTH-1:0] error_sum_c;
    wire signed [DCO_CC_WIDTH-1:0] error_sum_trun_c;
    reg  signed [DCO_CC_WIDTH-1:0] error_sum_trun_delay_r;

    /*************************************************************************/
    /* Select from dynamic or compile time set gains                         */
    /*************************************************************************/

    always @(DYNAMIC_VAL or reset_i or kp_i or ki_i)
    begin
        if (DYNAMIC_VAL)
        begin
            kp_x = kp_i;
            ki_x = ki_i;
        end
        else
```

```
        begin
            kp_x = KP;
            ki_x = KI;
        end
    end

    /*****************************************************************************/
    /* Input & output delays                                                     */
    /*****************************************************************************/

    always @ (posedge gen_clk_i or posedge reset_i)
    begin
        if(reset_i) error_sum_trun_delay_r <= {(DCO_CC_WIDTH){1'b0}};
        else error_sum_trun_delay_r <= error_sum_trun_c;
    end

    always @ (posedge gen_clk_i or posedge reset_i)
    begin
        if(reset_i) error_delay_r <= {(ERROR_WIDTH){1'b0}};
        else error_delay_r <= error_i;
    end

    /*****************************************************************************/
    /* Kp path                                                                   */
    /*****************************************************************************/

    //multiply by kp
    assign kp_error_c = error_i*kp_x;

    /*****************************************************************************/
    /* Ki path                                                                   */
    /*****************************************************************************/

    //multiply by ki
    assign ki_error_c = error_i*ki_x;

    //add to value in accumulator to perform integration
    assign ki_error_inte_c = ki_error_inte_delay_r+ki_error_c;
    assign ki_error_resize_c = $signed(ki_error_inte_c[KI_ACCUM_OVERHEAD+
        KI_MULT_RES_WIDTH-1:KI_ACCUM_OVERHEAD]);

    //accumulator register
    always @ (posedge gen_clk_i or posedge reset_i)
    begin
        if(reset_i) ki_error_inte_delay_r <= {(KI_ACCUM_OVERHEAD+
            KI_MULT_RES_WIDTH-1){1'b0}};
        else ki_error_inte_delay_r <= ki_error_inte_c;
    end

    /*****************************************************************************/
    /* Combine paths                                                             */
    /*****************************************************************************/

    //pad kp to match ki width then add
    assign kp_error_padded_c = $signed({kp_error_c, {(KI_WIDTH-KP_WIDTH){1'b0}}
        });
    assign error_sum_c = kp_error_padded_c + ki_error_resize_c;

    //divide combination result to fit in output value then assign
    assign error_sum_trun_c = error_sum_c[SUM_WIDTH-1:SUM_WIDTH-DCO_CC_WIDTH];
    assign dco_cc_o = error_sum_trun_delay_r;

endmodule //LoopFilter
```

A.5 Ring Oscillator

```
`timescale 1ns / 1ps
/*****************************************************************************/
/* Author    : Conor Dooley                                                  */
```

```
/* Date      : ??-November-2019                                              */
/* Function : Inverter based oscillator. Propagation delay and control code */
/*            set the frequency of operation.                               */
/*****************************************************************************/
module RingOsc #(
        parameter RINGSIZE = 421, //number of inverters, must be even
        parameter CTRL_WIDTH = 5  //width of the control code
    )
    (
        input  wire                    enable_i, //enable high
        input  wire                    reset_i, // reset high
        input  wire [CTRL_WIDTH-1:0] freq_sel_i, //bigger is shorter period
        output wire                    early_clk_o, //phase leading version of
            generated clock
        output wire                    clk_o
    );

    /*****************************************************************************/
    /* Define nets and constants                                                 */
    /*****************************************************************************/

    //first "inverter" is actually a nand with enable signal
    localparam INVERTERNUM = RINGSIZE-1;

    //DONT_TOUCH ensures that what vivado deems to be "useless" logic will not
        be removed
    (* DONT_TOUCH = "TRUE" *) wire [0:INVERTERNUM] ringwire_c;

    reg f_sel_mux_out_r;

    /*****************************************************************************/
    /* Inverter chain                                                            */
    /*****************************************************************************/

    //generate statement used for ease of use, each inverter connected in a row
    genvar i;
    generate
        for (i = 1;i <= INVERTERNUM;i = i+1)
            begin: RING
                not inverter(ringwire_c[i], ringwire_c[i-1]);
        end
    endgenerate

    /*****************************************************************************/
    /* Combiational logic                                                        */
    /*****************************************************************************/

    //output selection multiplexor. 2* ensures odd number of inverters.
    always @(freq_sel_i,ringwire_c,reset_i)
    begin
        if (reset_i)
            f_sel_mux_out_r = 1'b0;
        else if (freq_sel_i == {(CTRL_WIDTH){1'b0}})
            f_sel_mux_out_r = ringwire_c[INVERTERNUM];
        else
            f_sel_mux_out_r = ringwire_c[INVERTERNUM-2*freq_sel_i];
    end

    //nand gate replaces first inverter for enable
    assign ringwire_c[0] = !((f_sel_mux_out_r) & (enable_i));

    //output bufferrs
    buf outbuf (clk_o, ringwire_c[INVERTERNUM]);
    //75 ensures early_clk_o edge sufficiently before clk_o edge
    buf outbuf_early (early_clk_o, ringwire_c[INVERTERNUM-2*CTRL_WIDTH-75]);

endmodule //RingOsc
```

A.6 FPGA Clocked Oscillator

```
/*****************************************************************************/
/* Author    : Conor Dooley                                                  */
/* Date      : ??-October-2019                                               */
/* Function  : Phase accumulator based oscillator with period set by the     */
/*             width and control code                                        */
/*****************************************************************************/

module PhaseAccum #(parameter WIDTH = 4)(
    input  wire enable_i,
    input  wire [WIDTH-1:0] k_val_i, //control code - bigger is high freq
    input  wire fpga_clk_i,
    input  wire reset_i, //reset high
    output wire clk_o
    );

    /*************************************************************************/
    /* Define nets                                                           */
    /*************************************************************************/

    reg [WIDTH-1:0] cnt_val_r, next_cnt_val_r;

    /*************************************************************************/
    /* Counter logic                                                         */
    /*************************************************************************/

    //increment count by control code if enable is high, otherwise do nothing
    always @(enable_i, k_val_i, cnt_val_r)
    begin
        if(enable_i == 1'b1) next_cnt_val_r = k_val_i + cnt_val_r;
        else                 next_cnt_val_r = cnt_val_r;
    end

    //counter register
    always @(posedge fpga_clk_i or posedge reset_i)
    begin
        if (reset_i) cnt_val_r <= {(WIDTH){1'b0}};
        else         cnt_val_r <= next_cnt_val_r;
    end

    //assign count MSB is the output
    assign clk_o = cnt_val_r[WIDTH-1];

endmodule
```

A.7 Network ADPLL (RO)

```
`timescale 1ns / 1ps
/*****************************************************************************/
/* Author    : Conor Dooley                                                  */
/* Date      : ??-January-2019                                               */
/* Function  : ADPLL using inverter based oscillator with dual phase detector */
/*             and other connections required for cartesian network operation */
/*****************************************************************************/
module NetworkRingADPLL #(
        parameter RO_WIDTH      = 5,            //control width of oscillator
        parameter PDET_WIDTH    = 5,            //width of phase detector
            output
        parameter RINGSIZE      = 401,          //default number of inverters
        parameter BIAS          = 5'd15,        //bias point for oscillator
        //LoopFilter
        parameter DYNAMIC_VAL   = 0,            //whether filter gains will
            chain at runtime
        parameter KP_WIDTH      = 6,            //width of kp
        parameter KP_FRAC_WIDTH = 5,            //fractional width of kp
        parameter KP            = 5'b01001,     //kp compile time value
        parameter KI_WIDTH      = 8,            //fractional width of ki
        parameter KI_FRAC_WIDTH = 7,            //width of ki
        parameter KI            = 8'b00000001,  //ki compile time value
        //ErrorCombiner
```

```
    parameter WEIGHT_WIDTH  = 4              //width of error combiner
        weights
)
(
    input  wire                             reset_i,          //reset high
    input  wire                             enable_i,
    input  wire                             fpga_clk_i,
    input  wire                             ref_left_i,       //reference clock
         from left neighbour
    input  wire                             ref_above_i,      //reference clock
         from above neighbour
    input  wire        [PDET_WIDTH-1:0]     error_right_i,    //error input
        from right neighbour
    input  wire        [PDET_WIDTH-1:0]     error_bottom_i,   //error input
        from bottom neighbour
    input  wire        [KP_WIDTH-1:0]       kp_i,             //kp runtime
        value
    input  wire        [KI_WIDTH-1:0]       ki_i,             //ki runtime
        value
    input  wire        [WEIGHT_WIDTH-1:0]   weight_left_i,    //sum weights
    input  wire        [WEIGHT_WIDTH-1:0]   weight_above_i,
    input  wire        [WEIGHT_WIDTH-1:0]   weight_right_i,
    input  wire        [WEIGHT_WIDTH-1:0]   weight_below_i,
    output wire                             gen_clk_o,        //clock output
    output wire                             gen_div8_o,       //divided clock
        output
    output wire signed [PDET_WIDTH-1:0]     error_left_o,     //error output
        from left detector
    output wire signed [PDET_WIDTH-1:0]     error_above_o,    //error output
        from above detector
    output wire signed [RO_WIDTH-1:0]       dco_cc_o          //dco control
        code
);

/*****************************************************************************/
/* Define nets and assign outputs                                            */
/*****************************************************************************/

wire signed [RO_WIDTH-1:0]    lf_out_x;
wire        [RO_WIDTH-1:0]    f_sel_sw_ro_x;
wire                          gen_clk_x;
wire                          gen_div_x;
wire                          early_clk_x;
wire                          early_div_x;
wire signed [PDET_WIDTH-1:0]  error_x;
wire signed [PDET_WIDTH-1:0]  error_left_x;
wire signed [PDET_WIDTH-1:0]  error_above_x;

/*****************************************************************************/
/* Assign outputs                                                            */
/*****************************************************************************/

assign dco_cc_o       = lf_out_x;
assign gen_clk_o      = gen_clk_x;
assign gen_div8_o     = gen_div_x;
assign error_left_o   = error_left_x;
assign error_above_o  = error_above_x;

/*****************************************************************************/
/* Module instantiations                                                     */
/*****************************************************************************/

(* DONT_TOUCH = "TRUE" *)  PhaseDetectorDL #(.WIDTH(PDET_WIDTH)) pDetLeft (
    .reset_i(reset_i),
    .fpga_clk_i(fpga_clk_i),
    .reference_i(ref_left_i),
    .generated_i(gen_div_x),
    .pd_clock_cycles_o(error_left_x)
);

(* DONT_TOUCH = "TRUE" *)  PhaseDetectorDL #(.WIDTH(PDET_WIDTH)) pDetAbove
    (
```

```
        .reset_i(reset_i),
        .fpga_clk_i(fpga_clk_i),
        .reference_i(ref_above_i),
        .generated_i(gen_div_x),
        .pd_clock_cycles_o(error_above_x)
    );

    (* DONT_TOUCH = "TRUE" *) ErrorCombiner #(
        .WEIGHT_WIDTH(WEIGHT_WIDTH),
        .ERROR_WIDTH(PDET_WIDTH)
    )
    errorCombiner
    (
        //zero out unconnected, 2 weight on others
        .reset_i(reset_i),
        .weight_0_i(weight_above_i),
        .weight_1_i(weight_left_i),
        .weight_2_i(weight_right_i),
        .weight_3_i(weight_below_i),
        .error_0_i(error_above_x),
        .error_1_i(error_left_x),
        .error_2_i(error_right_i),
        .error_3_i(error_bottom_i),
        .error_comb_o(error_x)
    );

    LoopFilter #(
        .ERROR_WIDTH(PDET_WIDTH),
        .DCO_CC_WIDTH(RO_WIDTH),
        .KP_WIDTH(KP_FRAC_WIDTH),
        .KP(KP),
        .KI_WIDTH(KI_FRAC_WIDTH),
        .KI(KI),
        .DYNAMIC_VAL(DYNAMIC_VAL)
    )
    loopFilter
    (
        .gen_clk_i(early_div_x),
        .reset_i(reset_i),
        .error_i(error_x),
        .kp_i(kp_i),
        .ki_i(ki_i),
        .dco_cc_o(lf_out_x)
    );

    RingOsc #(
        .RINGSIZE(RINGSIZE),
        .CTRL_WIDTH(RO_WIDTH)
    )
    testRing
    (
        .enable_i (enable_i),
        .reset_i (reset_i),
        .freq_sel_i (f_sel_sw_ro_x),
        .early_clk_o (early_clk_x),
        .clk_o (gen_clk_x)
    );

    Div8 div8 (
        .reset_i(reset_i),
        .signal_i(gen_clk_x),
        .div1_o(gen_div_x)
    );

    Div8 div8Early (
        .reset_i(reset_i),
        .signal_i(early_clk_x),
        .div1_o(early_div_x)
    );

    /*****************************************************************/
    /* Combinational logic                                           */
```

```
    /*****************************************************************/

    //apply bias to loop filter output
    assign f_sel_sw_ro_x = BIAS + lf_out_x;

endmodule // NetworkRingADPLL
```

A.8 Network ADPLL (FPGA Clocked)

```
`timescale 1ns / 1ps
/*****************************************************************************/
/* Author    : Conor Dooley                                                  */
/* Date      : ??-January-2019                                               */
/* Function : ADPLL using inverter based oscillator with dual phase detector */
/*            and other connections required for cartesian network operation */
/*****************************************************************************/
module NetworkRingADPLL #(
        parameter RO_WIDTH      = 5,            //control width of oscillator
        parameter PDET_WIDTH    = 5,            //width of phase detector
            output
        parameter RINGSIZE      = 401,          //default number of inverters
        parameter BIAS          = 5'd15,        //bias point for oscillator
        //LoopFilter
        parameter DYNAMIC_VAL   = 0,            //whether filter gains will
            chain at runtime
        parameter KP_WIDTH      = 6,            //width of kp
        parameter KP_FRAC_WIDTH = 5,            //fractional width of kp
        parameter KP            = 5'b01001,     //kp compile time value
        parameter KI_WIDTH      = 8,            //fractional width of ki
        parameter KI_FRAC_WIDTH = 7,            //width of ki
        parameter KI            = 8'b00000001,  //ki compile time value
        //ErrorCombiner
        parameter WEIGHT_WIDTH  = 4             //width of error combiner
            weights
    )
    (
        input  wire                             reset_i,        //reset high
        input  wire                             enable_i,
        input  wire                             fpga_clk_i,
        input  wire                             ref_left_i,     //reference clock
             from left neighbour
        input  wire                             ref_above_i,    //reference clock
             from above neighbour
        input  wire         [PDET_WIDTH-1:0]    error_right_i,  //error input
            from right neighbour
        input  wire         [PDET_WIDTH-1:0]    error_bottom_i, //error input
            from bottom neighbour
        input  wire         [KP_WIDTH-1:0]      kp_i,           //kp runtime
            value
        input  wire         [KI_WIDTH-1:0]      ki_i,           //ki runtime
            value
        input  wire         [WEIGHT_WIDTH-1:0]  weight_left_i,  //sum weights
        input  wire         [WEIGHT_WIDTH-1:0]  weight_above_i,
        input  wire         [WEIGHT_WIDTH-1:0]  weight_right_i,
        input  wire         [WEIGHT_WIDTH-1:0]  weight_below_i,
        output wire                             gen_clk_o,      //clock output
        output wire                             gen_div8_o,     //divided clock
            output
        output wire signed  [PDET_WIDTH-1:0]    error_left_o,   //error output
            from left detector
        output wire signed  [PDET_WIDTH-1:0]    error_above_o,  //error output
            from above detector
        output wire signed  [RO_WIDTH-1:0]      dco_cc_o        //dco control
            code
    );

    /*****************************************************************/
    /* Define nets and assign outputs                                */
    /*****************************************************************/
```

```
wire signed [RO_WIDTH-1:0]    lf_out_x;
wire        [RO_WIDTH-1:0]    f_sel_sw_ro_x;
wire                          gen_clk_x;
wire                          gen_div_x;
wire                          early_clk_x;
wire                          early_div_x;
wire signed [PDET_WIDTH-1:0] error_x;
wire signed [PDET_WIDTH-1:0] error_left_x;
wire signed [PDET_WIDTH-1:0] error_above_x;

/*****************************************************************************/
/* Assign outputs                                                            */
/*****************************************************************************/

assign dco_cc_o       = lf_out_x;
assign gen_clk_o      = gen_clk_x;
assign gen_div8_o     = gen_div_x;
assign error_left_o   = error_left_x;
assign error_above_o = error_above_x;

/*****************************************************************************/
/* Module instantiations                                                     */
/*****************************************************************************/

(* DONT_TOUCH = "TRUE" *)  PhaseDetectorDL #(.WIDTH(PDET_WIDTH)) pDetLeft (
    .reset_i(reset_i),
    .fpga_clk_i(fpga_clk_i),
    .reference_i(ref_left_i),
    .generated_i(gen_div_x),
    .pd_clock_cycles_o(error_left_x)
);

(* DONT_TOUCH = "TRUE" *)  PhaseDetectorDL #(.WIDTH(PDET_WIDTH)) pDetAbove
    (
    .reset_i(reset_i),
    .fpga_clk_i(fpga_clk_i),
    .reference_i(ref_above_i),
    .generated_i(gen_div_x),
    .pd_clock_cycles_o(error_above_x)
);

(* DONT_TOUCH = "TRUE" *)  ErrorCombiner #(
    .WEIGHT_WIDTH(WEIGHT_WIDTH),
    .ERROR_WIDTH(PDET_WIDTH)
)
errorCombiner
(
    //zero out unconnected, 2 weight on others
    .reset_i(reset_i),
    .weight_0_i(weight_above_i),
    .weight_1_i(weight_left_i),
    .weight_2_i(weight_right_i),
    .weight_3_i(weight_below_i),
    .error_0_i(error_above_x),
    .error_1_i(error_left_x),
    .error_2_i(error_right_i),
    .error_3_i(error_bottom_i),
    .error_comb_o(error_x)
);

LoopFilter #(
    .ERROR_WIDTH(PDET_WIDTH),
    .DCO_CC_WIDTH(RO_WIDTH),
    .KP_WIDTH(KP_FRAC_WIDTH),
    .KP(KP),
    .KI_WIDTH(KI_FRAC_WIDTH),
    .KI(KI),
    .DYNAMIC_VAL(DYNAMIC_VAL)
)
loopFilter
(
```

```
        .gen_clk_i(early_div_x),
        .reset_i(reset_i),
        .error_i(error_x),
        .kp_i(kp_i),
        .ki_i(ki_i),
        .dco_cc_o(lf_out_x)
    );

    RingOsc #(
        .RINGSIZE(RINGSIZE),
        .CTRL_WIDTH(RO_WIDTH)
    )
    testRing
    (
        .enable_i (enable_i),
        .reset_i (reset_i),
        .freq_sel_i (f_sel_sw_ro_x),
        .early_clk_o (early_clk_x),
        .clk_o (gen_clk_x)
    );

    Div8 div8 (
        .reset_i(reset_i),
        .signal_i(gen_clk_x),
        .div1_o(gen_div_x)
    );

    Div8 div8Early (
        .reset_i(reset_i),
        .signal_i(early_clk_x),
        .div1_o(early_div_x)
    );

    /*****************************************************************/
    /* Combinational logic                                           */
    /*****************************************************************/

    //apply bias to loop filter output
    assign f_sel_sw_ro_x = BIAS + lf_out_x;

endmodule // NetworkRingADPLL
```